Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Κατανεμημένη αποθήκευση και επερώτηση RDF δεδομένων, μεγάλου όγκου, με χρήση μεθοδολογιών NoSQL και MapReduce

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλαος Π. Παπαηλίου

Επιβλέπων : Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2011

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

Κατανεμημένη αποθήκευση και επερώτηση RDF δεδομένων, μεγάλου όγκου, με χρήση μεθοδολογιών NoSQL και MapReduce

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλαος Π. Παπαηλίου

Επιβλέπων : Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Οκτωβρίου 2011.

....	....	....
Νεκτάριος Κοζύρης Αναπληρωτής Καθηγητής Ε.Μ.Π.	Τιμολέων Σελλής Καθηγητής Ε.Μ.Π.	Παναγιώτης Τσανάκας Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2011

....
Νικόλαος Π. Παπαηλίου

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Νικόλαος Π. Παπαηλίου
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

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

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

Περίληψη

Τα τελευταία χρόνια γίνονται μεγάλες προσπάθειες για την υλοποίηση του στόχου του Semantic Web. Διεθνείς οργανισμοί έχουν ορίσει πρότυπα για όλες τις λειτουργίες που θα πρέπει να εκτελούνται. Βασικό πρότυπο για την αποθήκευση και μεταφορά των δεδομένων είναι το RDF. Σύμφωνα με το RDF τα δεδομένα αποθηκεύονται στην μορφή των triples, subject-predicate-object. Η SparQL είναι η βασική γλώσσα με την οποία μπορούμε να κάνουμε ερωτήσεις και να επεξεργαζόμαστε μια RDF βάση δεδομένων.

Το διαδίκτυο αναπτύσσεται συνεχώς και τα δεδομένα που περιέχονται σε αυτό αυξάνονται κάθε μέρα και περισσότερο. Αν θέλουμε να υλοποιήσουμε, λοιπόν, το στόχο του Semantic Web, πρέπει να δημιουργήσουμε συστήματα, τα οποία θα είναι σε θέση να χειριστούν το μεγάλο όγκο δεδομένων του διαδικτύου. Η εργασία μας στοχεύει στη δημιουργία ενός συστήματος αποθήκευσης και επερώτησης τέτοιων RDF δεδομένων, μεγάλου όγκου.

Σύγχρονη τάση, στις βάσεις δεδομένων, αποτελούν οι NoSQL βάσεις, οι οποίες δεν βασίζονται στη γλώσσα SQL και είναι κυρίως column stores. Η HBase είναι μια τέτοια βάση η οποία είναι κατανεμημένη και αποθηκεύει τα δεδομένα της ταυτόχρονα σε πολλούς υπολογιστές. Έρευνες έχουν δείξει ότι, η HBase μπορεί να αποθηκεύσει τεράστιους πίνακες και να έχει αποδοτική πρόσβαση σε αυτούς.

Το MapReduce είναι μια καινούργια τεχνική παραλληλοποίησης, που έχει κερδίσει τεράστιο έδαφος και χρησιμοποιείται, σε μεγάλο βαθμό, για την παραλληλοποίηση εργασιών.

Δημιουργήσαμε, λοιπόν, ένα σύστημα αποθήκευσης των RDF δεδομένων σε 3 διαφορετικά index της HBase. Τα 3 index μας επιτρέπουν να απαντάμε αποδοτικά σε όλους τους συνδυασμούς ερωτημάτων SparQL. Για την εκτέλεση των ερωτημάτων SparQL, χρησιμοποιήσαμε άπληστο αλγόριθμο επιλογής του πλάνου εκτέλεσης των join. Ακόμα, υλοποιήσαμε MapReduce προγράμματα για την κατανεμημένη εκτέλεση των SparQL join. Χρησιμοποιήσαμε το MapReduce για την εισαγωγή των RDF δεδομένων στα index της HBase. Τέλος, δείχνουμε ότι το σύστημά μας είναι κλιμακώσιμο και μπορεί να ανταποκριθεί στον μεγάλο όγκο των δεδομένων.

Λέξεις Κλειδιά

RDF, SparQL, Hadoop, MapReduce, Hbase, NoSQL, Jena, LUBM, Semantic Web

Abstract

Recently, researchers are making great efforts to achieve the objective of the Semantic Web. International organizations have set standards for all the needed functionality. Basic standard for storing and transporting data is RDF. According to RDF, data is stored in the form of triples, subject-predicate-object. SparQL is the basic query language for processing an RDF database.

Internet is growing continuously and the data contained in it, grow larger every day. Therefore, if we want to achieve the objective of Semantic Web, we must create systems that will be able to handle the large volume of Internet data. Our work aims to create a system for storing and querying, such, huge RDF datasets.

Modern trend in the databases are the NoSQL bases, which do not implement SQL language and are mainly distributed column stores. HBase is such a base, which is distributed and stores data on multiple computers simultaneously. Studies have shown that HBase can store huge tables and provides efficient access to them. MapReduce is a new parallelization technique that has gained enormous ground and is used largely for the parallelization of several tasks.

In this work, we created a system of storing RDF data in 3 different HBase indexes. The 3 index schema allows us to respond efficiently to all combinations of SparQL queries. To answer SparQL queries, we used a greedy algorithm for choosing the execution plan of joins. Furthermore, we implemented MapReduce jobs for distributed execution, of SparQL joins. We, also, used MapReduce jobs to insert the RDF data into the indexes of HBase. Finally, we show that our system is scalable and can meet the chalenge of huge RDF datasets.

KeyWords

RDF, SparQL, Hadoop, MapReduce, Hbase, NoSQL, Jena, LUBM, Semantic Web

Στην οικογένειά μου, που με υποστήριξε στη φοιτητική μου ζωή…

Πίνακας Περιεχομένων

1. **Εισαγωγή**……………………………………………………………………….. 12
2. **Θεωρητικό Υπόβαθρο**……………………………………………………….. 14
    2.1. RDF…………………………………………………………………………... 14
    2.2. SPARQL……………………………………………………………………… 17
    2.3. MapReduce…………………………………………………………………… 20
    2.4. Hadoop Framework…………………………………………………………... 22
        2.4.1. Εισαγωγή………………………………………………………………… 22
        2.4.2. Ανάλυση του συστήματος……………………………………………….. 23
        2.4.3. Hadoop Distributed File System (HDFS)……………………………….. 25
        2.4.4. Ανοχή σε σφάλματα……………………………………………………... 26
        2.4.5. Speculative execution…………………………………………………… 27
        2.4.6. Βασικά Interfaces του Hadoop………………………………………….. 27
    2.5. HBase………………………………………………………………………... 29
        2.5.1. Εισαγωγή………………………………………………………………... 29
        2.5.2. Μοντέλο δεδομένων…………………………………………………….. 29
        2.5.3. Επιλογές σχεδίασης πινάκων HBase……………………………………. 32
3. **Συσχετιζόμενες εργασίες**……………………………………………………… 34
    3.1. Αποθήκευση RDF-triple……………………………………………………... 34
    3.2. Εκτέλεση SPARQL ερωτημάτων με MapReduce……………………………. 35
4. **Σχεδίαση Συστήματος**………………………………………………………….. 37
    4.1. Μοντέλο αποθήκευσης δεδομένων…………………………………………... 37
        4.1.1. Αριθμός και τύπος index………………………………………………… 37
        4.1.2. Αποθήκευση των index σε πίνακες της HBase………………………….. 41
        4.1.3. S_PO index………………………………………………………………. 42
            4.1.3.1. Μοντέλο πίνακα…………………………………………………….. 42
            4.1.3.2. Κατανομή των δεδομένων στο cluster……………………………... 43
        4.1.4. SP_O index……………………………………………………………….. 44
            4.1.4.1. Μοντέλο πίνακα…………………………………………………….. 44
            4.1.4.2. Κατανομή των δεδομένων στο cluster……………………………... 45
    4.2. Σχεδιασμός εκτέλεσης MapReduce join…………………………………….. 47
        4.2.1. Εισαγωγή…………………………………………………………………. 47
        4.2.2. Προδιαγραφές εισόδου/εξόδου…………………………………………... 47
        4.2.3. FullInputJoin……………………………………………………………… 50
            4.2.3.1. Map…………………………………………………………………... 51
            4.2.3.2. Reduce……………………………………………………………….. 51
        4.2.4. SingleInputJoin…………………………………………………………… 52
            4.2.4.1. Map…………………………………………………………………... 53
            4.2.4.2. Reduce……………………………………………………………….. 54
        4.2.5. Συγκριση των αλγορίθμων εκτέλεσης των join………………………….. 55
        4.2.6. PartialInputJoin…………………………………………………………… 58
            4.2.6.1. Map…………………………………………………………………... 59
            4.2.6.2. Reduce……………………………………………………………….. 59
            4.2.6.3. Απόδοση του PartialInputJoin……………………………………….. 61
            4.2.6.4. Πρόσθετες δυνατότητες των αλγορίθμων εκτέλεσης των join………. 62
        4.2.7. Join Planner……………………………………………………………….. 63
            4.2.7.1. Εισαγωγή…………………………………………………………….. 63
            4.2.7.2. Μοντέλο κόστους εκτέλεσης join…………………………………… 63
            4.2.7.3. Αλγόριθμος επιλογής του βέλτιστου πλάνου εκτέλεσης των join…… 66

4.2.7.4. Άπληστος αλγόριθμος επιλογής του πλάνου εκτέλεσης των join…… 66
4.2.7.5. Αναλυτική περιγραφή του άπληστου αλγορίθμου επιλογης των join.. 69
5. **Υλοποίηση Συστήματος**…… 73
5.1. Εισαγωγή…... 73
5.2. Δημιουργία πινάκων index και εισαγωγή των δεδομένων με χρήση του MapReduce…… 73
5.2.1. Συμπίεση δεδομένων με χρήση hash function…… 74
5.2.2. Api Import vs Bulk Import...... 76
5.2.3. Map...... 78
5.2.4. Partitioner...... 83
5.2.5. Reduce...... 87
5.2.6. Συνδεση των Hfile με τους πίνακες Hbase...... 90
5.3. Εκτέλεση των join με χρήση MapReduce….. 90
5.3.1. InputFormat για εισαγωγή από πίνακες index…… 90
5.3.1.1. Προσδιορισμός δεδομένων εισόδου…… 91
5.3.1.2. Χωρισμός των δεδομένων σε Input Splits…… 91
5.3.1.3. Record Reader….. 93
5.3.2. Πέρασμα παραμέτρων στους mappers και reducers…… 94
5.3.3. Mapper…… 95
5.3.4. Reducer…… 95
5.4. Επεξεργασία SPARQL ερωτημάτων….. 97
5.4.1. Jena….. 97
6. **Πειραματικά αποτελέσματα**…..99
6.1. Δημιουργία των index…… 100
6.1.1. Μεταβλητός αριθμός πόρων…... 100
6.1.2. Μεταβλητός όγκος δεδομένων…... 102
6.2. Επεξεργασία SPARQL ερωτημάτων….. 103
6.3. Σύγκριση με άλλα συστήματα……106
7. **Συμπεράσματα**…..108

Πίνακας Σχημάτων

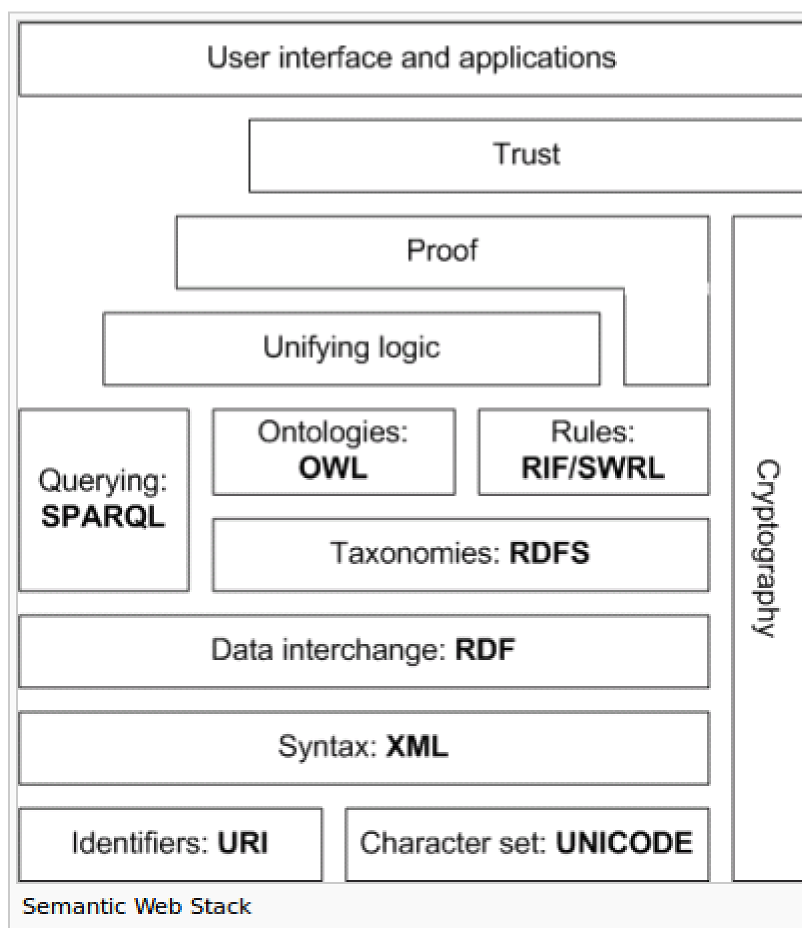
1. Semantic Web Stack…….. 12
2. RDF triples…… 14
3. Παράδειγμα RDF γράφου……. 15
4. Linked Data…….. 16
5. BGP query graph…….. 18
6. MapReduce…… 21
7. Hadoop architecture…….. 23
8. HDFS architecture……. 25
9. Index schema……. 38
10. Αρχιτεκτονική του συστήματος……. 73
11. Ολική διάταξη…… 77
12. Basic Partitions…….. 83
13. LUBM Partitions…… 84
14. Input Format…….. 91
15. Μεταβλητό μέγεθος πόρων……101
16. Μεταβλητός όγκος δεδομένων…...102

Πίνακας Πινάκων

1. Αντιστοιχία query pattern-index…… 39
2. Συμβολισμός query pattern…… 39
3. Παράδειγμα ανάκτησης δεδομένων BGP από index…… 40
4. Αντιστοιχία query pattern-index…… 41
5. S_PO index…… 42
6. S_PO index με offset…… 43
7. SP_O index…… 45
8. SP_O index με offset…… 46
9. Μορφή εξόδου join…… 49
10. Δεδομένα SP_O…… 49
11. Δεδομένα SP_O index…… 50
12. Δεδομένα PO_S index…… 51
13. Map output…… 51
14. Reduce input…… 52
15. Δεδομένα SP_O index…… 53
16. Δεδομένα PO_S index…… 53
17. Map output…… 54
18. Reduce input…… 54
19. Μέγεθος των BGP…… 55
20. Μέγεθος των BGP…… 56
21. Μέγεθος των BGP…… 57
22. PO_S index…… 58
23. Map output…… 59
24. Reduce input…… 60
25. Ενδιάμεσα δεδομένα reducer…… 60
26. Μέγεθος BGP…… 61
27. BGP variables…… 65
28. Μέγεθος BGP…… 70
29. SP_O index…… 70
30. PO_S index…… 70
31. Map output…… 80
32. Προσθήκη του offset…… 85
33. Reduce output…… 88
34. Αποτελέσματα import για μεταβλητό μέγεθος πόρων…… 101
35. Αποτελέσματα import για μεταβλητό όγκο δεδομένων…… 102

1) Εισαγωγή

Μια από τις σημαντικότερες προκλήσεις που υπάρχουν αυτή την εποχή στην τεχνολογία του διαδικτύου είναι η ιδέα του Semantic Web. Όπως ορίζεται από τον εμπνευστή του, Tim Berners-Lee, ιδρυτή του World Wide Web, το Semantic Web είναι "ένας ιστός δεδομένων τα οποία μπορούν να επεξεργαστούν άμεσα ή έμμεσα από μηχανές". Τα δεδομένα που βρίσκονται τοποθετημένα αυτή τη στιγμή στο διαδίκτυο είναι σε μορφή η οποία μπορεί να γίνει κατανοητή αποκλειστικά από τον άνθρωπο. Στόχος λοιπόν του Semantic Web είναι η δημιουργία όλων των απαραίτητων τεχνολογιών οι οποίες θα επιτρέψουν τη δυνατότητα κατανόησης και επεξεργασίας των δεδομένων του διαδικτύου από μηχανές. Για την επίτευξη αυτού του στόχου έχουν δημιουργηθεί διάφορα πρότυπα, τα οποία αποτελούν το Semantic Web Stack και παρουσιάζονται στο παρακάτω σχήμα.



Σχήμα 1: Semantic Web Stack

Στα πλαίσια αυτής της εργασίας θα ασχοληθούμε με τα πρότυπα SPARQL και RDF. Το πρότυπο RDF (Resource Description Framework) ορίζει ένα μοντέλο αναπαράστασης πληροφορίας για δεδομένα στο διαδίκτυο. Σύμφωνα με αυτό τα δεδομένα αποθηκεύονται με την μορφή subject-predicate-object, δηλαδή σαν εκφράσεις που περιέχουν υποκείμενο, ρηματική έκφραση και αντικείμενο. Για παράδειγμα αν θέλουμε να αποθηκεύσουμε την πληροφορία "Η Γη έχει σχήμα σφαιρικό" σε μορφή RDF, θα αποθηκεύσουμε ένα triple με υποκείμενο (subject) "Η Γη", ρηματική έκφραση (predicate) "έχει σχήμα" και αντικείμενο (object) "σφαιρικό".

Εφόσον τα δεδομένα αποθηκεύονται με τη μορφή RDF χρειαζόμαστε και μια

γλώσσα η οποία θα μας επιτρέπει την ανάκτηση, επεξεργασία και επερώτηση τέτοιου είδους δεδομένων. Για αυτό το σκοπό έχει δημιουργηθεί η πρότυπη γλώσσα SPARQL, η οποία έχει αναδειχτεί σαν ένα standard του Semantic Web Stack.

Όπως όλοι γνωρίζουμε τα τελευταία χρόνια παρατηρείται μια ραγδαία αύξηση του όγκου των δεδομένων που βρίσκονται στο διαδίκτυο. Αν θέλουμε λοιπόν να υλοποιήσουμε σε πλήρη βαθμό την ιδέα του Semantic Web θα πρέπει να αναπτύξουμε συστήματα τα οποία θα έχουν τη δυνατότητα να επεξεργάζονται τις τεράστιες αυτές ποσότητες δεδομένων σε αποδεκτό από τον τελικό χρήστη χρόνο. Όσον αφορά λοιπόν το πρόβλημα της αποθήκευσης και επερώτησης RDF δεδομένων έχουν παρουσιαστεί τον τελευταίο καιρό αρκετές υλοποιήσεις συστημάτων. Όπως δείχνει όμως και η πρόσφατα δημοσιευμένη μελέτη SP^2Bench[3] τα συστήματα αυτά δεν καταφέρνουν να ανταπεξέλθουν αποδοτικά στις προκλήσεις των τεράστιων σε όγκο δεδομένων που υπάρχουν στο διαδίκτυο.

Θα προσπαθήσουμε, λοιπόν, στα πλαίσια αυτής της εργασίας να ανταποκριθούμε σε αυτή τη σύγχρονη πρόκληση δημιουργώντας ένα κατανεμημένο σύστημα αποθήκευσης των RDF δεδομένων, καθώς και μια μέθοδο παραλληλοποίησης των SPARQL ερωτημάτων που θα μπορεί να κλιμακωθεί και να αντιμετωπίσει τα τεράστια σε όγκο δεδομένα.

Για το σκοπό της παραλληλοποίησης επιλέξαμε να χρησιμοποιήσουμε την τεχνική του MapReduce, μιας τεκμηριωμένης τεχνικής παραλληλοποίησης που προτάθηκε από την Google. Η τεχνική του MapReduce έχει χρησιμοποιηθεί σε πολλές επιστημονικές έρευνες καθώς και σε πρακτικές και εμπορικές εφαρμογές επιδεικνύοντας εξέχοντα αποτελέσματα. Για την υλοποίηση θα χρησιμοποιήσουμε ένα πακέτο ανοικτού κώδικα που ονομάζεται Hadoop και το οποίο διευκολύνει την ανάπτυξη εφαρμογών που χρησιμοποιούν την τεχνική του MapReduce. Χρησιμοποιώντας το Hadoop είναι εύκολο για ένα προγραμματιστή να στήσει ένα cluster(ομάδα) υπολογιστών, καθώς και να το χρησιμοποιήσει για να εκτελέσει παράλληλες εφαρμογές.

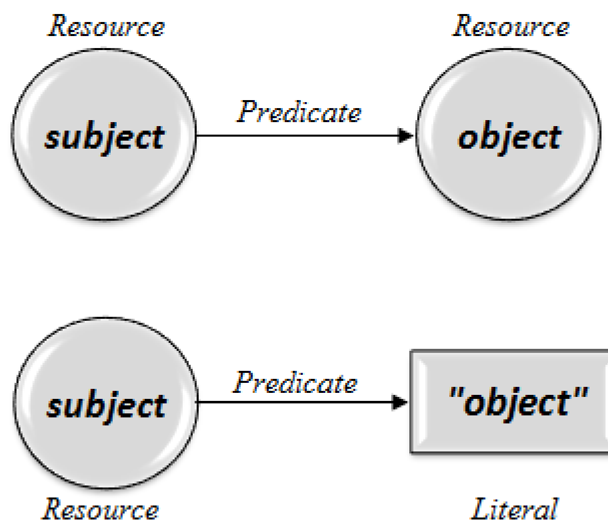
Όσον αφορά την κατανεμημένη αποθήκευση των RDF δεδομένων θα χρησιμοποιήσουμε την τεχνολογία των NoSQL βάσεων δεδομένων. Οι NoSQL βάσεις δεδομένων είναι μια καινούργια κατηγορία βάσεων δεδομένων οι οποίες διαφέρουν από τις κλασσικές σχεσιακές βάσεις δεδομένων αφού δεν παρέχουν διαπροσωπίες SQL. Οι NoSQL βάσεις δεδομένων έχουν δείξει ότι μπορούν να ανταπεξέλθουν σε μεγάλα φορτία read/write, καθώς και σε μεγάλο όγκο δεδομένων αποθήκευσης. Τέτοιες βάσεις έχουν χρησιμοποιηθεί σε πραγματικές εμπορικές εφαρμογές μεγάλης κλίμακας όπως είναι η βάση των 50TB του Facebook που χρησιμοποιείται για αναζήτηση των εισερχόμενων μηνυμάτων. Από της διάφορες υλοποιήσεις NoSQL βάσεων που υπάρχουν αυτή τη στιγμή επιλέξαμε την Hbase. Το πακέτο ανοικτού κώδικα της Hbase βασίζεται στο δημοσιευμένο άρθρο της Google, BigTable[4], και είναι μια column oriented βάση δεδομένων στην οποία αποθηκεύονται ζευγάρια key-value. Σημαντικό πλεονέκτημα της Hbase είναι και το γεγονός ότι βασίζεται στο πακέτο Hadoop που θα χρησιμοποιήσουμε για την παραλληλοποίηση. Τα δυο project έχουν δημιουργηθεί από την Apache και έχουν σχεδιαστεί ώστε να μπορούν να συνδυαστούν και να λειτουργούν αποδοτικά σε συνεργασία.

2) Θεωρητικό Υπόβαθρο

2.1) RDF

Όπως αναφέραμε και στην εισαγωγή το μοντέλο RDF χρησιμοποιείται για να εκφράσει απλές προτάσεις στη μορφη subject-predicate-object. Η μορφή αυτή αναπαράστασης των δεδομένων σε RDF triples είναι ιδιαίτερα χρήσιμη γιατί κάνει εύκολη την παρουσίαση των δεδομένων με μορφή γράφου. Ένα RDF triple αποτελεί το στοιχειώδες κομμάτι ενός RDF γράφου και αποτελείται από δυο κόμβους που συνδέονται με μια ακμή. Οι δυο αυτοί κόμβοι αντιστοιχούν στα subject και object ενώ η ακμή αντιστοιχεί στο predicate.

Το RDF έχει σχεδιαστεί για μοντελοποίηση δεδομένων του διαδικτύου, και γι' αυτό χρησιμοποιούνται αναφορές σε μορφή URI για την περιγραφή των δεδομένων. Βέβαια σε περιπτώσεις που θέλουμε να αναφερθούμε σε μια πολύ συγκεκριμένη ιδιότητα, μπορούμε να χρησιμοποιήσουμε και λέξεις για την περιγραφή της. Οι περιγραφές που έχουν μορφή URI ονομάζονται resources και αναπαριστώνται με ελλείψεις στους RDF γράφους, ενώ αυτές που έχουν μορφή string ονομάζονται literals και αναπαριστώνται με παραλληλόγραμμα. Στο παρακάτω σχήμα φαίνονται οι δυο εκδοχές αναπαράστασης ενός triple σε RDF γράφο.

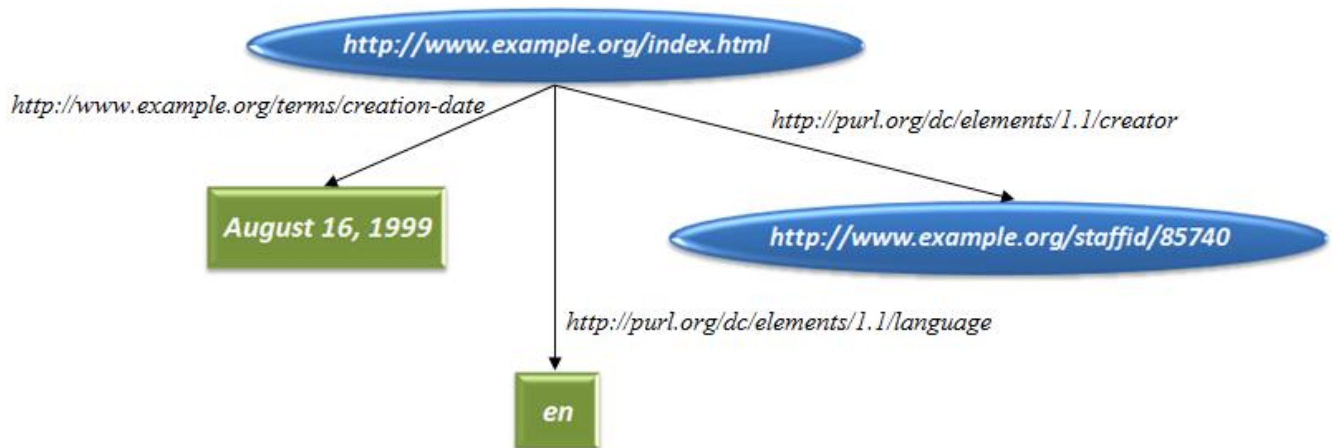


Σχήμα 2: RDF triples

Ένα σύνολο, λοιπόν, δεδομένων RDF αποτελούν ένα γράφο. Για παράδειγμα οι παρακάτω προτάσεις

http://www.example.org/index.html has a **creator** whose value is **John Smith.**
http://www.example.org/index.html has a **creation-date August 16, 1999**.
http://www.example.org/index.html has a **language** whose value is **English.**

μπορούν να εκφραστούν με τον RDF γράφο του παρακάτω σχήματος.



Σχήμα 3: Παράδειγμα RDF γράφου

Για την αναπαράσταση των RDF δεδομένων συνήθως χρησιμοποιείται η μορφή N-Triples η οποία φαίνεται παρακάτω:

<http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/creator>
<http://www.example.org/staffid/85740> .
<http://www.example.org/index.html> <http://www.example.org/terms/creation-date>
"August 16, 1999" .
<http://www.example.org/index.html> <http://purl.org/dc/elements/1.1/language> "en" .

Σύμφωνα με τη μορφή N-Triples τα δεδομένα αποθηκεύονται σε ένα αρχείο κειμένου, κάθε γραμμή του οποίου παριστάνει και ένα RDF triple. Στην κάθε γραμμή τοποθετούνται με τη σειρά τα subject, predicate και object χωρισμένα με κενά. Αν χρησιμοποιείται URI αναπαράσταση τοποθετούμε ολόκληρο το URI μεσα σε <>, ενώ αν έχουμε αναπαράσταση σε μορφή literal γράφουμε το string μέσα σε “”. Στο τέλος κάθε γραμμής υπάρχει μια τελεία η οποία δηλώνει το τέλος του triple.

Η σημαντικότερη ιδιότητα της μορφής αναπαράστασης σε triples είναι το γεγονός ότι είναι ισοδύναμη με την αναπαράσταση με χρήση γράφων. Αυτό σημαίνει ότι κάθε γράφος μπορεί να αναπαρασταθεί με τη μορφή αυτή και ταυτόχρονα οποιοδήποτε σύνολο από triples αποτελεί ένα γράφο.

Όπως αναφέρθηκε και στον ορισμό του N-Triples για την αναπαράσταση των δεδομένων σε μορφή URI χρησιμοποιούμε κάθε φορά το πλήρες URI reference. Για ευκολία, καθώς και για μείωση του όγκου των δεδομένων μπορούμε να χρησιμοποιήσουμε μια συντόμευση που ονομάζεται QName και δεν τοποθετείται μέσα σε <>. Ένα QName περιέχει το πρόθεμα (prefix) που είναι περιλαμβάνει το URI μιας συγκεκριμένης περιοχής ονομάτων και το τοπικό όνομα (local name) του δεδομένου, στην συγκεκριμένη περιοχή. Το prefix και το local name χωρίζονται από ένα χαρακτήρα “:”. Έτσι για παράδειγμα αν έχουμε ορίσει το prefix foo να αντιστοιχεί στο http://example.org/somewhere/ το QName foo:bar είναι μια συντομογραφία του http://example.org/somewhere/bar.

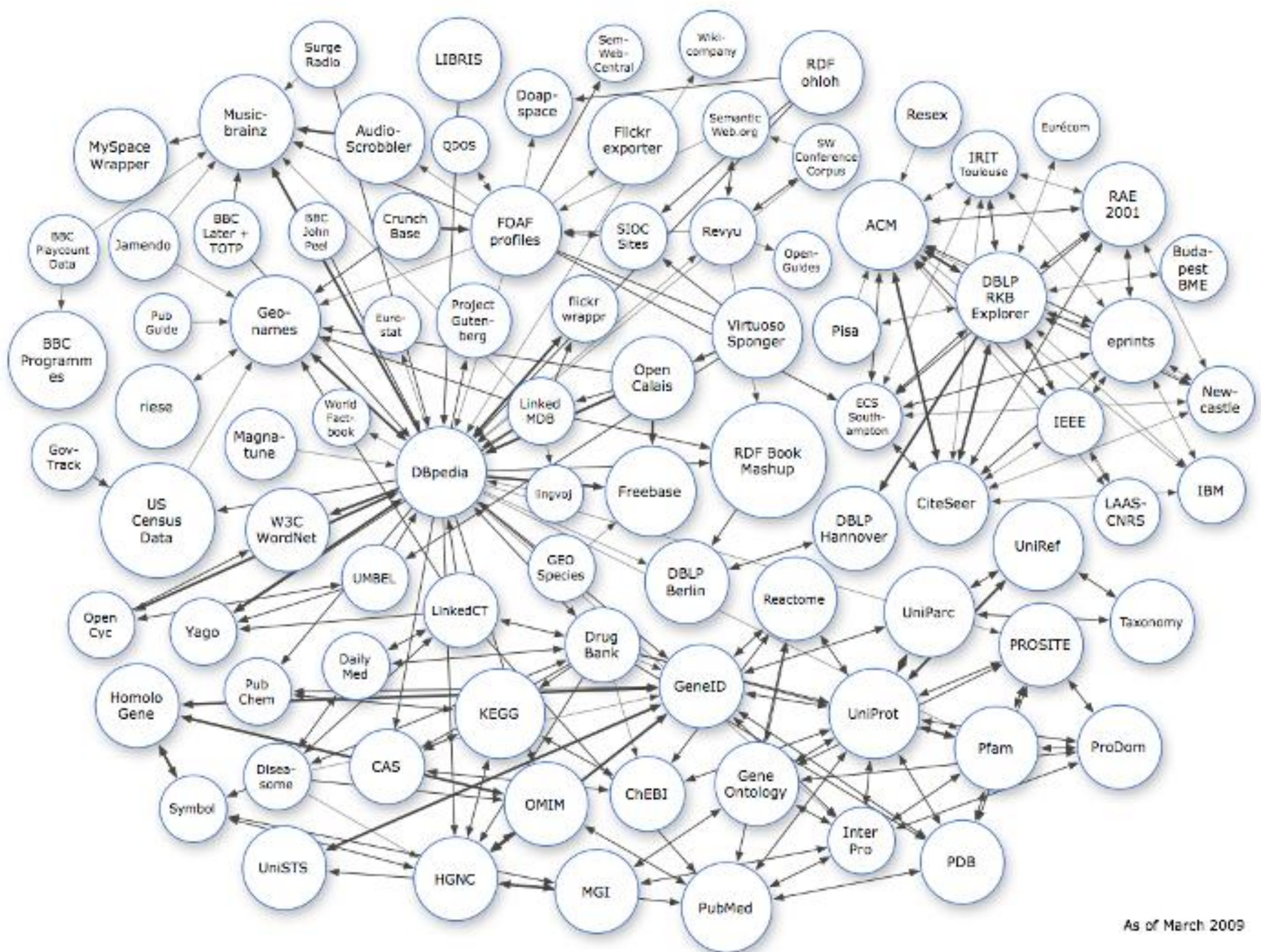
Υπάρχουν πολλοί φορείς στο διαδίκτυο οι οποίοι έχουν δημιουργήσει τέτοια namespace με στόχο την καταγραφή όλων των οντοτήτων που υπάρχουν στο διαδίκτυο. Υπάρχουν ακόμα και κάποια παγκοσμίως κατοχυρωμένα prefix, τα οποία δεν χρειάζεται να τα ορίζουμε κάθε φορά. Τέτοια prefix και τα αντίστοιχα URI τους είναι τα παρακάτω:

rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs: http://www.w3.org/2000/01/rdf-schema#
dc: http://purl.org/dc/elements/1.1/

owl: http://www.w3.org/2002/07/owl#
ex: http://www.example.org/ (or http://www.example.com/)
xsd: http://www.w3.org/2001/XMLSchema#

Αυτά τα σύνολα URI αναφορών που δημιουργούνται από διάφορους φορείς ονομάζονται λεξιλόγια (vocabulary). Για να ορίσουν το λεξιλόγιό τους οι οργανισμοί χρησιμοποιούν ένα κοινό prefix κάτω από το οποίο τοποθετούν όλους τους απαραίτητους όρους. Για παράδειγμα ένας οργανισμός σαν τον example.org δημιουργεί ένα prefix http://www.example.or/staffid/ το οποίο χρησιμοποιεί για να περιγράψει όλους τους υπαλλήλους της. Ακόμα και το rdf χρησιμοποιεί την ίδια τεχνική για να ορίσει το δικό του λεξιλόγιο που αποτελείται από τα prefix rdf και rdfs. Στο σχήμα 2 φαίνονται οι οργανισμοί που έχουν δημιουργήσει τέτοια λεξιλόγια, καθώς και πως αυτοί συνδέονται μεταξύ τους.

Η παραπάνω χρήση λεξιλογίων παρουσιάζει ένα από τα πλεονεκτήματα της χρήσης URI αναφορών για την περιγραφή των δεδομένων. Στο αρχικό μας παράδειγμα ο δημιουργός της σελίδας “John Smith” δεν αποθηκεύτηκε σαν ένα literal αλλά σαν μια URI αναφορά η οποία περιείχε τον κωδικό του συγκεκριμένου υπαλλήλου. Με τη χρήση αυτής της URI αναφοράς καταφέρνουμε να διαχωρίσουμε τον συγκεκριμένο “John Smith” από οποιονδήποτε άλλο άνθρωπο με το ίδιο όνομα. Ακόμα αφού χρησιμοποιείται μια URI αναφορά, η οποία είναι πλήρως καθορισμένη, μπορούμε να αποθηκεύσουμε και επιπλέον πληροφορίες για τον John Smith χρησιμοποιώντας το συγκεκριμένο URI σαν subject.



Σχήμα 4: Linked Data

Στο πλαίσιο RDF χρησιμοποιούνται μόνο URI αναφορές για τον ορισμό των predicates. Ο αυστηρός αυτός προσδιορισμός των predicates προσφέρει αρκετά πλεονεκτήματα σε σχέση με την χρήση απλών εκφράσεων τύπου string. Ο στόχος του Semantic Web είναι να συνδιάσει όλα τα δεδομένα στο διαδίκτυο, άρα χρησιμοποιώντας αυστηρούς ορισμούς των predicates διαχωρίζουμε τις ιδιότητες που παρουσιάζει ένας χρήστης από αυτές που παρουσιάζει κάποιος άλλο. Οι ιδιότητες αυτές μπορεί να προσδιορίζονταν από το ίδιο string δημιουργώντας παρεξηγήσεις. Επιπλέον, η χρήση URI αναφορών για τα predicate επιτρέπει την περεταίρω ανάλυσή τους, προσθέτοντας RDF triples που έχουν ως subject το συγκεκριμένο URI. Τέλος η χρήση URI αναφορών υποστηρίζει την εξέλιξη και τη χρήση κοινών λεξιλογίων. Μπορεί, δηλαδή ένας οργανισμός να χρησιμοποίηση ως predicate μια ιδιότητα που ορίζεται από έναν άλλον χωρίς να χρειάζεται να δημιουργήσει τη δικιά του. Αυτό βοηθάει στην δημιουργία παγκοσμίως αναγνωρισμένων οντοτήτων, οι οποίες συμβάλλουν στην συνένωση των δεδομένων του διαδικτύου.

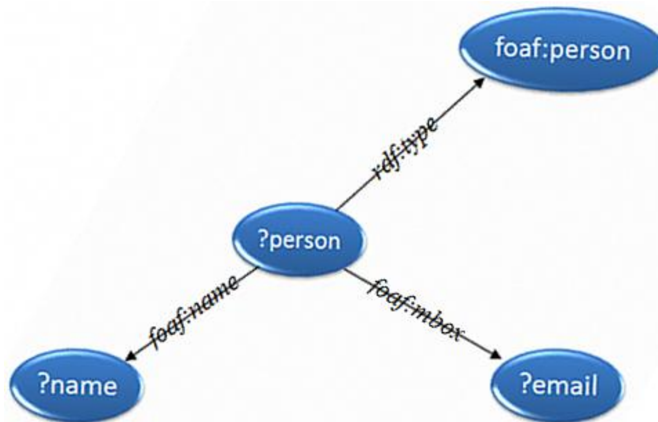
Το αποτέλεσμα όλων αυτών είναι η δημιουργία μιας κοινά αποδεκτής και συνεχώς επεκτάσιμης βάσης δεδομένων που περιέχει όλες τις πληροφορίες του διαδικτύου. Η βάση αυτή περιέχει εκφράσεις οι οποίες είναι εύκολα κατανοητές από τις εφαρμογές που θα κληθούν να τις επεξεργαστούν.

2.2) SPARQL

Το όνομα της SPARQL προέρχεται από τα αρχικά της φράσης “SPARQL Protocol and RDF Query Language”, και αποτελεί μια γλώσσα που χρησιμοποιείται για επερώτηση RDF δεδομένων. Όπως αναφέραμε και παραπάνω μια RDF βάση δεδομένων μπορεί να παρουσιαστεί σαν ένας γράφος. Η βασική ιδέα λοιπόν, στην οποία βασίζεται η SPARQL, για την διεξαγωγή των ερωτημάτων είναι το ταίριασμα υπογράφων. Κάθε ερώτημα SPARQL μπορεί να παρασταθεί και αυτό με ένα γράφο τα ονόματα των ακμών και των κορυφών του οποίου μπορεί να είναι σταθερά ή μεταβλητά. Ένας τέτοιος γράφος για παράδειγμα μπορεί να έχει την εξής μορφή.

```
?person rdf:type foaf:Person.
?person foaf:name ?name.
?person foaf:mbox ?email.
```

Τα ονόματα που αρχίζουν με “?” Θεωρούνται μεταβλητές και μπορούν να ταιριάξουν με οποιοδήποτε resource ή literal. Η SPARQL, δηλαδή προσπαθεί να ταιριάξει τον παρακάτω γράφο στο συνολικό γράφο των δεδομένων. Το αποτέλεσμα του ερωτήματος είναι όλα τα δυνατά URI ή string που ταιριάζουν σε κάθε μεταβλητή.



Σχήμα 5: BGP query graph

Οι γράφοι των ερωτημάτων εκφράζονται όπως φάνηκε παραπάνω με τα αντίστοιχα triples. Τα triples αυτά περιέχουν μεταβλητές, ονομάζονται BGP (Basic Graph Patterns) και αποτελούν τη βασική μονάδα δημιουργίας των SPARQL ερωτημάτων. Η ολοκληρωμένη μορφή του ερωτήματος φαίνεται παρακάτω:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name ?email
WHERE {
  ?person rdf:type foaf:Person.
  ?person foaf:name ?name.
  ?person foaf:mbox ?email.
}
```

Στην αρχή του ερωτήματος υπάρχει το τμήμα PREFIX, στο οποίο ορίζουμε τα URI όλων των prefix που θα χρησιμοποιήσουμε για τη δημιουργία του ερωτήματος. Η χρήση των prefix γίνεται με σκοπό τη μείωση του μεγέθους των ερωτημάτων και την καλύτερη κατανόησή τους. Το δεύτερο τμήμα του ερωτήματος δηλώνει ποιό είναι το είδος του query που θα εκτελεστεί. Τέλος υπάρχει το τμήμα WHERE στο οποίο προσδιορίζεται ο γράφος ο οποίος θα χρησιμοποιηθεί στην εκτέλεση του ερωτήματος.

Τα είδη των query που μπορούμε να δημιουργήσουμε με χρήση της SPARQL είναι:

SELECT: Ταιριάζει τον γράφο που δίνεται στο τμήμα WHERE με τον συνολικό γράφο της βάσης μας και επιστρέφει τις τιμές που λαμβάνουν οι μεταβλητές σε κάθε ταίριασμα. Δεν επιστρέφονται οι τιμές όλων των μεταβλητών αλλά μόνο αυτές που αναφέρονται μετά την λέξη SELECT. Αν μετά τη λέξη SELECT τοποθετήσουμε τον χαρακτήρα "*" τότε επιλέγονται όλες οι μεταβλητές.

CONSTRUCT: Αυτό το είδος ερωτήματος χρησιμοποιείται για την προσθήκη επιπλέον πληροφορίας στη βάση μας. Στο συγκεκριμένο ερώτημα πρέπει να δηλώσουμε 2 BGP γράφους, έναν στο τμήμα WHERE και ένα μετά την λέξη CONSTRUCT. Αρχικά ταιριάζει ο γράφος του τμήματος WHERE με τη βάση μας και βρίσκονται όλες οι τιμές των μεταβλητών. Για κάθε τέτοιο ταίριασμα, προστίθεται στη βάση μας ο γράφος του τμήματος CONSTRUCT με τιμές των μεταβλητών τις τιμές που λαμβάνονται από το ταίριασμα. Για παράδειγμα με το παρακάτω ερώτημα βρίσκουμε όλα τα ονόματα που περιέχονται στη βάση μας και για κάθε όνομα προσθέτουμε στη βάση το triple <http://example.org/person#Alice> vcard:FN ?name.

```
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT   { <http://example.org/person#Alice> vcard:FN ?name }
WHERE       { ?x foaf:name ?name }
```

ASK: Αυτό το ερώτημα αποτελεί μια πιο απλή μορφή του SELECT. Σε αυτή την περίπτωση δεν ζητάμε να μας επιστραφούν οι τιμές των μεταβλητών. Μας επιστρέφεται μια τιμή τύπου boolean η οποία δηλώνει αν ο δοσμένος γράφος ταιριάζει ή όχι στον γράφο της βάσης μας.

DESCRIBE: Η πιο απλή μορφή αυτού του ερωτήματος φαίνεται παρακάτω και επιστρέφει όλα τα triples που περιέχονται στη βάση μας και περιέχουν το IRI http://example.org/ σε κάποιο resource τους.

```
DESCRIBE <http://example.org/>
```

Αντί να δηλώσουμε άμεσα το URI που επιθυμούμε να ψάξουμε, μπορούμε να χρησιμοποιούμε μεταβλητές οι οποίες λαμβάνουν τις τιμές τους από το ταίριασμα του γράφου του τμήματος WHERE. Ένα τέτοιο παράδειγμα είναι:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
DESCRIBE ?x
WHERE    { ?x foaf:name "Alice" }
```

Αφού η SPARQL βασίζεται στο ταίριασμα RDF γράφων, για τη δημιουργία πιο σύνθετων ερωτημάτων υπάρχουν τρόποι ορισμού σύνθετων προτύπων γράφων. Τέτοιοι ορισμοί γίνονται συνδιάζοντας μικρότερους γράφους με διάφορους τρόπους, όπως:

- **Basic Graph Patterns**: ο βασικός τρόπος ορισμού γράφων με χρήση ενός συνόλου από triples
- **Group Graph Pattern**: ένα σύνολο από επιμέρους γράφους οι οποίοι πρέπει όλοι να ταιριάζουν ταυτόχρονα στη βάση μας
- **Optional Graph Pattern**: ένας προαιρετικός γράφος που μπορεί να ταιριάζει ή όχι ανάλογα με τις τιμές των μεταβλητών του κύριου γράφου.
- **Alternative Graph Pattern**: χρησιμοποιούνται δυο ή περισσότεροι εναλλακτικοί γράφοι.

Παρακάτω δίνονται ένα παράδειγμα για optional και ένα για alternative graph pattern:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox ?hpage
WHERE  { ?x foaf:name  ?name .
       OPTIONAL { ?x foaf:mbox ?mbox } .
       OPTIONAL { ?x foaf:homepage ?hpage }
     }
```

Εδώ χρησιμοποιείται το optional graph pattern αφού ζητάμε τα ονόματα όλων των ανθρώπων στη βάση και επιπλέον πληροφορία όπως το mail και το homepage που μπορεί να μην είναι διαθέσιμη για όλους. Αυτό το ερώτημα θα μας επιστρέψει ακόμα και τα ονόματα αυτών που δεν έχουν ούτε mail ούτε homepage.

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc11:  <http://purl.org/dc/elements/1.1/>
SELECT ?x ?y
WHERE  { { ?book dc10:title ?x } UNION { ?book dc11:title  ?y } }
```

Εδώ φαίνεται το alternative graph pattern και χρησιμοποιείται για να παρουσιάσει τα βιβλία που ορίζονται είτε από το dc10 είτε από το dc11 prefix.

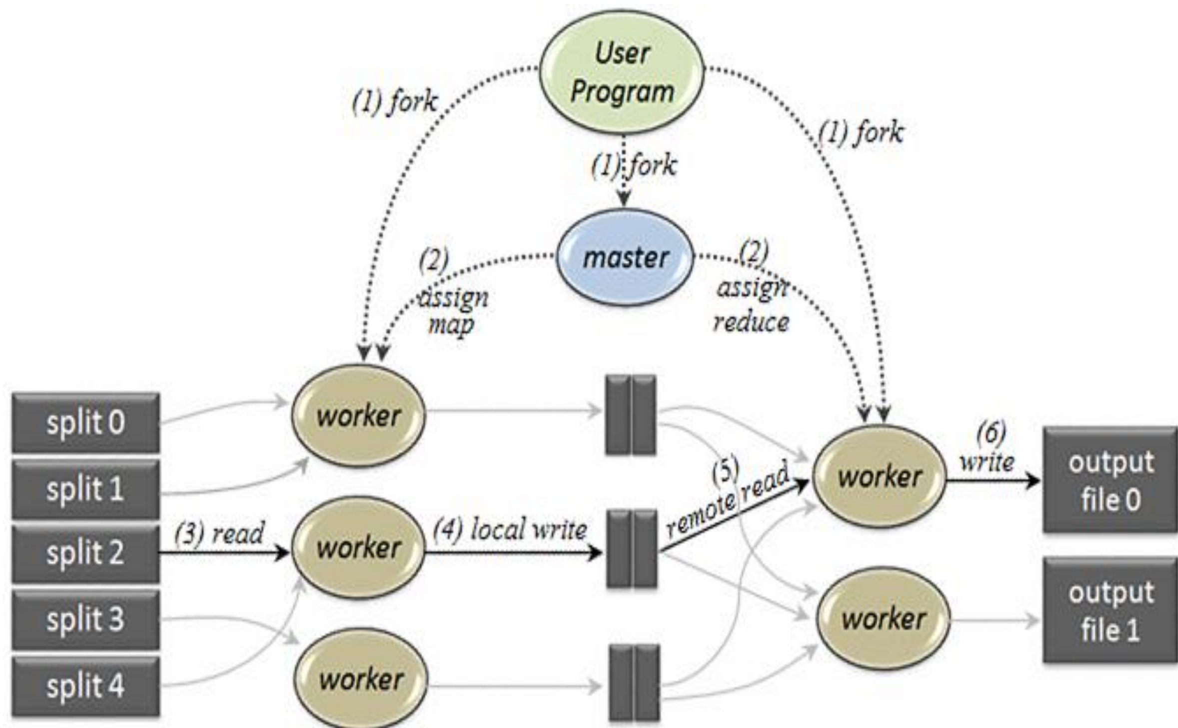
2.3) MapReduce

Το MapReduce είναι ένα προγραμματιστικό πλαίσιο που παρουσιάστηκε από την Google και διευκολύνει την κατανεμημένη επεξεργασία τεράστιων ποσοτήτων δεδομένων, χρησιμοποιώντας ένα μεγάλο αριθμό από υπολογιστές. Οι υπολογιστές που χρησιμοποιούνται για την παράλληλη αυτή επεξεργασία είναι συνδεδεμένοι μεταξύ τους μέσω δικτύου και αποτελούν το λεγόμενο cluster υπολογιστών. Η ιδέα αυτή βασίζεται στις γνωστές από το συναρτησιακό προγραμματισμό, συναρτήσεις Map και Reduce οι οποίες αποτελούν ένα μοντέλο προγραμματισμού με το οποίο μπορούμε να εκφράσουμε πολλές από τις σημερινές πραγματικές εφαρμογές. Ο προγραμματιστής πρέπει απλά να ορίσει μια συνάρτηση Map και μια συνάρτηση Reduce και το σύστημα αναλαμβάνει αυτόματα την παραλληλοποίηση και την εκτέλεση της εφαρμογής σε ένα οσοδήποτε μεγάλο cluster από υπολογιστές.

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

Συνάρτηση Reduce: Η συνάρτηση Reduce, που ορίζεται και αυτή από τον προγραμματιστή, δέχεται ένα από τα ενδιάμεσα κλειδιά και ένα σετ με όλες τις τιμές 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 παράγει τα αποτελέσματά της χωρίς να χρειάζεται να επικοινωνήσει με καμία άλλη. Αφού όλες οι διεργασίες είναι ανεξάρτητες μεταξύ τους μπορούν να εκτελεστούν όλες ταυτόχρονα. Αυτό σημαίνει ότι αποφεύγονται τα γνωστά προβλήματα επικοινωνίας που εμφανίζονται σε άλλα είδη παράλληλου προγραμματισμού όταν αυξάνεται ο αριθμός των παράλληλων διεργασιών. Στο βαθμό λοιπόν που υπάρχουν αρκετά δεδομένα εισόδου μπορούμε να μειώσουμε το χρόνο εκτέλεσης σχεδόν γραμμικά σε σχέση με τον αριθμό των υπολογιστών στο cluster μας.



Σχήμα 6: MapReduce

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

1) Η βιβλιοθήκη του MapReduce αναλαμβάνει να χωρίσει τα δεδομένα εισόδου σε M κομμάτια των οποίων το μέγεθος μπορεί να ορισθεί από το χρήστη μέσω μιας παραμέτρου. Ο προγραμματιστής μπορεί ακόμα και να φτιάξει τη δική του συνάρτηση η οποία χωρίζει τα δεδομένα εισόδου με τον τρόπο που θεωρεί αυτός κατάλληλο. Στη συνέχεια δημιουργούνται πολλά αντίγραφα του προγράμματος τα οποία τρέχουν σε όλα τα μηχανήματα του cluster.

2) Ένα από τα αντίγραφα του προγράμματος είναι ξεχωριστό και ονομάζεται master. Τα υπόλοιπα αντίγραφα ονομάζονται workers και ο master αναλαμβάνει να αναθέσει στον κάθε worker την κατάλληλη διεργασία. Υπάρχουν M map και R reduce διεργασίες που πρέπει να ανατεθούν για την εκτέλεση ενός MapReduce job. Ο master λοιπόν εντοπίζει τους ανενεργούς workers και τους αναθέτει την κατάλληλη διεργασία.

3) Όταν ένας εργάτης αναλάβει μια διεργασία map διαβάζει το αντίστοιχο κομμάτι δεδομένων (input split) και χρησιμοποιεί μια συνάρτηση ώστε να διαβάσει τα δεδομένα εισόδου και να παράγει τα κατάλληλα ζευγάρια key/value. Η συνάρτηση αυτή μπορεί να δημιουργηθεί από τον προγραμματιστή ή μπορεί να χρησιμοποιηθεί μια από τις συναρτήσεις που υπάρχουν διαθέσιμες (π.χ. για ανάγνωση ενός αρχείου κειμένου, με δημιουργία ζευγαριών key/value για κάθε γραμμή κειμένου). 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) Σε 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 ή 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 με βάση το κλειδί του κάθε ζευγαριού. Μπορούμε να χρησιμοποιήσουμε την προεπιλεγμένη συνάρτηση διαχωρισμού η οποία είναι hash(key) mod R ή να ορίσουμε την δικιά μας συνάρτηση. Βασικός περιορισμός της συνάρτησης

διαχωρισμού είναι ότι πρέπει όλα τα ζευγάρια με το ίδιο κλειδί να τοποθετούνται στην ίδια περιοχή. Αφού αποθηκευτούν τα ζευγάρια στο δίσκο του μηχανήματος ο worker στέλνει τις θέσεις τους στον master ο οποίος είναι υπεύθυνος να προωθήσει της θέσεις αυτές στους reducers.

5) Ο reducer ενημερώνεται από τον master για τις θέσεις στις οποίες βρίσκονται τα ζευγάρια key/value που πρέπει να επεξεργαστεί και χρησιμοποιεί remote procedure calls για να διαβάσει αυτά τα δέδομένα από τους τοπικούς δίσκους των mappers. Όταν διαβάσει όλα τα ζευγάρια εισόδου ο reducer τα ταξινομεί με βάση το κλειδί του κάθε ζευγαριού. Έτσι όλα τα ζευγάρια με το ίδιο κλειδί ομαδοποιούνται μαζί, έτσι ώστε για κάθε κλειδί ο reducer να επεξεργάζεται όλες τις τιμές ταυτόχρονα.

6) Ο reducer επεξεργάζεται με τη σειρά όλα τα ενδιάμεσα ζευγάρια key/value 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 εκτελεστούν όλες οι διεργασίες map και reduce 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 (ένα για κάθε περιοχή). Αυτά τα αρχεία μπορούν να ενωθούν σε ένα ενιαίο αρχείο ή να χρησιμοποιηθούν κατευθείαν σαν είσοδος για άλλες κατανεμημένες εργασίες.

Ένας πιο αυστηρός ορισμός των συναρτήσεων map και reduce με βάση την είσοδο και την έξοδό τους δίνεται παρακάτω:

```
Map(k1,v1) → list(k2,v2)
Reduce(k2, list (v2)) →list(v3)
```

Από τους παραπάνω τύπου συμπεραίνουμε ότι η συνάρτηση map εφαρμόζεται πάνω σε ένα ζευγάρι key/value συγκεκριμένου τύπου και παράγει μια λίστα από ζευγάρια key/value διαφορετικού τύπου. Για κάθε τιμή κλειδί έχουμε μια συνάρτηση 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 μπορούν να έχουν διαφορετικό τύπο από τις τιμές εισόδου.

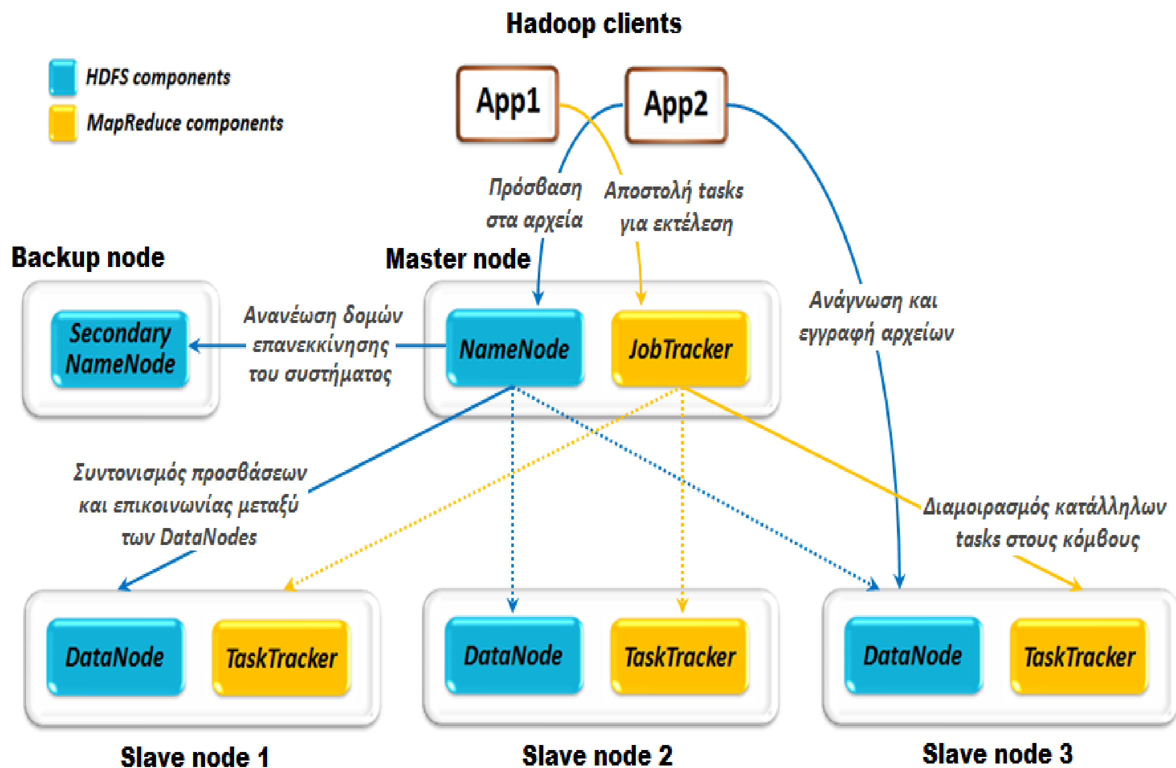
2.4) Hadoop Framework

2.4.1) Εισαγωγή

Το Hadoop είναι ένα προγραμματιστικό πλαίσιο το οποίο υλοποιεί την ιδέα του MapReduce και χρησιμοποιείται για παραλληλοποίηση εφαρμογών με μεγάλο σε όγκο δεδομένα. Το Hadoop είναι ένα από τα κύρια έργα της εταιρίας Apache και είναι ένα πακέτο ανοικτού κώδικα, γεγονός που δίνει μεγάλη ελευθερία παραμετροποίησης στον προγραμματιστή. Το πακέτο αυτό επιτρέπει την εύκολη δημιουργία μεγάλων cluster υπολογιστών και αναλαμβάνει όλες τις απαραίτητες ενέργειες για την εκτέλεση εργασιών MapReduce. Ο προγραμματιστής αρκεί να υλοποιήσει τις κατάλληλες συναρτήσεις map και reduce ώστε να παραλληλοποιήσει την εφαρμογή του. Η εφαρμογή αυτή στη συνέχεια μπορεί να τρέξει σε οποιοδήποτε cluster μηχανημάτων ανεξάρτητα από τον αριθμό των κόμβων του και την υπολογιστική δύναμη κάθε κόμβου. Ακόμα το Hadoop έχει μεγάλη ανοχή στα σφάλματα υλικού, δηλαδή ακόμα και αν κάποιος υπολογιστής έχει πρόβλημα δεν θα διακοπεί η εκτέλεση του προγράμματος αλλά η δουλειά που εκτελούσε θα μοιραστεί σε άλλους διαθέσιμους κόμβους.

2.4.2) Ανάλυση του συστήματος

Ένα Hadoop cluster αποτελείται από έναν master και πολλούς slave κόμβους. Για την δημιουργία, λοιπόν, ενός Hadoop cluster είναι απαραίτητη η εγκατάσταση διάφορων διεργασιών στον master και στους slave κόμβους του cluster. Οι διεργασίες αυτές θα είναι υπεύθυνες για την σωστή επικοινωνία μεταξύ των κόμβων καθώς και για την εκτέλεση των εφαρμογών MapReduce.



Σχήμα 7: Hadoop architecture

Στον master κόμβο εγκαθιστούμε τις παρακάτω διεργασίες:

- **JobTracker:** O 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 του διαθέσιμα τοπικά ή τουλάχιστον αυτά να βρίσκονται σε κάποιον γειτονικό του κόμβο. O JobTracker είναι κρίσιμο σημείο για τη λειτουργία του MapReduce αφού αν σταματήσει να λειτουργεί θα σταματήσει και η εκτέλεση όλων των εργασιών MapReduce. Όπως φαίνεται και στο σχήμα οι λειτουργίες του JobTracker είναι:
    - Οι εφαρμογές πελάτη του υποβάλλουν τις εργασίες MapReduce που θέλουν να εκτελέσουν.
    - O JobTracker επικοινωνεί με τον NameNode για να ενημερωθεί για την τοποθεσία των δεδομένων εισόδου.
    - Λαμβάνοντας υπόψη του την τοποθεσία των δεδομένων και την διαθεσιμότητα των κόμβων του cluster, ο JobTracker, αποφασίζει για το πού θα εκτελεστεί κάθε εργασία.
    - Στη συνέχεια αναθέτει τις εργασίες στους TaskTrackers που τρέχουν στους αντίστοιχους κόμβους.
    - Κατά τη διάρκεια της εκτέλεσης της εργασίας ο JobTracker ενημερώνεται για την

πρόοδό της καθώς και για την κατάσταση των TaskTracker. Αν κάποιος TaskTracker αντιμετωπίσει προβλήματα ο JobTracker είναι υπεύθυνος για την αντιμετώπιση του προβλήματος και την ανάθεση της συγκεκριμένης εργασίας σε άλλο κόμβο.

◦ Τέλος ο JobTracker ενημερώνει τον κώδικα πελάτη για την ολοκλήρωση της εργασίας. Οι εφαρμογές πελάτη μπορούν στη συνέχεια να επικοινωνήσουν με τον JobTracker για να πάρουν επιπλέον πληροφορίες για την εκτέλεση της εργασίας.

• **NameNode:** Ο NameNode είναι η κεντρική εφαρμογή που ελέγχει το HDFS, το κατανεμημένο σύστημα αρχείων του Hadoop. Η διεργασία αυτή διαθέτει το δέντρο του συστήματος αρχείων, το οποίο περιέχει όλα τα αρχεία του συστήματος, και επίσης γνωρίζει που ακριβώς βρίσκεται αποθηκευμένο κάθε αρχείο στο cluster. Ο NameNode δέχεται αιτήσεις από τον κώδικα πελάτη όταν 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 την λειτουργία του HDFS αφού αν σταματήσει να λειτουργεί το filesystem δεν ανταποκρίνεται. Υπάρχει βέβαια μια προαιρετική διεργασία ο SecondaryNameNode που δημιουργεί checkpoint σε τακτά χρονικά διαστήματα και μπορεί να επαναφέρει το σύστημα αρχείων μετά από τη διακοπή λειτουργίας του NameNode. Η διεργασία του NameNode είναι από τις πιο σημαντικές ενός Hadoop cluster και είναι 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:

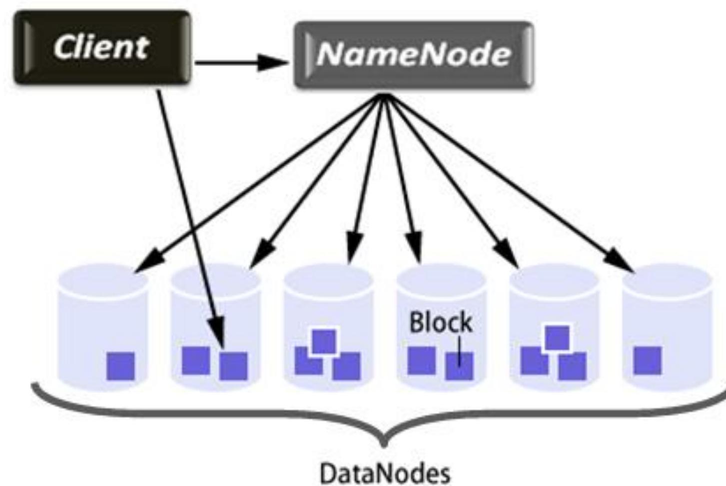
• **TaskTracker:** Αυτή η διεργασία έχει τη δυνατότητα να λαμβάνει και να εκτελεί τις εργασίες που του στέλνει ο JobTracker. Κάθε TaskTracker μπορεί να εκτελεί ταυτόχρονα έναν προκαθορισμένο αριθμό από εργασίες MapReduce. Αυτός ο αριθμός ορίζεται στις παραμέτρους του συστήματος και εξαρτάται από την υπολογιστική δύναμη που έχει κάθε κόμβος στο cluster. Ο TaskTracker δημιουργεί μια ξεχωριστή JVM διεργασία η οποία εκτελεί την πραγματική εργασία MapReduce. Με αυτό τον τρόπο διασφαλίζεται η λειτουργία του TaskTracker ακόμα και σε περίπτωση αποτυχίας κατά την εκτέλεση της πραγματικής εργασίας. Επιπλέον ο TaskTracker αναλαμβάνει την παρακολούθηση των διεργασιών καθώς και την ενημέρωση του JobTracker για τυχόν προβλήματα. Τέλος, ο TaskTracker στέλνει, σε τακτά χρονικά διαστήματα, μηνύματα στον JobTracker (heartbeat messages) τα οποία δηλώνουν ότι βρίσκεται ακόμα σε κατάσταση λειτουργίας. Τα μηνύματα αυτά περιέχουν και άλλες πληροφορίες για την κατάσταση του TaskTracker, όπως ο αριθμός των εργασιών που τρέχουν στον TaskTracker.

• **DataNode:** Αυτή είναι η διεργασία η οποία αναλαμβάνει την αποθήκευση δεδομένων στο κατανεμημένο σύστημα αρχείων HDFS. Ο DataNode επεξεργάζεται τα αιτήματα που του στέλνει ο NameNode και τα οποία αφορούν την εκτέλεση λειτουργιών του συστήματος αρχείων. Για την εκτέλεση αυτών των λειτουργιών ο DataNode επεξεργάζεται τα αρχεία του HDFS τα οποία είναι αποθηκευμένα στο συγκεκριμένο κόμβο. Οι εφαρμογές πελάτη μπορούν να ενημερωθούν από τον NameNode για την τοποθεσία των αρχείων και να επικοινωνήσουν κατευθείαν με τον συγκεκριμένο DataNode. Τέλος είναι δυνατή η επικοινωνία μεταξύ δυο ξεχωριστών DataNode σε περιπτώσεις δημιουργίας αντιγράφων ενός αρχείου, καθώς και μεταξύ ενός TaskTracker και ενός DataNode για ανάγνωση των δεδομένων εισόδου μια εργασίας MapReduce.

2.4.3) 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 8: HDFS architecture

Παρακάτω αναφέρονται μερικά από τα βασικότερα πλεονεκτήματα της σχεδίασης του HDFS.

- Ένα πλεονέκτημα της χρήσης του HDFS είναι το γεγονός ότι έχει σχεδιαστεί για να συνεργάζεται με το Hadoop για την εκτέλεση MapReduce εργασιών. Το κυριότερο στοιχείο αυτής της συνεργασίας είναι ότι ο JobTracker έχει τη δυνατότητα να προγραμματίσει τις εργασίες map/reduce γνωρίζοντας τη θέση των δεδομένων στο cluster. Έτσι μπορεί να κάνει μια βέλτιστη ανάθεση των εργασιών ώστε ο κάθε TaskTracker να βρίσκει τα δεδομένα εισόδου του στο ίδιο μηχάνημα και να μην είναι απαραίτητη η μεταφορά τους μέσω δικτύου. Πρακτικές εφαρμογές έχουν δείξει ότι η χρήση του Hadoop σε συνεργασία με άλλα κατανεμημένα συστήματα αρχείων οδηγεί σε χαμηλότερη απόδοση σε σχέση με τη χρήση του HDFS.
- Το HDFS έχει σχεδιαστεί ώστε να 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. Αυτό σημαίνει ότι δεν υπάρχει ανάγκη για ύπαρξη κάποιου συστήματος RAID αφού η αξιοπιστία εξασφαλίζεται σε μεγάλο βαθμό από το HDFS. Η προκαθορισμένη τιμή των αντιγράφων που κρατούνται είναι 3, δηλαδή για κάθε μπλοκ μνήμης έχουμε 3 αντίγραφά του σε διαφορετικούς κόμβους. Αυτή η τιμή αποτελεί παράμετρο του συστήματος και μπορεί να αλλάξει από τον προγραμματιστή ώστε να επιτυγχάνεται ακόμα μεγαλύτερη αξιοπιστία. Οι DataNodes επικοινωνούν μεταξύ τους και ανταλλάσουν τα αντίγραφα των αρχείων για την επίτευξη ισοκατανομής των δεδομένων στο cluster.
- Με τη χρήση του HDFS έχουμε τη δυνατότητα να αποθηκεύουμε εύκολα μεγάλα σε όγκο αρχεία. Αυτό βασίζεται στο γεγονός ότι στο HDFS τα αρχεία αποθηκεύονται κατά μπλοκ. Βασική παράμετρος του συστήματος είναι το μέγεθος των μπλοκ δεδομένων η προκαθορισμένη τιμή της οποίας είναι 64 MB. Αυτό σημαίνει ότι αν το αρχείο μας είναι μεγαλύτερο από 64 MB θα σπάσει σε μπλοκ τα οποία θα ισοκατανεμηθούν σε όλο το cluster. Αυτό βοηθάει και στην παραλληλοποίηση αφού τα δεδομένα εισόδου έχουν ήδη χωριστεί σε κομμάτια τα οποία μπορούν να επεξεργαστούν από διαφορετικές διεργασίες.
- Βασικό πλεονέκτημα του HDFS είναι το γεγονός ότι είναι κλιμακώσιμο και μπορεί να εγκατασταθεί εύκολα σε μηχανήματα χωρίς ιδιαίτερες απαιτήσεις σε υλικό. Είναι

σχεδιασμένο μάλιστα, έτσι ώστε να μπορούμε να επεκτείνουμε ένα εγκατεστημένο cluster προσθέτοντας ή αφαιρώντας κόμβους, χωρίς να υπάρχει απώλεια δεδομένων. Μάλιστα με την εισαγωγή νέων κόμβων γίνεται αυτόματα και εξισορρόπηση των δεδομένων στο cluster αφού γίνεται μεταφορά αρχείων στους νέους κόμβους.

• Οι DataNodes και Namenodes είναι web servers γεγονός που διευκολύνει τον έλεγχο και την παρουσίαση της κατάστασης τόσο του συστήματος αρχείων όσο και του cluster.

2.4.4) Ανοχή σε σφάλματα

Ένα άλλο πλεονέκτημα του MapReduce είναι το γεγονός ότι έχει μεγάλη ανοχή σε σφάλματα. Ακόμα και σε cluster τα οποία περιέχουν κόμβους εξαιρετικά επιρρεπής σε σφάλματα, το Hadoop έχει την ικανότητα να επιτυγχάνει την εκτέλεση απαιτητικών εργασιών MapReduce. Ο βασικός τρόπος με τον οποίο το Hadoop καταφέρνει να ανταπεξέλθει στα σφάλματα είναι η επανεκτέλεση των εργασιών οι οποίες αποτυγχάνουν. Οι TaskTrackers επικοινωνούν σε τακτά χρονικά διαστήματα με τον JobTracker ενημερώνοντάς τον για την κατάστασή τους. Αν κάποιος TaskTracker δεν έχει επικοινωνήσει με τον JobTracker για ορισμένο χρονικό διάστημα, ο JobTracker θεωρεί ότι ο TaskTracker σταμάτησε να λειτουργεί. Ο JobTracker γνωρίζει τις διεργασίες που εκτελούσε ο TaskTracker και τις αναθέτει σε κάποιον άλλον κόμβο.

Υπάρχουν δυο περιπτώσεις στις οποίες μπορεί να παρουσιαστεί μια διακοπή λειτουργίας ενός TaskTracker. Αν η διακοπή συμβεί κατά τη διάρκεια της εκτέλεσης μιας reduce εργασίας, μόνο η εργασία που ήταν υπό εκτέλεση πρέπει να ξαναεκτελεστεί. Εργασίες reduce που έχουν ήδη ολοκληρωθεί από τον συγκεκριμένο TaskTracker δεν πρέπει να ξαναεκτελεστούν. Οι εργασίες reduce γράφουν τα αποτελέσματά τους στο HDFS και άρα δεν υπάρχει πρόβλημα απώλειας τον δεδομένων εξόδου.

Στην περίπτωση που η διακοπή λειτουργίας του TaskTracker συμβεί κατά τη διάρκεια εκτέλεσης μιας map εργασίας τότε θα πρέπει να ξαναεκτελεστούν όλες οι εργασίες map που έχουν εκτελεστεί ή εκτελούνταν από τον TaskTracker. Οι εργασίες map γράφουν τα αποτελέσματά τους στον τοπικό δίσκο του μηχανήματος στο οποίο εκτελούνταν. Αυτό σημαίνει ότι σε περίπτωση που δεν ανταποκρίνεται ένας TaskTracker δεν μπορούμε να έχουμε πρόσβαση στον συγκεκριμένο κόμβο και άρα ούτε στα αποτελέσματα τον εργασιών map που έχουν εκτελεστεί σε αυτόν.

Η μεγάλη αυτή ανοχή του Hadoop σε σφάλματα προέρχεται βέβαια και από τις ιδιότητες του MapReduce. Όπως έχουμε αναφέρει τόσο οι mappers όσο και οι reducers είναι διεργασίες που δεν επικοινωνούν καθόλου μεταξύ τους. Αυτό σημαίνει ότι αν μια εργασία αποτύχει μπορεί να ξαναεκτελεστεί χωρίς να επηρεάσει τις υπόλοιπες. Κάτι τέτοιο θα ήταν πολύ δύσκολο να υλοποιηθεί σε ένα πλαίσιο παραλληλοποίησης στο οποίο οι διεργασίες επικοινωνούν και εξαρτώνται από την κατάσταση άλλων διεργασιών.

Μια ενδιαφέρουσα περίπτωση σφάλματος είναι η διακοπή λειτουργίας του master κόμβου. Σε αυτή την περίπτωση το σύστημα δεν μπορεί να ολοκληρώσει επιτυχώς την εκτέλεση του. Όμως δεν είναι δύσκολο να αποθηκεύουμε σε τακτά χρονικά διαστήματα την κατάσταση και τις δομές δεδομένων του master κόμβου ώστε να μπορέσουμε να επανεκκινήσουμε το σύστημά μας ακόμα και από μια τέτοια βλάβη. Βέβαια υπάρχει μόνο ένας master κόμβος στο cluster και η πιθανότητα να σταματήσει να λειτουργεί είναι μικρή, ειδικά αν έχουμε επιλέξει να είναι κάποιος από τους πιο αξιόπιστους υπολογιστές του cluster μας.

2.4.5) Speculative execution

Όπως έχουμε αναφέρει, με τη χρήση του Hadoop, μπορούμε να στήσουμε πολύ μεγάλα cluster με κόμβους που μπορεί να έχουν εντελώς διαφορετική υπολογιστική ισχύ. Συχνά σε τέτοια cluster υπάρχουν μερικοί κόμβοι οι οποίοι είναι πιο αργοί από τους υπόλοιπους με αποτέλεσμα να καθυστερούν τον τερματισμό της εργασίας μας. Σε τέτοιες περιπτώσεις χρησιμοποιούμε την τεχνική του speculative execution. Τη στιγμή λοιπόν που βλέπουμε ότι αρκετές από τις διεργασίες έχουν τερματιστεί και υπάρχουν ανενεργοί κόμβοι στο cluster, αναθέτουμε αντίγραφα των εργασιών που αργούν να ολοκληρωθούν σε άλλους κόμβους. Όποιο από τα αντίγραφα της εργασίας τελειώσει πρώτο ενημερώνει τον JobTracker ο οποίος με τη σειρά του ενημερώνει τα υπόλοιπα αντίγραφα να σταματήσουν την εκτέλεσή τους. Η τεχνική του speculative execution μπορεί να απενεργοποιηθεί από τον προγραμματιστή αν το cluster δεν διαθέτει αργούς κόμβους για λόγους απόδοσης.

2.4.6) Βασικά Interfaces του Hadoop

Παρακάτω αναφέρονται τα βασικότερα interfaces του Hadoop τα οποία ορίζουν τον τρόπο εισόδου, εξόδου και επεξεργασίας τον δεδομένων. Ο προγραμματιστής μπορεί να υλοποιήσει όπως επιθυμεί όλα αυτά τα interfaces ή να χρησιμοποιήσει κάποια από τις διαθέσιμες υλοποιήσεις που υπάρχουν στο πακέτο του Hadoop.

Input files: Ορίζει τον τύπο αρχείων ο οποίος χρησιμοποιείται για τα δεδομένα εισόδου της εργασίας MapReduce. Τα αρχεία αυτά είναι συνήθως αποθηκευμένα στο HDFS αν και μπορεί να είναι σε οποιοδήποτε άλλο σύστημα αρχείων χρησιμοποιείται στο cluster. Το input file ορίζει τη διαμόρφωση αυτών των αρχείων τα οποία μπορεί να είναι δυαδικά, αρχεία κειμένου, ή κάτι εντελώς διαφορετικό.
InputFormat: Παρέχει τις παρακάτω λειτουργίες:

- Επιλέγει τα αρχεία ή τα αντικείμενα που θα χρησιμοποιηθούν σαν είσοδος
- Χωρίζει τα αρχεία αυτά σε InputSplits καθένα από τα οποία θα αποτελέσει την είσοδο ενός mapper
- Ορίζει έναν RecordReader για την ανάγνωση των InputSplits.

Υπάρχουν διάφορες διαθέσιμες υλοποιήσεις του interface αυτού στο Hadoop. Η βασική υλοποίησή της ονομάζεται TextInputFormat και χρησιμοποιείται για την εισαγωγή δεδομένων από αρχεία κειμένου. Χρησιμοποιώντας αυτό το InputFormat έχουμε ζευγάρια key/value τα οποία έχουν ως τιμή μια γραμμή του κειμένου και ως κλειδί τον σειριακό αριθμό του πρώτου byte της γραμμής. Ένα άλλο InputFormat που συμπεριλαμβάνεται στο Hadoop είναι το KeyValueInputFormat. Με τη χρήση αυτής της υλοποίησης μπορούμε να διαβάσουμε δεδομένα από αρχεία που είναι ειδικά διαμορφωμένα και περιέχουν σε κάθε γραμμή ένα ζεύγος key/value. Σαν κλειδί του ζευγαριού λαμβάνεται ότι περιέχεται στη γραμμή μέχρι τον πρώτο tab χαρακτήρα και σαν τιμή όλη υπόλοιπη γραμμή. Αυτό το InputFormat χρησιμοποιείται για την επεξεργασία δεδομένων που είναι αποτέλεσμα κάποιας άλλης εργασίας MapReduce. Ο προγραμματιστής μπορεί να επιλέξει μια από τις υλοποιήσεις που περιέχονται στο Hadoop ή να φτιάξει μια καινούργια που θα ανταποκρίνεται στις ανάγκες του.
InputSplits: Το interface αυτό περιγράφει τη βασική μονάδα στην οποία χωρίζονται τα δεδομένα εισόδου για την επεξεργασία τους. Το InputFormat που χρησιμοποιούμε κάθε φορά είναι υπεύθυνο για το χωρισμό των δεδομένων εισόδου σε κομμάτια τα οποία θα επεξεργαστούν από διαφορετικές διεργασίες. Τα InputSplit δεν περιέχουν τα πραγματικά δεδομένα εισόδου αλλά μια περιγραφή τους με την οποία οι mappers θα μπορέσουν να τα ανακτήσουν από το HDFS. Μια τέτοια περιγραφή μπορεί να περιέχει για παράδειγμα την αρχική και τελική γραμμή ενός αρχείου κειμένου, καθώς και την τοποθεσία αυτού του

αρχείου στο cluster. Αυτές οι περιγραφές πρέπει να αποθηκεύονται σε αντικείμενα τα οποία είναι σειριοποιήσιμα, ώστε να μπορούν να μεταφερθούν μέσω δικτύου στους mappers.

RecordReader: Το κάθε InputSplit δημιουργείται και αποστέλλεται στον αντίστοιχο κόμβο για επεξεργασία. O mapper όμως πρέπει να γνωρίζει τον τρόπο με τον οποίο θα μετατρέψει τα δεδομένα στα οποία αναφέρεται το InputSplit, σε ζευγάρια key/value τα οποία θα χρησιμοποιηθούν ώς είσοδος. Αυτό γίνεται με χρήση μιας υλοποίησης του interface RecordReader. O RecordReader που χρησιμοποιείται κάθε φορά ορίζεται από το InputFormat. O RecordReader καλείται συνεχώς και επιστρέφει κάθε φορά ένα ζευγάρι key/value μέχρι να καταναλώσει όλα τα δεδομένα εισόδου που ορίζονται από το InputSplit. Κάθε κλήση οδηγεί στην εκτέλεση της συνάρτησης map πάνω στο συγκεκριμένο ζευγάρι key/value.

Mapper: Αυτό είναι το βασικό interface το οποίο καλείται ο προγραμματιστής να υλοποιήσει ώστε να ορίσει την λειτουργία της συνάρτησης map. Όπως έχουμε αναφέρει η συνάρτηση map παίρνει ως είσοδο ένα ζευγάρι key/value και παράγει μια λίστα από άλλα ζευγάρια key/value. Το αντικείμενο Context που διαθέτει ο Mapper χρησιμοποιείται κάθε φορά για την αποθήκευση των ζευγαριών εξόδου καθώς και για άλλες λειτουργίες όπως είναι η ενημέρωση του Mapper για τις παραμέτρους του συστήματος και η ενημέρωση του συστήματος για την εξέλιξη της εργασίας του Mapper.

Partition & Shuffle: Τα ενδιάμεσα ζευγάρια key/value που παράγονται από τους mappers πρέπει να μοιραστούν σε R περιοχές και να μετακινηθούν στο κόμβο όπου θα εκτελεστεί ο αντίστοιχος reducer. Το λειτουργία του διαμοιρασμού των ζευγαριών σε R περιοχές αναλαμβάνει ο Partitioner, ενώ η φάση της μετακίνησης των δεδομένων ονομάζεται shuffling. Κάθε φορά που ο mapper παράγει ένα ζευγάρι key/value καλείται ο Partitioner ο οποίος επιστρέφει τον αριθμό που αντιστοιχεί στην περιοχή που τοποθετείται το ζευγάρι. Ο προεπιλεγμένος Partitioner που χρησιμοποιείται στο Hadoop χωρίζει τα ζευγάρια με βάση τη συνάρτηση hash(key) mod R. Η συνάρτηση αυτή εξασφαλίζει ισοκατανομή των ζευγαριών στις περιοχές, ώστε να μην υπερφορτώνονται συγκεκριμένοι reducers. Ο προγραμματιστής μπορεί να φτιάξει βέβαια τον δικό του Partitioner ώστε να ελέγξει τον τρόπο διαμοιρασμού των δεδομένων.

Sort: Το πλαίσιο του Hadoop αναλαμβάνει να ταξινομήσει τα δεδομένα κάθε περιοχής πριν τα παρουσιάσει στον Reducer. Μπορούμε να χρησιμοποιήσουμε και εξωτερική ταξινόμηση όταν τα δεδομένα της περιοχής δεν χωράνε στη μνήμη.

Reducer: Όπως και με τον mapper ο προγραμματιστής οφείλει να παρέχει μια υλοποίηση της συνάρτησης reduce. Η συνάρτηση reduce παίρνει ως είσοδο ένα κλειδί και μια λίστα με όλες τις τιμές που αντιστοιχούν σε αυτό και παράγει μια λίστα από ζευγάρια key/value. Ο reducer χρησιμοποιεί και αυτός το αντικείμενο Context για την αποθήκευση των ζευγαριών εξόδου καθώς και για την ενημέρωση του συστήματος για την εξέλιξή του.

OutputFormat: Τα ζευγάρια key/value που παράγονται από τους reducers πρέπει να γραφτούν σε αρχεία και να αποθηκευτούν στο HDFS. Αυτό γίνεται με τη χρήση κάποιου OutputFormat το λειτουργεί παρόμοια με το InputFormat που έχουμε περιγράψει. Οι περισσότερες υλοποιήσεις του OutputFormat είναι υποκλάσεις της FileOutputFormat συμφωνα με την οποία κάθε reducer δημιουργεί ένα ξεχωριστό αρχείο σε έναν κοινό φάκελο στο HDFS. Αυτά τα αρχεία ονομάζονται part-nnnnn, όπου nnnn είναι ο σειριακός αριθμός της περιοχής (partition) που επεξεργάζεται κάθε reducer. Η προεπιλεγμένη υλοποίηση του OutputFormat που χρησιμοποιείται στο Hadoop είναι η TextOutputFormat, η οποία γράφει τα ζευγάρια εξόδου των reducers σε ξεχωριστές γραμμές αρχείου κειμένου με μορφή κατάλληλη ώστε να μπορούν να διαβαστούν με χρήση του KeyValueInputFormat. Τα αρχεία εξόδου που δημιουργούντια με χρήση του TextOutputFormat είναι επίσης εύκολα αναγνώσιμα και μπορούν να χρησιμεύσουν για έλεγχο της σωστής εκτέλεσης της εφαρμογής από τον προγραμματιστή. Άλλες υλοποιήσεις αυτού του interface είναι οι:

- SequenceFileOutputFormat η οποία γράφει δυαδικά αρχεία που μπορούν να επεξεργαστούν με τη χρήση του SequenceFileInputFormat και βοηθάει στην μείωση του όγκου των δεδομένων εξόδου.
- NullOutputFormat η οποία αγνοεί τα δεδομένα εξόδου που παράγονται από τους reducers και δεν δημιουργεί αρχεία εξόδου. Αυτό χρησιμοποιείται και σε περιπτώσεις που ο προγραμματιστής δεν θέλει να χρησιμοποιήσει το πλαίσιο του Hadoop για να δημιουργήσει την έξοδο της εφαρμογής του και αποθηκεύει τα δεδομένα εξόδου του με κάποια διαδικασία μέσα στην συνάρτηση reduce.

RecordWriter: Ο RecordWriter χρησιμοποιείται για να γράψει τα ζευγάρια που παράγονται από τους reducers στα αρχεία εξόδου.

Combiner: Αυτό είναι ένα προαιρετικό στάδιο του MapReduce και βρίσκεται μεταξύ του map και του reduce. Ο Combiner τρέχει μετά από το mapper και στον ίδιο κόμβο. Πριν τα δεδομένα εξόδου του mapper σταλούν στον reducer μπορούν να επεξεργαστούν από τον Combiner. Ο Combiner είναι μια "μικρή-reduce" διεργασία η οποία εφαρμόζεται σε δεδομένα που προέρχονται μόνο από ένα κόμβο. Χρησιμοποιώντας τον Combiner μειώνουμε τα δεδομένα που μεταφέρονται στο δίκτυο, καθώς και τον φόρτο εργασίας των reducers. Ο Combiner είναι και αυτός μια υλοποίηση του interface του Reducer. Το πιο σημαντικό στοιχείο είναι ότι σε πολλές περιπτώσεις μπορούμε να χρησιμοποιήσουμε ως Combiner τον reducer, και χωρίς επιπλέον κόπο να βελτιώσουμε δραστικά την απόδοση της εφαρμογής μας.

2.5) HBase

2.5.1) Εισαγωγή

Η HBase είναι ένα πακέτο ανοικτού κώδικα που υλοποιεί μια μη σχεσιακή, κατανεμημένη βάση δεδομένων που έχει σχεδιαστεί σύμφωνα με το άρθρο της Google BigTable[4]. Το πακέτο αυτό έχει δημιουργηθεί και αυτό από την εταιρεία Apache ώστε να λειτουργεί σε συνεργασία 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. Τα τελευταία χρόνια η απόδοση της HBase έχει αυξηθεί, σε βαθμό στον οποίο χρησιμοποιείται σε εμπορικές εφαρμογές υψηλών απαιτήσεων όπως στην πλατφόρμα των μηνυμάτων του Facebook.

2.5.2) Μοντέλο δεδομένων

Αρχικά πρέπει να ξεκαθαρίσουμε το γεγονός ότι η HBase δεν έχει καμία σχέση με τις γνωστές μας σχεσιακές βάσεις δεδομένων καθώς δεν χρησιμοποιεί την διαπροσωπία της SQL, ούτε το entity-relationship μοντέλο αποθήκευσης δεδομένων. Όπως αναφέρεται στο άρθρο του BigTable σύμφωνα με το οποίο σχεδιάστηκε, η HBase είναι:

"A sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes."

Η παραπάνω φράση είναι εξαιρετικά περιεκτική όμως περιγράφει όλα τα χαρακτηριστικά του μοντέλου δεδομένων που χρησιμοποιείται από την HBase. Για να καταλάβουμε πλήρως το νόημα της πρέπει να εξετάσουμε αναλυτικά κάθε λέξη που

περιέχεται σε αυτή.

map: Η HBase είναι ουσιαστικά ένα μεγάλο map. Άλλοι πιο γνώριμοι όροι που εκφράζουν την δομή του map σε διάφορες γλώσσες προγραμματισμού είναι: hash table στη Java, dictionary στην Python, hash στη Ruby και associative array στην PHP. Το map, λοιπόν, είναι μια αφηρημένη δομή δεδομένων που περιέχει μια συλλογή από κλειδιά και μια συλλογή από τιμές, όπου κάθε κλειδί είναι συνδεδεμένο με μια μόνο τιμή. Μια τέτοια δομή παρουσιάζεται παρακάτω:

```
{
  "a" : "hello",
  "b" : "world",
  "1" : "!!!"
}
```

persistent: Με τη χρήση αυτής της λέξης δηλώνουμε ότι τα δεδομένα που αποθηκεύονται στην HBase αποθηκεύονται μόνιμα δηλαδή δεν χάνονται με τον τερματισμό λειτουργίας του προγράμματος. Ακόμα και αν το cluster μας πάθει κάποια βλάβη και σταματήσει να λειτουργεί τα δεδομένα που έχουμε αποθήκευση στην HBase δεν χάνονται. Τα δεδομένα αποθηκεύονται στους σκληρούς δίσκους των κόμβων με τρόπο τέτοιο ώστε να μπορούμε να τα ανακτήσουμε με την επανεκκίνηση του cluster.

distributed: Η Hbase αποτελεί ένα κατανεμημένο σύστημα αποθήκευσης αφού βασίζεται στο σύστημα αρχείων HDFS. Το HDFS αναλαμβάνει να κατανείμει τα αρχεία της HBase σε όλους τους κόμβους του cluster. Με αυτό τον τρόπο εξασφαλίζονται όλες οι ιδιότητες που παρέχει το HDFS τόσο για ανοχή σε σφάλματα όσο και για διατήρηση πολλαπλών αντιγράφων και χωρισμό των μεγάλων αρχείων σε μπλοκ.

sorted: Τα ζευγάρια key/value που αποθηκεύονται στην HBase αποθηκεύονται με αλφαβητική σειρά με βάση το κλειδί. Με την ταξινόμηση των ζευγαριών εξασφαλίζουμε την ιδιότητα της τοπικότητας των δεδομένων. Αυτό σημαίνει ότι ένα ζευγάρι με κλειδί “aa” θα βρίσκεται ακριβώς δίπλα σε ένα ζευγάρι με κλειδί “ab” και πολύ μακριά από ένα με κλειδί “z”. Με αυτό τον τρόπο βελτιώνεται η απόδοση ειδικά σε περιπτώσεις σειριακής ανάγνωσης των πινάκων. Ο προγραμματιστής μπορεί να επιλέξει τα κλειδιά που θα χρησιμοποιήσει με τέτοιο τρόπο ώστε οι προσβάσεις του στη μνήμη να παρουσιάζουν την μεγαλύτερη δυνατή τοπικότητα. Για παράδειγμα αν θέλαμε να δημιουργήσουμε έναν πίνακα που τα κλειδιά του είναι domain names θα μπορούσαμε να πετύχουμε καλύτερη τοπικότητα αν αποθηκεύαμε τα κλειδιά αντεστραμμένα (π.χ. “com.google.www”). Αυτό σημαίνει ότι ζευγάρια που ανήκουν στο ίδιο domain (π.χ. “com.google.maps”) θα βρίσκονταν πολύ κοντά μεταξύ τους. Κάτι τέτοιο δεν θα μπορούσε να επιτευχθεί αν τα κλειδιά μας ήταν της μορφής “www.google.com”. Αφήνεται, λοιπόν η ευθύνη της επιλογής των κατάλληλων κλειδιών στον προγραμματιστή. Τέλος αξίζει να τονιστεί το γεγονός η ταξινόμηση γίνεται μόνο με βάση το κλειδί και δεν υπάρχει καμία ταξινόμηση στις τιμές που περιέχουν τα ζευγάρια.

multidimensional: Στην HBase χρησιμοποιούνται όροι όπως “table” και “columns” οι οποίοι μας παραπέμπουν στις κλασικές σχεσιακές βάσεις δεδομένων. Στην πραγματικότητα όμως η HBase είναι εντελώς διαφορετική από αυτές. Όπως αναφέρεται και στον ορισμό η HBase είναι ένα “multidimensional (πολυδιάστατο) map”, δηλαδή ένα map που μπορεί να παίρνει τιμές που είναι και αυτές map.

```
{
 //...
  "aaa" : {
   "A" : {
     "foo" : "y",
     "bar" : "d"
   },
   "B" : {
     "" : "w"
   }
  },
  "aab" : {
   "A" : {
     "foo" : "world",
     "bar" : "domination"
   },
   "B" : {
     "" : "ocean"
   }
  },
 //...
}
```

Στο παραπάνω παράδειγμα φαίνεται ένα τέτοιο πολυδιάστατο map, στο οποίο οι τιμές των ζευγαριών είναι και αυτές map. Μια τέτοια δομή είναι και η δομή ενός πίνακα HBase. Τα κλειδιά του υψηλότερου επιπέδου (“aaa”, “aab”) αποτελούν τις γραμμές του πίνακα ενώ τα κλειδιά δεύτερου επιπέδου (“A”, “B”) ονομάζονται column families.

Τα column families ενός πίνακα HBase ορίζονται κατά την δημιουργία ενός πίνακα και είναι εξαιρετικά δύσκολο να αλλάξουν στη συνέχεια. Κάθε column family μπορεί να περιέχει οποιοδήποτε αριθμό από qualifiers. Όπως βλέπουμε και στο παραπάνω παράδειγμα το column family “A” περιέχει δύο qualifiers τα “foo” και “bar”, ενώ το “B” περιέχει μόνο ένα που έχει ως κλειδί ένα κενό string. Τα qualifiers δεν χρειάζεται να ορισθούν, μπορούμε δηλαδή να χρησιμοποιήσουμε οποιοδήποτε string επιθυμούμε ως qualifier χωρίς κάποιο περιορισμό. Το συνολικό όνομα της στήλης έχει λοιπόν τη μορφή family:qualifier, δηλαδή στο παραπάνω παράδειγμα έχουμε τις στήλες “A:foo”, “A:bar” και “B:”.

Η βασικότερη μονάδα που αποθηκεύεται σε έναν πίνακα HBase είναι το cell το οποίο προσδιορίζεται από την τριπλέτα (“row”, “column_family:qualifier”). Κάθε τέτοιο cell μπορεί να περιέχει πολλαπλές τιμές οι οποίες διαχωρίζονται με χρήση ενός timestamp. Το timestamp είναι ένας ακέραιος 64-bit και μπορεί να περιέχει την πραγματική ώρα εγγραφής του cell ή μια άλλη τιμή που ορίζεται από τον προγραμματιστή. Αυτή η τιμή χρησιμοποιείται από εφαρμογές που θέλουν να αποφεύγουν συγκρούσεις δεδομένων. Η HBase αναλαμβάνει να ταξινομεί τις τιμές αυτές με βάση το πεδίο timestamp ώστε κάθε φορά να παρέχεται στον χρήστη η τελευταία εκδοχή του πεδίου. Ο προγραμματιστής μπορεί να προσδιορίσει τον αριθμό των αντιγράφων που επιθυμεί να κρατούνται κάθε στιγμή και το σύστημα της HBase αναλαμβάνει να απελευθερώνει αυτόματα τις εγγραφές που δεν χρειάζονται με έναν gardage-collector. Τέλος ο προγραμματιστής μπορεί να ζητήσει τα δεδομένα ενός cell παρέχοντας το πεδίο timestamp, με αυτό τον τρόπο μπορεί να ανακτήσει παλαιότερες τιμές του πεδίου.

sparse: Αυτός ο όρος δηλώνει ότι οι πίνακες που δημιουργούνται από την HBase είναι εξαιρετικά αραιοί. Η ιδιότητα αυτή προκύπτει από το γεγονός ότι μια γραμμή μπορεί να περιέχει εντελώς διαφορετικές στήλες από μια άλλη γραμμή του ίδιου πίνακα. Όπως αναφέραμε ο χρήστης μπορεί να επιλέξει οποιοδήποτε qualifier για την κάθε στήλη χωρίς να το ορίζει εξαρχης και έτσι προκύπτουν εντελώς διαφορετικές στήλες για κάθε γραμμή.

2.5.3) Επιλογές σχεδίασης πινάκων HBase

Όλες οι στήλες που ανήκουν στο ίδιο column family έχουν υποχρεωτικά και τα ίδια χαρακτηριστικά. Ο κύριος λόγος, λοιπόν, για τον οποίο δημιουργούμε ξεχωριστά column families είναι ο διαχωρισμός των επιλογών μεταξύ των στηλών του πίνακα. Τέτοιες επιλογές μπορεί να είναι:

Συμπίεση

Από προεπιλογή, τα δεδομένα που περιέχονται στα cell της HBase δεν συμπιέζονται. Όμως μπορούμε να επιλέξουμε μεταξύ δυο τύπων συμπίεσης.
Record compression: Αυτό το είδος εφαρμόζει τον αλγόριθμο συμπίεσης σε κάθε γραμμή ξεχωριστά.
Block compression: Ο αλγόριθμος συμπίεσης εφαρμόζεται σε μπλοκ δεδομένων του πίνακα τα οποία μπορεί να περιέχουν περισσότερες από μια γραμμές του.
Με τη χρήση του block compression μπορούμε να πετύχουμε καλύτερη συμπίεση των δεδομένων από πλευρά όγκου, ενώ με χρήση του record compression μπορούμε, θεωρητικά, να πετύχουμε καλύτερο χρόνο προσπέλασης μιας συγκεκριμένης γραμμής

Bloom Filters

Αν ένα column family υποστηρίζει bloom filters, σημαίνει ότι κρατάμε ένα δευτερεύον index για τα ονόματα που περιέχονται μέσα σε αυτό το column family. Αυτά τα index είναι ιδιαίτερα χρήσιμα σε περιπτώσεις που υπάρχουν πολλές διαφορετικές στήλες στον πίνακά μας και κάθε στήλη περιέχει λίγα δεδομένα. Δημιουργώντας αυτό το δευτερεύον index επιτυγχάνουμε μείωση στο χρόνο αναζήτησης μιας συγκεκριμένης στήλης.
Όπως όλα τα index έτσι και αυτό χρειάζεται επιπλέον χώρο αποθήκευσης και απαιτεί επιπλέον χρόνο για την δημιουργία και ανανέωσή του. Σε πίνακες που υποστηρίζουν bloom filters οι ενέργειες εισαγωγής και αναζήτησης δεδομένων είναι αρκετά γρήγορες, αντίθετα η διαγραφή δεδομένων καθυστερεί αφού σε ορισμένες περιπτώσεις απαιτείται η δημιουργία του index από την αρχή.

MAX_LENGTH

Με αυτή την επιλογή ορίζουμε το μεγαλύτερο δυνατό μέγεθος δεδομένων που μπορούν να αποθηκευτούν σε ένα cell. Γενικά είναι καλή επιλογή να αποθηκεύουμε όσο το δυνατόν μικρότερες τιμές σε κάθε cell. Για πρακτικούς λόγους δεν θα πρέπει να αποθηκεύουμε δεδομένα τα οποία είναι μεγαλύτερα από μερικά MB. Τέλος, αν θέλουμε να έχουμε στήλες που περιέχουν μεγάλα δεδομένα, καλό είναι να τις τοποθετούμε σε ξεχωριστά, ειδικά διαμορφωμένα column families.

MAX_VERSIONS

Με αυτή την επιλογή καθορίζουμε τον αριθμό των διαφορετικών αντιγράφων που κρατώνται για κάθε cell. Όπως καταλαβαίνουμε, αυτή είναι μια πολύ σημαντική επιλογή αφού επηρεάζει άμεσα το συνολικό χώρο που καταλαμβάνει ο πίνακάς μας. Η προεπιλεγμένη τιμή είναι 3, αλλά αν η εφαρμογή μας δεν χρειάζεται τέτοια αντίγραφα θα πρέπει να τη μειώσουμε για να πετύχουμε εξοικονόμηση χώρου αποθήκευσης.

IN_MEMORY

Με την επιλογή IN_MEMORY μπορούμε να επιβάλουμε στην HBase να κρατάει τις τιμές των cell στη μνήμη σε μεγαλύτερο βαθμό από ότι θα έκανε σε κανονική περίπτωση. Με αυτό τον τρόπο μπορούμε να βελτιώσουμε τον χρόνο διαδοχικών προσβάσεων στα ίδια δεδομένα. Φυσικά, για να επιτευχθεί αυτό η εφαρμογή μας θα απαιτεί περισσότερη μνήμη RAM από το σύστημα. Ο προγραμματιστής θα πρέπει λοιπόν με χρήση αυτής της επιλογής, να βρει τη χρυσή τομή ανάμεσα στην απόδοση και την κατανάλωση μνήμης RAM.

3) Συσχετιζόμενες εργασίες

3.1) Αποθήκευση RDF-triple

Σε αυτό το σημείο θα παρουσιάσουμε εργασίες που έχουν δημοσιευθεί σε επιστημονικά περιοδικά και έχουν σχετιζόμενη, με τη δική μας, θεματολογία. Τέτοιες εργασίες έχουν χρησιμοποιηθεί ως πηγή έμπνευσης για την δική μας υλοποίηση και κάποιες θα χρησιμοποιηθούν για σύγκριση.

Όσον αφορά την εκτέλεση SPARQL ερωτημάτων σε RDF βάσεις υπάρχουν πολλές προσπάθειες για δημιουργία ενός συστήματος, με απόδοση ικανή να υποστηρίζει τις εφαρμογές του Semantic Web. Αυτές οι προσπάθειες στοχεύουν κυρίως στην μείωση του χρόνου εκτέλεσης των ερωτημάτων, καθώς και στην δημιουργία κλιμακώσιμων συστημάτων, τα οποία θα μπορούν να ανταπεξέλθουν στο μεγάλο όγκο δεδομένων που περιέχονται στο διαδίκτυο.

Αρχικά τα συστήματα που δημιουργήθηκαν χρησιμοποιούσαν κλασικές σχεσιακές βάσεις δεδομένων για την αποθήκευση των RDF triple. O πιο απλός τρόπος αποθήκευσης ήταν η αποθήκευση όλων των triples σε έναν τεράστιο πίνακα με στήλες subject, predicate, object. Φυσικά αυτή η επιλογή δεν ήταν καθόλου αποδοτική αφού δεν διευκόλυνε την εύκολη ανάκτηση των trples.

Μια πιο εξελιγμένη ιδέα παρουσιάστηκε στο άρθρο [17], και είναι η αποθήκευση ενός πίνακα για κάθε διαφορετικό predicate. O αριθμός των διαφορετικών predicate, σε μια RDF βάση, είναι συνήθως περιορισμένος και έτσι διευκολύνεται η διατήρηση ενός περιορισμένου αριθμού τέτοιων πινάκων. Μια τέτοια μορφή αποθήκευσης διευκολύνει ιδιαίτερα την απάντηση ερωτημάτων που έχουν προκαθορισμένο το predicate, όμως υστερεί στις ερωτήσεις που το predicate είναι μεταβλητή. Στην αρχή, υπήρξαν πολλά άρθρα που έδειχναν ότι η απόδοση της συγκεκριμένης μορφής αποθήκευσης μειώνεται δραστικά όταν αντιμετωπίζουμε πολλά δεδομένα.

Παρ' όλες τις αντιρρήσεις που υπήρξαν για την παραπάνω μορφή αποθήκευσης, πρόσφατα άρθρα παρουσιάζουν εφαρμογές οι οποίες δεν δημιουργούν πίνακες μόνο για τα predicate, αλλά για όλους τους δυνατούς συνδυασμούς. Μια τέτοια εκδοχή παρουσιάζεται στο άρθρο του Hexastore [18], στο οποίο χρησιμοποιούνται 6 διαφορετικά indexes, που καλύπτουν όλες τους δυνατούς συνδυασμούς για τη μορφή subject-predicate-object. Αυτά τα index είναι τα SPO, PSO, POS, OPS, SOP, OSP. Για παράδειγμα το SPO περιέχει για κάθε διαφορετικό subject μια λίστα από τα predicates και κάθε predicate έχει ένα δείκτη σε ένα πίνακα που περιέχει όλα τα object. Δημιουργώντας όλα αυτά τα index έχουμε τη δυνατότητα να ανακτήσουμε από τη βάση μας οποιαδήποτε triple pattern με το ελάχιστο δυνατό κόστος από άποψη χρόνου. Βέβαια ο τρόπος αυτός αποθήκευσης είναι ιδιαίτερα απαιτητικός σε μνήμη αφού χρειάζεται χώρο ο οποίος είναι τουλάχιστον 5 φορές μεγαλύτερος από την αποθήκευση ενός μόνο index. Η χρήση τέτοιων index για αποθήκευση των δεδομένων γίνεται όλο και πιο δημοφιλής αν και υπάρχει μια διάσταση των απόψεων σχετικά με τον βέλτιστο αριθμό indexes που πρέπει να χρησιμοποιούμε. Τέτοιες υλοποιήσεις είναι το BitMat [19] το RDF-3X [15] το Virtuoso [14] και άλλα.

Το πιο ολοκληρωμένο πακέτο λογισμικού που έχει δημιουργηθεί για την εκτέλεση SPARQL ερωτημάτων είναι το πακέτο ελεύθερου κώδικα Jena [11]. Δημιουργήθηκε από την εταιρία HP και στη συνέχεια έγινε πακέτο ελεύθερου κώδικα, το οποίο είναι επεκτάσιμο και διαθέτει πολύ καλή τεκμηρίωση για την διευκόλυνση των προγραμματιστών. Η Jena παρέχει ένα προγραμματιστικό περιβάλλον το οποίο υλοποιεί τις τεχνολογίες RDF, RDFS, OWL και SPARQL. Παρέχει επίσης το πακέτο ARQ που έχει δημιουργηθεί για την αποθήκευση και επερώτηση RDF δεδομένων. Το ARQ, όμως, έχει σχεδιαστεί για δεδομένα μικρού μεγέθους καθώς διατηρεί τις απαραίτητες δομές δεδομένων στην μνήμη και όχι στο

δίσκο. Αυτό σημαίνει ότι δεν μπορεί να επεξεργαστεί RDF βάσεις που είναι αρκετά μεγάλες ώστε να μη χωράνε στη μνήμη. Βασικό πλεονέκτημα της Jena είναι το γεγονός ότι μπορεί ο προγραμματιστής να δημιουργήσει το δικό του υποσύστημα (π.χ. για αποθήκευση των RDF δεδομένων) και να το συνδέσει με το σύστημα της Jena, δημιουργώντας έτσι ένα ολοκληρωμένο σύστημα που παρέχει τις νέες δυνατότητες.

Μια από τις πιο γρήγορες και κλιμακώσιμες υλοποιήσεις είναι το BigOWLIM [13]. Βασικό μειονέκτημά του όμως, αποτελεί το γεγονός ότι και αυτό χρησιμοποιεί δομές που αποθηκεύονται στη μνήμη και όχι στο δίσκο, με συνέπεια να απαιτεί ισχυρά μηχανήματα για την εκτέλεσή του. Αυτό αποτελεί βασικό εμπόδιο στην κλιμάκωση του συστήματος, αφού το BigOWLIM αποτελεί εφαρμογή που τρέχει αποκλειστικά σε έναν υπολογιστή. Το σύστημα αυτό απαιτεί, επίσης, πολύ χρόνο για να φορτώσει τα RDF δεδομένα και να δημιουργήσει τις κατάλληλες δομές. Τέλος, οι δομές που δημιουργούνται από το BigOWLIM δεν περιέχουν όλους τους δυνατούς συνδυασμούς indexes και έτσι, το σύστημα δεν αποδίδει καλά σε ερωτήματα που το object είναι μεταβλητή.

Ένα άλλο μεγάλο project που ασχολείται με την αποθήκευση και επερώτηση RDF δεδομένων, είναι το Virtuoso [14]. Για την αποθήκευση των RDF δεδομένων χρησιμοποιούνται δυο indexes τα οποία έχουν καλή απόδοση σε ερωτήσεις με καθορισμένο subject ή object. Και αυτή η υλοποίηση χρησιμοποιεί δομές που αποθηκεύονται στην μνήμη και τρέχει αποκλειστικά σε έναν υπολογιστή. Αυτό μειώνει τη δυνατότητα κλιμάκωσης της εφαρμογής και την χρήση της για βάσεις δεδομένων μεγαλύτερου όγκου.

Το RDF-3X [15] θεωρείται αυτή τη στιγμή ως η πιο γρήγορη υλοποίηση ενός συστήματος επερώτησης RDF δεδομένων. Για την αποθήκευση των RDF triple χρησιμοποιούνται και οι 6 διαφορετικοί συνδυασμοί indexes, με αποτέλεσμα το σύστημα να έχει πολύ καλή απόδοση σε όλα τα ήδη των SPARQL ερωτημάτων. Ακόμα χρησιμοποιεί ένα πολύ εξελιγμένο σύστημα συμπίεσης των δεδομένων του με αποτέλεσμα η διατήρηση των 6 διαφορετικών index να μην απαιτεί μεγάλη αύξηση στον χώρο που αυτά καταλαμβάνουν στο δίσκο. Αποτελεί, όμως, και αυτό εφαρμογή που τρέχει σε ένα μηχάνημα και άρα είναι ιδιαίτερα απαιτητικό σε μνήμη και επεξεργαστική ισχύ.

3.2) Εκτέλεση SPARQL ερωτημάτων με MapReduce

Το MapReduce είναι ένα από τα πιο υποσχόμενα πλαίσια παραλληλοποίησης που υπάρχουν αυτή τη στιγμή. Το πλαίσιο αυτό καθιστά εύκολη την δημιουργία εφαρμογών που τρέχουν ταυτόχρονα σε πολλούς υπολογιστές χωρίς να έχουν ιδιαίτερες απαιτήσεις από hardware. Έτσι έχει δημιουργηθεί μια πολλή ισχυρή τάση για χρησιμοποίηση του MapReduce σε εφαρμογές που επεξεργάζονται δεδομένα τεράστιου όγκου. Πολλές εταιρίες και ερευνητές έχουν συνδυάσει το Hadoop με παράλληλες βάσεις δεδομένων προσφέροντας έτσι κλιμακώσιμες υπηρεσίες. Αυτό έχει οδηγήσει τον τελευταίο καιρό σε μια μεγάλη διαμάχη μεταξύ των υποστηρικτών των παράλληλων σχεσιακών βάσεων δεδομένων και αυτών που υποστηρίζουν τη χρήση ενός συνδυασμού MapReduce και NoSQL βάσεων δεδομένων.

Όπως αναφέραμε και στην εισαγωγή στόχος της εργασίας μας είναι η δημιουργία ενός συστήματος που θα εκτελεί SPARQL ερωτήματα με χρήση MapReduce. Η βασικότερη προσπάθεια δημιουργίας ενός τέτοιου συστήματος παρουσιάζεται στο άρθρο [16]. Για την αποθήκευση των δεδομένων χρησιμοποιούνται αρχεία τα οποία είναι οργανωμένα με τέτοιο τρόπο ώστε να δημιουργούν ένα POS index. Για κάθε συνδυασμό predicate-object δημιουργείται ένα αρχείο που περιέχει όλα τα subject. Στην εργασία αυτή παρουσιάζεται αναλυτικά μια μέθοδος επιλογής του βέλτιστου πλάνου εκτέλεσης των join που είναι απαραίτητα για την διεκπεραίωση ενός ερωτήματος. Ο αλγόριθμος επιλογής είναι θεωρητικά τεκμηριωμένος και εξασφαλίζει εκτέλεση του ερωτήματος με τον μικρότερο

δυνατό αριθμό MapReduce job. Τα join αυτά εκτελούνται κατανεμημένα με υλοποίηση των αντίστοιχων map και reduce διεργασιών. Η βασική διαφορά αυτού του συστήματος από το δικό μας είναι το γεγονός ότι το σύστημά μας θα χρησιμοποιεί 3 διαφορετικά index τα οποία θα αποθηκεύονται σε πίνακες της HBase και όχι σε αρχεία. Η χρήση των 3 index θα μας επιτρέψει να έχουμε καλή απόδοση για όλα τα διαφορετικά είδη ερωτημάτων. Το σύστημα του άρθρου [16] διαθέτει ένα μόνο index γεγονός που σημαίνει ότι δεν θα έχει καλή απόδοση ειδικά σε ερωτήματα που περιέχουν μεταβλητό predicate.

Μια άλλη σχετική εργασία παρουσιάζεται στο άρθρο [20]. Σε αυτή την εργασία χρησιμοποιούνται πίνακες της HBase για αποθήκευση των δεδομένων σε 6 index. Χρησιμοποιούνται δηλαδή 6 πίνακες, ένας για κάθε διαφορετικό συνδυασμό των subject-predicate-object. Εδώ υλοποιούνται εφαρμογές MapReduce οι οποίες εκτελούν ένα join ανά job. Τα αποτελέσματα των πειραμάτων δείχνουν ότι το σύστημα έχει αρκετά καλή απόδοση, αλλά τα αποτελέσματα αναφέρονται σε σχετικά μικρές RDF βάσεις.

Τέλος, ένα παρόμοιο project είναι το BioMANTA [21] το οποίο χρησιμοποιεί RDF Molecules για την αποθήκευση των δεδομένων. Αυτό σημαίνει ότι διασπά τον συνολικό γράφο της βάσης, σε επιμέρους γράφους οι οποίοι αποτελούν τα molecules και αποθηκεύονται ξεχωριστά. Τα ερωτήματα διασπόνται σε ερωτήματα για τους αντίστοιχος γράφους που εκτελούνται παράλληλα. Ο τρόπος αυτός αποθήκευσης και επερώτησης διαφέρει αρκετά από τον τρόπο που έχουμε επιλέξει για την δικιά μας εργασία.

4) Σχεδίαση συστήματος

4.1) Μοντέλο αποθήκευσης δεδομένων

Για την αποθήκευση των δεδομένων θα χρησιμοποιήσουμε την NoSQL βάση της HBase. Η επιλογή αυτής της βάσης έγινε επειδή έχει αποδειχθεί εξαιρετικά επεκτάσιμη και συνδυάζεται εύκολα με το προγραμματιστικό πλαίσιο MapReduce. Θα πρέπει, λοιπόν, να δημιουργήσουμε έναν αποδοτικό τρόπο αποθήκευσης των δεδομένων που θα εκμεταλλεύεται όλα τα πλεονεκτήματα της HBase.

4.1.1) Αριθμός και τύπος index

Έχουμε επιλέξει να αποθηκεύουμε τα δεδομένα με τη χρήση indexes, συνεπώς μια σημαντική απόφαση που πρέπει να πάρουμε αφορά τον αριθμό και τον τύπο των index που θα χρησιμοποιήσουμε. Όπως έχουμε αναφέρει υπάρχουν πολλές απόψεις στο θέμα της επιλογής του αριθμού των index. Σε άλλες εφαρμογές χρησιμοποιείται ένα index με στόχο την εξοικονόμηση χώρου ενώ σε άλλες χρησιμοποιούνται ακόμα και έξι διαφορετικά με στόχο την καλύτερη δυνατή απόδοση του συστήματος.

Σε εφαρμογές που χρησιμοποιούν πολλά index βλέπουμε ότι παρουσιάζονται τα μειονεκτήματα του μεγάλου χώρου αποθήκευσης, καθώς και της αργής εισαγωγής και διαγραφής στοιχείων από τη βάση. Η αργή εισαγωγή και διαγραφή είναι αποτέλεσμα της ανάγκης για ανανέωση όλων των διαφορετικών index. Όσον αφορά τον όγκο αποθήκευσης, αυτός διπλασιάζεται, τριπλασιάζεται ή ακόμα και πενταπλασιάζεται ανάλογα με τον αριθμό των διαφορετικών index που επιλέγεται κάθε φορά.

Εφαρμογές με ένα index στηρίζονται στο γεγονός ότι τα query σπάνια περιέχουν BGP με μεταβλητό predicate. Έτσι επιλέγουν να υλοποιήσουν το πιο συχνά χρησιμοποιούμενο index. Έτσι εξασφαλίζουν μικρό χώρο αποθήκευσης και γρήγορες εισαγωγές νέων στοιχείων, όμως σε περιπτώσεις που τα ερωτήματα δεν περιέχουν τα αναμενόμενα query patterns, η απόδοση των συστημάτων μειώνεται δραματικά.

Υπάρχουν, βέβαια, και εφαρμογές που προσπαθούν να συμβιβάσουν τις παραπάνω ακραίες λύσεις διατηρώντας δυο ή τρία index. Τα index αυτά επιλέγονται με στόχο την καλύτερη δυνατή εξυπηρέτηση όλων των ειδών ερωτημάτων και καταφέρνουν να εξισορροπήσουν τα πλεονεκτήματα και τα μειονεκτήματα των δυο παραπάνω περιπτώσεων.

Κύριος στόχος της εφαρμογής μας είναι η γρήγορη εκτέλεση όλων των διαφορετικών SPARQL ερωτημάτων. Ο τελικός χρήστης μιας εφαρμογής του Semantic Web βλέπει μόνο το χρόνο εκτέλεσής και όχι το μέγεθος των δεδομένων που χρειάζονται για την εκτέλεσή της. Για αυτό το λόγο το σύστημά μας είναι σχεδιασμένο ώστε να έχει κατανεμημένη αποθήκευση των δεδομένων του. Αυτό σημαίνει ότι τα δεδομένα δεν θα αποθηκεύονται κεντρικά σε έναν υπολογιστή αλλά θα διαμοιράζονται και θα βρίσκονται αποθηκευμένα σε όλους τους υπολογιστές του cluster μας. Έτσι δεν απαιτείται η ύπαρξη ενός τεράστιου και πολύ ακριβού, κεντρικού συστήματος σκληρών δίσκων. Αρκεί κάθε υπολογιστής στο cluster να διαθέτει ένα τυπικό σκληρό δίσκο του εμπορίου. Έτσι κι αλλιώς η τεχνολογία των σκληρών δίσκων έχει αναπτυχθεί σε επίπεδο, στο οποίο απλοί και ιδιαίτερα οικονομικοί υπολογιστές διαθέτουν δίσκους με τεράστια χωρητικότητα.

Βέβαια, όπως είπαμε σε συστήματα με πολλά index παρουσιάζεται και το μειονέκτημα της αργής εισαγωγής και διαγραφής στοιχείων από τη βάση. Αυτή είναι μια πολύ σημαντική συνέπεια που πρέπει να λάβουμε σοβαρά υπόψη μας. Στη συγκεκριμένη εργασία, όμως, θα ασχοληθούμε μόνο με την διαδικασία απάντησης ερωτημάτων και όχι με

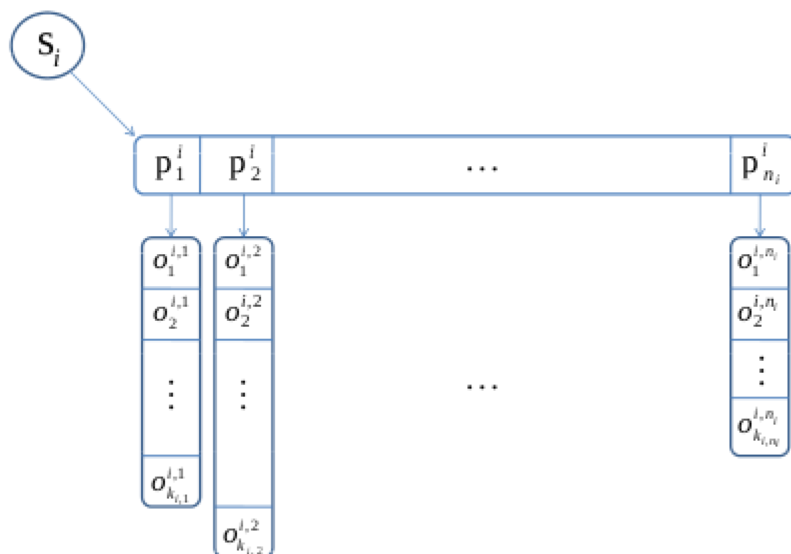
ενέργειες εισαγωγών και διαγραφών. Θα χρησιμοποιήσουμε, λοιπόν, ολοκληρωμένες βάσεις στις οποίες ο χρήστης θα έχει τη δυνατότητα εκτέλεσης μόνο ενεργειών αναζήτησης, μέσω ερωτημάτων SPARQL.

Τα ερωτήματα SPAQRL δημιουργούνται από συνδυασμούς BGP ερωτημάτων. Το BGP είναι ουσιαστικά ένα triple, το οποίο μπορεί να έχει μεταβλητές σε οποιαδήποτε θέση του. Για παράδειγμα το ερώτημα BGP

?x rdf:type ub:Student

απαιτεί την ανάκτηση από τη βάση όλων των μαθητών. Ένα τέτοιο ερώτημα μπορεί να περιέχει δυο ή ακόμα και τρεις μεταβλητές. Στην περίπτωση που έχουμε τρεις μεταβλητές το BGP ταιριάζει με όλα τα triples που περιέχονται στη βάση.

Θα χρησιμοποιήσουμε λοιπόν 3 διαφορετικά index, με τα οποία θα είμαστε σε θέση να ανακτήσουμε, άμεσα από τη βάση, τα triples που αντιστοιχούν σε οποιοδήποτε query pattern. Αυτά τα index θα είναι τα spo, osp, pos. Η παραπάνω ονομασία χρησιμοποιείται ευρέως για τον προσδιορισμό του τύπου των index. Για παράδειγμα το index spo, δημιουργεί μια ξεχωριστή δομή για κάθε διαφορετικό subject. Σε αυτή τη δομή περιέχονται όλα τα predicate που σχετίζονται με το συγκεκριμένο subject. Για κάθε τέτοιο predicate αποθηκεύονται όλα τα object που αντιστοιχούν στον συγκεκριμένο συνδυασμό subject-predicate. Η δομή ενός spo index φαίνεται στο παρακάτω σχήμα όπου για κάθε subject έχουμε μια λίστα από predicates και κάθε predicate δείχνει σε μια λίστα από objects [18].



Σχήμα 9: Index schema

Το index spo θα μπορούσε να χρησιμοποιηθεί για άμεση ανάκτηση των δεδομένων που ταιριάζουν με το pattern:

ub:GraduateStudent11 ub:takesCourse ?course

όμως δεν θα ήταν καθόλου χρήσιμο σε μια ερώτηση όπως:

?student ub:takesCourse ub:Course12

Για να απαντήσουμε την παραπάνω ερώτηση χρησιμοποιώντας μόνο το spo index, θα έπρεπε να ελέγξουμε όλη τη βάση μας, γεγονός που θα καθιστούσε το σύστημά μας

αναποτελεσματικό.

Για την επίτευξη του στόχου της απόδοσης πρέπει, λοιπόν, να διατηρούμε όλα τα απαραίτητα index καθώς και να γνωρίζουμε ποιό index πρέπει να συμβουλευτούμε κάθε φορά. Υπάρχουν οκτώ διαφορετικοί τύποι BGP ερωτημάτων οι οποίοι αντιστοιχούν στις δυνατές μεταθέσεις μιας, δυο ή τριών μεταβλητών μέσα σε ένα triple. Ακόμα υπάρχουν έξι διαφορετικοί συνδυασμοί από index που μπορούμε να δημιουργήσουμε. Οι συνδυασμοί των index προκύπτουν από τις διαφορετικές μεταθέσεις των s, p, o και είναι οι spo, pso, osp, sop , pos, ops. Στον παρακάτω πίνακα φαίνονται όλες οι διαφορετικές περιπτώσεις καθώς και ποιά από τα index μπορούμε να χρησιμοποιούμε, κάθε φορά, για ανακτήσουμε τα αντίστοιχα δεδομένα.

Query pattern			**Index**
Subject	**Predicate**	**Object**	
_	_	_	όλα
?	_	_	pos, ops
_	?	_	sop, osp
_	_	?	spo, pso
?	?	_	osp, ops
_	?	?	spo, sop
?	_	?	pos, pso
?	?	?	Όλα

Πίνακας 1: Αντιστοιχία query pattern-index

Στον πίνακα χρησιμοποιούμε το σύμβολο "?" για να δηλώσουμε την ύπαρξη κάποιας μεταβλητής στη συγκεκριμένη θέση. Με το σύμβολο "_" δηλώνουμε την ύπαρξη κάποιας σταθεράς. Για παράδειγμα το BGP:

?student ub:takesCourse ub:Course12

ανήκει στην κατηγορία:

Subject	**Predicate**	**Object**
?	_	_

Πίνακας 2: Συμβολισμός query pattern

Παρατηρώντας τον παραπάνω πίνακα διαπιστώνουμε, ότι χρησιμοποιώντας 3 από τα 6 index μπορούμε να έχουμε άμεση πρόσβαση στα δεδομένα όλων των δυνατών query. Αυτό, βέβαια, δεν σημαίνει ότι χρησιμοποιώντας τα 3 από τα 6 index θα έχουμε τη μέγιστη δυνατή απόδοση. Όπως έχει παρουσιαστεί και στο άρθρο του Hexastore [18], η χρήση όλων των διαφορετικών index μπορεί να έχει καλύτερα αποτελέσματα όταν κοιτάμε τον συνδυασμό δυο ή περισσότερων BGP και τον τρόπο με τον οποίο εκτελούμε τα join μεταξύ τους.

Ένα τέτοιο παράδειγμα φαίνεται από τα παρακάτω BGP με τα οποία ζητάμε όλους τους συμμαθητές του GraduteStudent11. Στην στήλη index φαίνονται τα αντίστοιχα index από τα οποία μπορούμε να ανακτήσουμε τα δεδομένα. Τα αποτελέσματα των δυο BGP

πρέπει να ενωθούν (join) με βάση τη μεταβλητή course.

BGP	Index
ub:GraduateStudent11 ub:takesCourse ?course	spo, pso
?student ub:takesCourse ?course	pos, ops

Πίνακας 3: Παράδειγμα ανάκτησης δεδομένων BGP από index

Όσον αφορά το πρώτο BGP δεν υπάρχει διαφορά ανάμεσα στα δυο indexes. Και τα δυο μας επιστρέφουν μια λίστα που περιέχει όλα τα μαθήματα, τα οποία παρακολουθεί ο συγκεκριμένος μαθητής. Τώρα θέλουμε να χρησιμοποιήσουμε τη λίστα των μαθημάτων ώστε να βρούμε όλους τους μαθητές που τα παρακολουθούν. Αν χρησιμοποιήσουμε το index ops, θα πρέπει για κάθε μάθημα της λίστας να εξετάσουμε όλη τη λίστα των predicate του μέχρι να βρούμε το takesCourse και να πάρουμε τη λίστα των subject που αντιστοιχούν σε αυτό. Αντίθετα, αν χρησιμοποιήσουμε το index pos, πάμε κατευθείαν στη δομή του predicate takesCourse και αναζητούμε σε αυτή τα μαθήματα της λίστας που προέκυψε από το πρώτο BGP.

Αρχικά, θα μπορούσε να πει κανείς, ότι με τη χρήση του pos θα είχαμε καλύτερη απόδοση αφού χρειάζεται να προσπελάσουμε τα στοιχεία μόνο μιας λίστας. Με τη χρήση του ops πρέπει να προσπελάσουμε ν λίστες, όπου ν ο αριθμός των στοιχείων της λίστας που προκύπτει από το πρώτο BGP.

Η υπόθεση, αυτή, δεν είναι πάντοτε ορθή. Σε περίπτωση που η λίστα του predicate takesCourse είναι πολύ μεγάλη και οι λίστες του index ops είναι μικρές, η απόδοση είναι καλύτερη με τη χρήση του ops. Μάλιστα στο συγκεκριμένο παράδειγμα οι υποθέσεις αυτές είναι πολύ λογικές, αφού η λίστα του predicate takesCourse περιέχει όλα τα δυνατά μαθήματα και άρα είναι πολύ μεγάλη. Οι λίστες του ops για κάθε μάθημα είναι μικρές, αφού ένα μάθημα έχει ένα περιορισμένο αριθμό predicate που του αντιστοιχούν.

Διαπιστώνουμε, έτσι ότι θα μπορούσαμε να πετύχουμε καλύτερη απόδοση με τη χρήση 6 index. Για να γίνει όμως αυτό το σύστημά μας θα πρέπει να κρατάει πολύ αναλυτικά στατιστικά στοιχεία ώστε να μπορεί να κάνει την σωστή επιλογή των index κάθε φορά. Μια τέτοια επιλογή θα μπορούσε να επηρεάσει ακόμα και τον τρόπο με τον οποίο το σύστημα επιλέγει τη βέλτιστη σειρά εκτέλεσης των join.

Πρέπει, επίσης, να εξετάσουμε και τη δυνατότητα υλοποίησης όλων αυτών των διαφορετικών τρόπων join στο πλαίσιο MapReduce. Με βάση, λοιπόν, τον τρόπο που έχουμε επιλέξει να υλοποιήσουμε τα join καθώς και το σύστημα που χρησιμοποιούμε για επιλογή του βέλτιστου πλάνου εκτέλεσης των join, διαπιστώνουμε ότι η πλήρης αξιοποίηση και των 6 index δεν θα είχε την αναμενόμενη βελτίωση απόδοσης. Επιλέγουμε λοιπόν, να χρησιμοποιήσουμε μόνο 3 από τα 6 index. Η επιλογή αυτή θα γίνει, ακόμα πιο κατανοητή όταν παρουσιάσουμε τον τρόπο υλοποίησης των join καθώς και τον αλγόριθμο επιλογής του βέλτιστου πλάνου εκτέλεσης των join.

Τα 3 index που επιλέξαμε να χρησιμοποιήσουμε είναι τα spo, pos, osp και ο τρόπος που τα χρησιμοποιούμε για να ανακτήσουμε τα δεδομένα των διαφόρων ερωτημάτων φαίνεται στον παρακάτω πίνακα. Ένα BGP με τρεις μεταβλητές απαιτεί την ανάκτηση όλων των triple που περιέχονται στη βάση. Αυτό σημαίνει ότι μπορούμε να απαντήσουμε χρησιμοποιώντας οποιοδήποτε από τα 3 index. Σε περιπτώσεις, όμως, που χρειάζεται να κάνουμε join του συγκεκριμένου BGP με κάποιο άλλο, φροντίζουμε να χρησιμοποιούμε το πιο αποδοτικό από αυτά. Ένα BGP με καμία μεταβλητή μπορεί και αυτό να ανακτηθεί αποδοτικά από όλα τα index. Για αυτό το ερώτημα, όμως, θα ήταν επιθυμητή η χρήση ενός index με υψηλή επιλεκτικότητα. Ένα τέτοιο index είναι σε πολλές περιπτώσεις το osp, αφού συνήθως δεν υπάρχουν πολλά properties που να συνδέουν ένα συγκεκριμένο ζεύγος object-subject. Έτσι οι τελικές λίστες των predicate είναι μικρές και η αναζήτηση σε αυτές είναι

αποδοτική.

Query pattern			Index
Subject	Predicate	Object	
_	_	_	όλα
?	_	_	pos
_	?	_	osp
_	_	?	spo
?	?	_	osp
_	?	?	spo
?	_	?	pos
?	?	?	όλα

Πίνακας 4: Αντιστοιχία query pattern-index

4.1.2) Αποθήκευση των index σε πίνακες της HBase

Για την αποθήκευση των δεδομένων μας, θέλουμε να χρησιμοποιήσουμε την κατανεμημένη NoSQL βάση δεδομένων της HBase. Για να καταφέρουμε κάτι τέτοιο, θα πρέπει να βρούμε ένα τρόπο που θα μας επιτρέψει να δημιουργήσουμε index, σαν αυτά που περιγράφηκαν παραπάνω, με χρήση πινάκων HBase.

Όπως περιγράψαμε και στην εισαγωγή, η βασικότερη μονάδα που αποθηκεύεται σε έναν πίνακα HBase είναι το cell το οποίο προσδιορίζεται από την τριπλέτα "row", "column_family:qualifier". Σε κάθε cell αποθηκεύεται μια τιμή. Η HBase λέγεται και keyvalue store, δηλαδή, χρησιμοποιείται για αποθήκευση ζευγαριών key/value. Το key αποτελείται από την τετράδα ("row", "column_family:qualifier", "Timestamp"), ενώ το value είναι η τιμή που περιέχεται στο cell.

Ο σχεδιασμός της HBase μας διευκολύνει ιδιαίτερα στη δημιουργία index για την RDF βάση μας. Για την διευκόλυνση της ανάκτησης μιας συγκεκριμένης γραμμής του πίνακα, η HBase διατηρεί B+ δέντρα, τα οποία αναφέρονται στο κλειδί κάθε γραμμής και μπορούν εύκολα να προσδιορίσουν σε ποιόν κόμβο καθώς και σε ποιό αρχείο βρίσκεται η συγκεκριμένη γραμμή. Μια ακόμη ενδιαφέρουσα ιδιότητα της HBase είναι το γεγονός ότι ταξινομεί και αποθηκεύει τις γραμμές ενός πίνακα με λεξικογραφική σειρά. Αυτό μας επιτρέπει να εκτελούμε γρήγορα ερωτήματα που αναφέρονται σε διαστήματα κλειδιών. Μπορούμε, δηλαδή, να ανακτήσουμε άμεσα όλες τις γραμμές που περιέχονται σε ένα διάστημα της μορφής [start_key, end_key). Για να εκτελέσουμε τέτοια ερωτήματα, αρκεί η εύρεση, μέσω του B+ δέντρου, του αρχείου που περιέχει το start_key. Στη συνέχεια ανακτούμε όλες τις γραμμές που είναι αποθηκευμένες μετά από αυτό το κλειδί σταματώντας όταν κάποια γραμμή βρεθεί εκτός του διαστήματος.

Εκτός από τα παραπάνω, η HBase δίνει τη δυνατότητα να δημιουργήσουμε δευτερεύοντα B+ δέντρα που αναφέρονται και στις στήλες ενός πίνακα. Αυτά τα δευτερεύοντα B+ δέντρα ονομάζονται bloom filters και προσφέρουν επιπλέον δυνατότητες δεικτοδότησης για την ευκολότερη πρόσβαση στα δεδομένα ενός πίνακα. Η διατήρηση, βέβαια, τέτοιων δομών απαιτεί επιπλέον χώρο αποθήκευσης και προσθέτει επιπλέον χρόνο στην δημιουργία και ανανέωση του πίνακα. Η χρήση των bloom filter έχει καλύτερη εφαρμογή σε μεγάλους πίνακες, με γραμμές οι οποίες περιέχουν πάρα πολλές στήλες.

Χρησιμοποιώντας αυτές τις ιδιότητες μπορούμε εύκολα να σχεδιάσουμε πίνακες, οι

οποίοι θα λειτουργούν όπως τα index που περιγράφηκαν προηγουμένως. Παρακάτω θα περιγράψουμε αναλυτικά τον σχεδιασμό ενός spo index. Η δημιουργία των υπόλοιπων index είναι αντίστοιχη. Θα δημιουργήσουμε, λοιπόν, έναν πίνακα που θα περιέχει τους συνδυασμούς subject-predicate-object και στον οποίο θα μπορούμε να έχουμε αποδοτική πρόσβαση, τόσο διαθέτοντας μόνο το subject όσο και διαθέτοντας ένα συνδυασμό subject-predicate. Έχουμε σχεδιάσει δυο διαφορετικούς τύπους τέτοιων πινάκων οι οποίοι παρουσιάζονται παρακάτω.

4.1.3) S_PO index

Στην ενότητα αυτή θα αναλύσουμε το S_PO index, αλλά ουσιαστικά παρουσιάζουμε τις ιδιότητες ενός συγκεκριμένου μοντέλου πίνακα που μπορεί να χρησιμοποιηθεί σαν index. Αντίστοιχα, μπορούμε να δημιουργήσουμε και τους πίνακες P_OS και O_SP οι οποίοι θα έχουν και αυτοί τις ιδιότητες που περιγράφονται παρακάτω.

4.1.3.1) Μοντέλο πίνακα

Κάθε γραμμή του πίνακα S_PO αναφέρεται σε ένα συγκεκριμένο subject το οποίο χρησιμοποιείται και ως rowid. Ο πίνακας περιέχει ένα μόνο column family μέσα στο οποίο βρίσκονται όλες οι στήλες. Ως qualifier, για τις στήλες, χρησιμοποιούμε το predicate που συνδέεται με το συγκεκριμένο subject. Τέλος στο value τοποθετούμε μια λίστα με όλα τα διαφορετικά object που συνδέονται με το συγκεκριμένο συνδυασμό subject-predicate. Τα δεδομένα ενός τέτοιου πίνακα φαίνονται παρακάτω.

RowId	ColumnId	Value
GraduateStudent11	memberOf	University1
GraduateStudent11	takesCourse	Course10_Course4_Course9
GraduateStudent11	Type	GraduateStudent
GraduateStudent12	memberOf	University3
GraduateStudent12	takesCourse	Course15_Course7
GraduateStudent12	Type	GraduateStudent

Πίνακας 5: S_PO index

Τα είδη των ερωτημάτων που το index πρέπει να μπορεί να απαντήσει αποδοτικά είναι τα (_, _, ?) και (_, ?, ?). Οι συμβολισμοί αυτοί αναφέρονται σε BGP patterns τα οποία έχουν κάποια μεταβλητή στη θέση του "?" και κάποια σταθερά στη θέση του "_".

Στην πρώτη περίπτωση, (_, _, ?), μπορεί να έχουμε ένα ερώτημα που ζητάει όλα τα μαθήματα, τα οποία παρακολουθεί ο μεταπτυχιακός φοιτητής GraduateStudent11. Ένα τέτοιο ερώτημα έχει τη μορφή:

ub:GraduateStudent11 ub:takesCourse ?course

Για να την απαντήσουμε αρκεί να πάρουμε τα δεδομένα που βρίσκονται στο cell με rowid:"GraduateStudent11" και qualifier:"takesCourse". Η HBase μπορεί να εκτελέσει αποδοτικά μια τέτοια αναζήτηση, αφού αρχικά βρίσκει μέσω του B+ δέντρου την τοποθεσία της γραμμής και στη συνέχεια μέσω του bloom filter ή δυαδικής αναζήτησης εντοπίζει την ζητούμενη στήλη. Οι στήλες αποθηκεύονται και αυτές ταξινομημένες με λεξικογραφική

σειρά γεγονός που μας επιτρέπει να κάνουμε δυαδική και όχι γραμμική αναζήτηση για τον εντοπισμό τους. Στην περίπτωση βέβαια που έχουμε δημιουργήσει bloom filters, η αναζήτηση αυτή γίνεται άμεσα μέσω μιας δενδρικής δομής.

Το δεύτερο είδος ερωτήματος, στο οποίο το index πρέπει να απαντάει αποδοτικά είναι το (_, ?, ?). Ένα παράδειγμα αντίστοιχου ερωτήματος είναι:

ub:GraduateStudent12 ?x ?y

Το ερώτημα αυτό ζητάει όλες τις διαθέσιμες πληροφορίες, που υπάρχουν στη βάση μας, για τον GraduateStudent12. Για να ανακτήσουμε όλες τις απαραίτητες πληροφορίες, αρκεί να εντοπίσουμε τα δεδομένα ολόκληρης της γραμμής με rowid: “GraduateStudent12”. Αυτό είναι ιδιαίτερα αποδοτικό στην HBase αφού εντοπίζουμε άμεσα την συγκεκριμένη γραμμή μέσω του B+ δέντρου.

Ακόμα, το index μπορεί να χρησιμοποιηθεί για απάντηση ερωτημάτων της μορφής (?, ?, ?) και (_, _,). Για το ερώτημα (?, ?, ?), αρκεί να κάνουμε μια ανάγνωση ολόκληρου του πίνακα. Αντίθετα για το (_, _, _), βρίσκουμε πρώτα το cell προσδιορίζοντας τα rowid και qualifier, στη συνέχεια διαβάζουμε το αντίστοιχο value και βλέπουμε αν εκεί περιέχεται το συγκεκριμένο object. Η αναζήτηση αυτή είναι γραμμική γιατί η λίστα των object δεν είναι ταξινομημένη.

4.1.3.2) Κατανομή των δεδομένων στο cluster

Το σύστημά μας θα εκτελεί, κατανεμημένα, τα SPARQL ερωτήματα και άρα τα index που θα χρησιμοποιήσουμε, πρέπει να βοηθούν στον αποδοτικό καταμερισμό των δεδομένων κάθε BGP ερωτήματος. Έστω, λοιπόν, ότι θέλουμε να εκτελέσουμε, με χρήση του MapReduce, το join:

ub:GraduateStudent11 ub:takesCourse ?course
?student ub:takesCourse ?course

Θα πρέπει να βρούμε τα δεδομένα που αντιστοιχούν στα δυο ερωτήματα και να τα μοιράσουμε αποδοτικά σε όλους τους υπολογιστές του cluster. Αυτό πρέπει να γίνει χωρίς να χρειαστεί να διαβάσουμε όλα τα δεδομένα και στη συνέχεια να τα μοιράσουμε. Θα διαμορφώσουμε, λοιπόν, το index ώστε να συμβάλλει σε αυτό τον διαχωρισμό.

Για να πετύχουμε κάτι τέτοιο μπορούμε να χωρίσουμε κάθε γραμμή του προηγούμενου πίνακα σε επιμέρους γραμμές χρησιμοποιώντας ένα offset στο τέλος του rowid. Ο νέος πίνακας θα έχει την παρακάτω μορφή

RowId_Offset	ColumnId	Value
GraduateStudent11_1	memberOf	University1
GraduateStudent11_1	takesCourse	Course10_Course9
GraduateStudent11_2	takesCourse	Course4
GraduateStudent11_2	Type	GraduateStudent
GraduateStudent12_1	memberOf	University3
GraduateStudent12_1	takesCourse	Course15
GraduateStudent12_2	takesCourse	Course7
GraduateStudent12_2	Type	GraduateStudent

Πίνακας 6: S_PO index με offset

Η μέγιστη τιμή του offset που χρησιμοποιείται μπορεί να αντιστοιχεί στον αριθμό των κόμβων του cluster μας ή να είναι κάποιο πολλαπλάσιό του. Κατά την εισαγωγή κάθε RDF triple στον πίνακα, μπορούμε να επιλέξουμε έναν τυχαίο αριθμό στο διάστημα [0, max_offset] για να προσδιορίσουμε το offset που θα χρησιμοποιήσουμε. Η επιλογή τυχαίου αριθμού μας βοηθάει να χωρίσουμε αποδοτικά τα δεδομένα μας χωρίς να διαθέτουμε επιπλέον γνώση για αυτά και μας εξασφαλίζει ισοκατανομή των δεδομένων στις διαφορετικές γραμμές.

Με χρήση του παραπάνω index μπορούμε να μοιράσουμε άμεσα τα δεδομένα όλων των ερωτημάτων. Για παράδειγμα, αν θέλουμε να μοιράσουμε τα δεδομένα του BGP

ub:GraduateStudent11 ub:takesCourse ?course

ο κάθε κόμβος του cluster μας να επεξεργάζεται τις γραμμές που ανήκουν στο διάστημα [GraduateStudent11_startOffset, GraduateStudent11_endOffset). Ως qualifier χρησιμοποιείται όπως και πριν το "takesCourse". Αντίστοιχα διαχωρίζονται και τα δεδομένα στην περίπτωση που έχουμε ένα query pattern της μορφής (_, ?, ?).

Πρέπει να τονίσουμε το γεγονός ότι η τοποθέτηση του offset στο τέλος του rowid συμβάλλει στην τοπικότητα των δεδομένων. Αυτό σημαίνει ότι τα δεδομένα με το ίδιο ή γειτονικά offset θα βρίσκονται πολύ κοντά μεταξύ τους. Μειώνεται έτσι ο φόρτος μεταφοράς δεδομένων στο δίκτυο αφού ο κάθε κόμβος διαβάζει διαδοχικά δεδομένα και όχι δεδομένα που βρίσκονται σε εντελώς διαφορετικές περιοχές. Η ιδιότητα της τοπικότητας των αναγνώσεων βοηθάει και στην ανάθεση των δεδομένων στον κόμβο, στον οποίο αυτά είναι αποθηκευμένα ή τουλάχιστον σε κάποιον γειτονικό.

4.1.4) SP_O index

Στην ενότητα αυτή θα αναλύσουμε ένα άλλο μοντέλο αποθήκευσης index σε πίνακα HBase. Για αυτό το μοντέλο θα χρησιμοποιούμε ονόματα που έχουν το χαρακτήρα "_" στην τρίτη θέση. Η προέλευση αυτής της ονομασίας θα γίνει εμφανής από την ανάλυση που ακολουθεί. Για την ανάλυση θα χρησιμοποιήσουμε το SP_O index, αλλά ουσιαστικά θα παρουσιάσουμε τις ιδιότητες όλων των index αυτής της κατηγορίας. Αντίστοιχα, μπορούμε να δημιουργήσουμε και τους πίνακες PO_S και OS_P οι οποίοι θα έχουν και αυτοί τις ίδιες ιδιότητες.

4.1.4.1) Μοντέλο πίνακα

Κάθε γραμμή του πίνακα SP_O αναφέρεται σε ένα συγκεκριμένο ζεύγος subject-predicate το οποίο χρησιμοποιείται και ως rowid. Ο πίνακας περιέχει ένα μόνο column family μέσα στο οποίο βρίσκονται όλες οι στήλες. Ως qualifier, για τις στήλες, χρησιμοποιούμε το object που συνδέεται με το συγκεκριμένο συνδυασμό subject-predicate. Τα δεδομένα ενός τέτοιου πίνακα φαίνονται παρακάτω.

RowId	ColumnId
GraduateStudent11_memberOf	University1
GraduateStudent11_takesCourse	Course4
GraduateStudent11_takesCourse	Course9

GraduateStudent11_takesCourse	Course10
GraduateStudent11_type	GraduateStudent
GraduateStudent12_memberOf	University3
GraduateStudent12_takesCourse	Course7
GraduateStudent12_takesCourse	Course15
GraduateStudent12_type	GraduateStudent

Πίνακας 7: SP_O index

Όπως έχουμε αναφέρει σε προηγούμενη ενότητα ένα spo index θα πρέπει να είναι σε θέση να απαντήσει αποδοτικά τα (_, _, ?) και (_, ?, ?) BGP patterns.

Στην πρώτη περίπτωση, (_, _, ?), μπορεί να έχουμε ένα ερώτημα που ζητάει όλα τα μαθήματα, τα οποία παρακολουθεί ο μεταπτυχιακός φοιτητής GraduateStudent11. Ένα τέτοιο ερώτημα έχει τη μορφή:

ub:GraduateStudent11 ub:takesCourse ?course

Για να την απαντήσουμε αρκεί να πάρουμε όλα τα δεδομένα της γραμμής με rowid "GraduateStudent11_takesCourse". Οι τιμές τις μεταβλητής course θα προκύψουν από όλες τις διαφορετικές στήλες που περιέχονται στη γραμμή. Μια τέτοια αναζήτηση είναι εξαιρετικά αποδοτική αφού αναζητούμε το B+ δέντρο της HBase για να βρούμε την τοποθεσία της γραμμής. Δεν χρειάζεται να κάνουμε επιπλέον αναζήτηση σε επίπεδο στήλης όπως χρειαζόταν με τα προηγούμενα index.

Το δεύτερο είδος ερωτήματος, στο οποίο το index πρέπει να απαντάει αποδοτικά είναι το (_, ?, ?). Ένα παράδειγμα αντίστοιχου ερωτήματος είναι:

ub:GraduateStudent12 ?x ?y

Το ερώτημα αυτό ζητάει όλες τις διαθέσιμες πληροφορίες, που υπάρχουν στη βάση μας, για τον GraduateStudent12. Για να ανακτήσουμε όλες τις απαραίτητες πληροφορίες, εκτελούμε ένα range scan στην HBase με διάστημα [GraduateStudent12, increment(GraduateStudent12)). Η συνάρτηση increment μας δίνει την τιμή του αμέσως επόμενου, λεξικογραφικά, κλειδιού. Και σε αυτή την περίπτωση η πρόσβαση είναι ιδιαίτερα γρήγορη αφού, αρχικά βρίσκουμε την πρώτη γραμμή του διαστήματος μέσω του B+δέντρου και στη συνέχεια διαβάζουμε διαδοχικά όλες τις γραμμές μέχρι να βρούμε μια που είναι μεγαλύτερη από το τέλος του διαστήματος.

Ακόμα, το index μπορεί να χρησιμοποιηθεί για απάντηση ερωτημάτων της μορφής (?, ?, ?) και (_, _,). Για το ερώτημα (?, ?, ?), αρκεί να κάνουμε μια ανάγνωση ολόκληρου του πίνακα. Το ερώτημα (_, _, _), είναι εξαιρετικά αποδοτικό αφού αντιστοιχεί σε ένα ερώτημα ύπαρξης ή όχι ενός συγκεκριμένου cell.

4.1.4.2) Κατανομή των δεδομένων στο cluster

Το σύστημά μας θα εκτελεί, κατανεμημένα, τα SPARQL ερωτήματα και άρα τα index που θα χρησιμοποιήσουμε, πρέπει να βοηθούν στον αποδοτικό καταμερισμό των δεδομένων κάθε BGP ερωτήματος. Έστω, λοιπόν, ότι θέλουμε να εκτελέσουμε, με χρήση του MapReduce, το join:

ub:GraduateStudent11 ub:takesCourse ?course

?student ub:takesCourse ?course

Θα πρέπει να βρούμε τα δεδομένα που αντιστοιχούν στα δυο ερωτήματα και να τα μοιράσουμε αποδοτικά σε όλους τους υπολογιστές του cluster. Αυτό πρέπει να γίνει χωρίς να χρειαστεί να διαβάσουμε όλα τα δεδομένα και στη συνέχεια να τα μοιράσουμε. Θα διαμορφώσουμε, λοιπόν, το index ώστε να συμβάλλει σε αυτό τον διαχωρισμό.

Για να πετύχουμε κάτι τέτοιο μπορούμε να χωρίσουμε κάθε γραμμή του προηγούμενου πίνακα σε επιμέρους γραμμές χρησιμοποιώντας ένα offset στο τέλος του rowid. Αυτή τη φορά τοποθετούμε το offset στην αρχή του rowed.Ο νέος πίνακας θα έχει την παρακάτω μορφή.

RowId	ColumnId
1_GraduateStudent11_memberOf	University1
1_GraduateStudent11_takesCourse	Course4
1_GraduateStudent11_type	GraduateStudent
1_GraduateStudent12_takesCourse	Course7
1_GraduateStudent12_takesCourse	Course15
2_GraduateStudent11_takesCourse	Course9
2_GraduateStudent11_takesCourse	Course10
2_GraduateStudent12_memberOf	GraduateStudent
2_GraduateStudent12_type	GraduateStudent

Πίνακας 8: SP_O index με offset

Η μέγιστη τιμή του offset που χρησιμοποιείται μπορεί να αντιστοιχεί στον αριθμό των κόμβων του cluster μας ή να είναι κάποιο πολλαπλάσιό του. Κατά την εισαγωγή κάθε RDF triple στον πίνακα, μπορούμε να επιλέξουμε έναν τυχαίο αριθμό στο διάστημα [0, max_offset] για να προσδιορίσουμε το offset που θα χρησιμοποιήσουμε. Η επιλογή τυχαίου αριθμού μας βοηθάει να χωρίσουμε αποδοτικά τα δεδομένα μας χωρίς να διαθέτουμε επιπλέον γνώση για αυτά και μας εξασφαλίζει ισοκατανομή των δεδομένων στις διαφορετικές γραμμές.

Με χρήση του παραπάνω index μπορούμε να μοιράσουμε άμεσα τα δεδομένα όλων των ερωτημάτων. Επιλέγουμε ο αριθμός max_offset να είναι ίδιος με τον διαφορετικό αριθμό των mapper που τρέχουν στο σύστημά μας. Για παράδειγμα, αν θέλουμε να μοιράσουμε τα δεδομένα του BGP

ub:GraduateStudent11 ub:takesCourse ?course

ο κάθε mapper του cluster μας, να επεξεργάζεται τη γραμμή id_GraduateStudent11_takesCourse. Αντίστοιχα διαχωρίζονται και τα δεδομένα στην περίπτωση που έχουμε ένα query pattern της μορφής (_, ?, ?).

Πρέπει να τονίσουμε το γεγονός ότι η τοποθέτηση του offset στην αρχή του rowid τοποθετεί τα κομμάτια της γραμμής σε διαφορετικά regions, ένα για κάθε mapper. Ο κάθε mapper επεξεργάζεται τη δικιά του γραμμή η οποία βρίσκεται στο δικό του region. Έτσι καταφέρνουμε να μειώσουμε τα δεδομένα που μεταφέρονται μέσω δικτύου. Ο mapper δεν χρειάζεται να πάρει τα δεδομένα εισόδου του από κάποιον άλλο κόμβο, απλά τα διαβάζει τοπικά από το δικό του region του index.

4.2) Σχεδιασμός εκτέλεσης MapReduce join

4.2.1) Εισαγωγή

Βασικό στοιχείο της υλοποίησης ενός συστήματος που απαντάει SPARQL ερωτήματα είναι ο σχεδιασμός του τρόπου με τον οποίο θα εκτελούνται τα διάφορα join μεταξύ των BGP patterns. Το σύστημά μας θα εκτελεί τέτοια join με τη χρήση της τεχνικής του MapReduce. Θα πρέπει λοιπόν να προδιαγράψουμε τον αλγόριθμο που θα χρησιμοποιήσουμε καθώς και όλες τις επιπλέον λεπτομέρειες ενός τέτοιου συστήματος.

Για το σκοπό της εκτέλεσης των κατανεμημένων join θα χρησιμοποιήσουμε δυο διαφορετικές μεθοδολογίες. Κάθε μεθοδολογία έχει τα δικά της πλεονεκτήματα και μειονεκτήματα. Οι δυο αυτές μεθοδολογίες θα είναι συμπληρωματικές στο σύστημά μας και θα μπορούμε να επιλέξουμε κάθε φορά την κατάλληλη για την εκτέλεση του κάθε join. Η πρώτη μεθοδολογία ονομάζεται FullInputJoin ενώ η δεύτερη SingleInputJoin.

Για να μπορέσουν οι δυο αυτές μεθοδολογίες να συνδυαστούν και να λειτουργήσουν μαζί θα πρέπει η είσοδος και η έξοδός τους να έχουν την ίδια μορφή. Αρχικά, λοιπόν, θα παρουσιάσουμε τις προδιαγραφές εισόδου και εξόδου που θα χρησιμοποιηθούν και στη συνέχεια, θα αναλύσουμε τους αλγορίθμους εκτέλεσης των join.

4.2.2) Προδιαγραφές εισόδου/εξόδου

Για να εκτελέσουμε ένα SPARQL ερώτημα χρειάζεται να εκτελέσουμε πολλαπλά join διαφορετικών μεταβλητών. Τα αποτελέσματα του ενός join, λοιπόν, χρησιμοποιούνται ως είσοδο στο επόμενο join. Γι' αυτό το λόγο επιλέγουμε να έχουμε ίδιες προδιαγραφές τόσο στην είσοδο, όσο και στην έξοδο των MapReduce εργασιών μας. Η χρήση κοινών προδιαγραφών εισόδου/εξόδου θα μειώσει την πολυπλοκότητα του συστήματος και θα διευκολύνει τη συνεργασία μεταξύ διαδοχικών join.

Οι κοινές προδιαγραφές εισόδου/εξόδου πρέπει να προσδιοριστούν με κριτήριο τόσο την εύκολη ανάγνωση από τα index της HBase, όσο και την δυνατότητα αναπαράστασης όλων των διαφορετικών αποτελεσμάτων που μπορεί να προκύψουν από ένα join.

Τα αποτελέσματα που δημιουργούνται από ένα join είναι όλοι οι συνδιαμοί των τιμών των μεταβλητών που περιέχονται σε αυτό και ικανοποιούν όλα τα BGP queries του. Για παράδειγμα τα αποτελέσματα του join

ub:GraduateStudent11 ub:takesCourse ?course
?student ub:takesCourse ?course

θα είναι όλα τα διαφορετικά ζευγάρια τιμών των μεταβλητών course και student, που ικανοποιούν και τα δυο BGP queries. Τα αποτελέσματα που αντιστοιχούν σε αυτές τις μεταβλητές ονομάζονται και bindings των μεταβλητών. Επειδή όπως είπαμε θα εκτελούμε join σε διαφορετικές μεταβλητές, τα οποία θα διαδέχονται το ένα το άλλο, είναι φυσιολογικό, κάποια πιο σύνθετα join, να περιέχουν και πολύ περισσότερες μεταβλητές στην έξοδό τους. Για κάθε μια από αυτές τις μεταβλητές θα πρέπει να βρούμε ένα τρόπο να αποθηκεύουμε στο αρχείο εξόδου τις αντίστοιχες τιμές της (bindings).

Όσον αφορά την είσοδο ενός join, αυτή αποτελείται από τις επιμέρους εισόδους όλων των BGP queries που περιέχει. Τα δεδομένα που αντιστοιχούν σε ένα τέτοιο BGP query αποτελούνται και αυτά από ένα σύνολο συνδυασμών από τιμές για όλες τις μεταβλητές που υπάρχουν στο query. Η είσοδος, λοιπόν, ενός join μπορεί και αυτή να

αποτελείται από ένα αρχείο με όλους τους απαραιτήτους συνδυασμούς από bindings.

Κάθε ξεχωριστό ζευγάρι key/value του αρχείου εισόδου/εξόδου πρέπει να περιέχει ένα τέτοιο συνδυασμό απο binding όλων των απαραίτητων μεταβλητών. Ο αριθμός των διαφορετικών μεταβλητών που περιέχονται σε έναν τέτοιο συνδυασμό, μπορεί να είναι οσοδήποτε μεγάλος και εξαρτάται πολύ από τον αριθμό των προηγούμενων join που έχουν γίνει. Ο αιθμός των μεταβλητών που βρίσκονται στην έξοδο των διαδοχικών join συνεχώς αυξάνεται μέχρι το τελευταίο join του οποίου η έξοδος πρέπει να περιέχει όλες τις διαφορετικές μεταβλητές που παρουσιάζονται στο SPARQL query.

Όπως έχουμε αναφέρει η είσοδος ενός MapReduce job πρέπει να έχει τη μορφή ζευγαριών key/value. Έτσι επιλέγουμε να αποθηκεύσουμε όλες τις απαραίτητες πληροφορίες στο value του ζευγαριού. Τα ζευγάρια θα αποθηκεύονται σε ξεχωριστές γραμμές αρχείων κειμένου και άρα το key θα προκύπτει, όπως ορίζεται και από το TextInputFormat του Hadoop, από το byteid της κάθε γραμμής.

Η μορφή που θα έχει το πεδίο value αυτών των ζευγαριών θα ακολουθεί τον παρακάτω γενικό κανόνα:

jpat var1$$bindings var2$$bindings .... varN$$bindings

όπου var1...N είναι οι διαφορετικές μεταβλητές που συνδέονται με το συγκεκριμένο join. Στη θέση του bindings τοποθετούμε μια ή περισσότερες τιμές της αντίστοιχης μεταβλητής. Το jpat είναι ένα id μοναδικό για κάθε BGP pattern ή έξοδο join, το οποίο μας βοηθά να αναγνωρίσουμε την προέλευση του κάθε ζεύγους key/value. Για παράδειγμα, τα παρακάτω αποτελέσματα είναι ένα μέρος των αποτελεσμάτων του join που παρουσιάσαμε προηγουμένως.

P2 ?course$$Course1 ?student$$GraduateStudent11_ GraduateStudent3_ GraduateStudent7
P2 ?course$$Course2 ?student$$GraduateStudent1_ GraduateStudent5
P2 ?course$$Course3 ?student$$GraduateStudent2_ GraduateStudent3_ GraduateStudent6

Για να μετατρέψουμε αυτή την μορφή εξόδου στην κλασσική μορφή εξόδου που χρησιμοποιείται από όλα τα SPARQL engines, αρκεί για κάθε γραμμή να βρούμε όλους τους διαφορετικούς συνδυασμούς τιμών των μεταβλητών. Αυτή η έξοδος μπορεί να παρουσιαστεί όπως στον παρακάτω πίνακα. Ακόμα και σε περιπτώσεις που ο αριθμός των μεταβλητών εξόδου είναι πολύ μεγάλος, μπορούμε εύκολα να μετατρέψουμε την συγκεκριμένη μορφή εξόδου σε αυτή που χρησιμοποιείται συνήθως. Όσον αφορά την μετατροπή της εξόδου σε xml αρχεία, αυτή είναι ακόμη πιο εύκολη. Η ομαδοποίηση που προκαλεί στους συνδυασμούς των μεταβλητών εξόδου μπορεί να αποτυπωθεί άμεσα στις δενδρικές δομές των xml αρχείων. Η μετατροπή αυτή βέβαια δεν έχει μεγάλη επίδραση στην απόδοση του συστήματός μας, αφού εκτελείται μόνο μια φορά στο τελευταίο join.

?course	?student
Course1	GraduateStudent11
Course1	GraduateStudent3
Course1	GraduateStudent7
Course2	GraduateStudent1
Course2	GraduateStudent5
Course3	GraduateStudent2
Course3	GraduateStudent3

Course3	GraduateStudent6

Πίνακας 9: Μορφή εξόδου join

Εκτός, όμως, από την δυνατότητα περιγραφής όλων των δυνατών εξόδων ενός join οι προδιαγραφές εισόδου/εξόδου θα πρέπει να διευκολύνουν και την ανάγνωση δεδομένων εισόδου από τη βάση. Τα δεδομένα της βάσης μας είναι αποθηκευμένα στα index της hbase, οπότε θα πρέπει να βρούμε έναν τρόπο να μετατρέψουμε αποδοτικά τα δεδομένα των index στην παραπάνω μορφή. Αυτή η μετατροπή είναι ιδιαίτερα εύκολη. Για παράδειγμα, έστω ότι έχουμε το BGP query

ub:GraduateStudent11 ?p ?o

και τα δεδομένα που βρίσκονται στο index SP_O φαίνονται στο παρακάτω πίνακα

RowId	ColumnId
GraduateStudent11_memberOf	University1
GraduateStudent11_takesCourse	Course4
GraduateStudent11_takesCourse	Course9
GraduateStudent11_takesCourse	Course10
GraduateStudent11_type	GraduateStudent
GraduateStudent12_memberOf	University3
GraduateStudent12_takesCourse	Course7
GraduateStudent12_takesCourse	Course15
GraduateStudent12_type	GraduateStudent

Πίνακας 10: Δεδομένα S_PO

Όπως εξηγήσαμε σε προηγούμενη ενότητα, για να προσδιορίσουμε τα δεδομένα που αντιστοιχούν σε ένα τέτοιο query αρκεί να κάνουμε ένα range scan με κλίμακα [GraduateStudent11, GraduateStudent11). Για κάθε γραμμή που μας επιστρέφει το scan μπορούμε να δημιουργήσουμε ένα key/value εισόδου. Η είσοδος που θα προέκυπτε από τα δεδομένα του index θα ήταν:

P1 ?p$$memberOf ?s$$University1
P1 ?p$$takesCourse ?s$$Course4
P1 ?p$$takesCourse ?s$$Course9
P1 ?p$$takesCourse ?s$$Course10
P1 ?p$$type ?s$$GraduateStudent

Όπως μπορούμε να παρατηρήσουμε η παραπάνω είσοδος θα μπορούσε να βελτιωθεί, αν ομαδοποιούσαμε τα bindings που αντιστοιχούν στην ιδιότητα takesCourse. Αυτό θα μπορούσε να γίνει εύκολα με χρήση ενός buffer, ο οποίος θα αποθήκευε όλα τα διαδοχικά bindings της μεταβλητής s που αντιστοιχούν στην ίδια τιμή της μεταβλητής p. Όταν ο buffer γεμίσει ή όταν η τιμή του p αλλάξει δημιουργούμε ένα καινούργιο ζευγάρι key/value. Η συγκέντρωση όλων των bindings του s σε ένα key/value δεν θα ήταν σωστή λύση, αφού για μεγάλες βάσεις το μέγεθος του value θα ήταν τεράστιο και δεν θα είχαμε καλή παραλληλοποίηση. Η χρήση του buffer μας βοηθάει να ομαδοποιούμε τα δεδομένα ώστε να μειώνουμε τον όγκο των δεδομένων εξόδου και επιπλέον αποφεύγουμε το πρόβλημα της τεράστιας αύξησης του μεγέθους του value. Η είσοδος που θα προέκυπτε με

χρήση buffer θα ήταν:

P1 ?p$$memberOf ?s$$University1
P1 ?p$$takesCourse ?s$$Course4_Course9_Course10
P1 ?p$$type ?s$$GraduateStudent

Η είσοδος αυτή θα μπορούσε να δημιουργηθεί και από ένα S_PO index και μάλιστα χωρίς χρήση buffer. Τα δεδομένα στο S_PO index είναι ήδη ομαδοποιημένα και μπορούμε κατευθείαν από κάθε γραμμή της HBase να δημιουργήσουμε το αντίστοιχο key/value εισόδου.

4.2.3) FullInputJoin

Ο πρώτος αλγόριθμος που θα χρησιμοποιήσουμε για την εκτέλεση των MapReduce join ονομάζεται FullInputJoin. Όπως φαίνεται και από το όνομά του, η εκτέλεση του συγκεκριμένου join γίνεται με πλήρη ανάκτηση των δεδομένων όλων των BGP queries του. Ακολουθούν αναλυτικές περιγραφές των λειτουργιών που επιτελούνται τόσο κατά τη διάρκεια του map όσο και κατά τη διάρκεια του reduce. Οι περιγραφές αυτές θα βασιστούν σε ένα παράδειγμα ώστε να είναι πιο κατανοητές. Έστω, λοιπόν, ότι θέλουμε να εκτελέσουμε το παρακάτω join

ub:GraduateStudent11 ub:takesCourse ?course
?student ub:takesCourse ?course

Τα δεδομένα εισόδου για τα δυο BGP queries μπορούν να ανακτηθούν από τα index SP_O και PO_S αντίστοιχα. Για το παράδειγμά μας τα δεδομένα που βρίσκονται στα index φαίνονται στους παρακάτω πίνακες.

RowId	ColumnId
GraduateStudent11_takesCourse	Course4
GraduateStudent11_takesCourse	Course6
GraduateStudent11_takesCourse	Course7

Πίνακας 11: Δεδομένα SP_O index

RowId	ColumnId
takesCourse_Course4	GraduateStudent11
takesCourse_Course4	GraduateStudent13
takesCourse_Course5	GraduateStudent1
takesCourse_Course5	GraduateStudent5
takesCourse_Course5	GraduateStudent7
takesCourse_Course6	GraduateStudent4
takesCourse_Course6	GraduateStudent8
takesCourse_Course6	GraduateStudent11

takesCourse_Course6	GraduateStudent12
takesCourse_Course7	GraduateStudent3
takesCourse_Course7	GraduateStudent7
takesCourse_Course7	GraduateStudent11

Πίνακας 12: Δεδομένα PO_S index

4.2.3.1) Map

Οι διεργασίες map λαμβάνουν ως είσοδο από τα index όλα τα παρακάτω ζευγάρια key/value και πρέπει να εκτελέσουν το join ως προς τη μεταβλητή course.

P0 ?course$$Course4_Course6_Course7
P1 ?course$$Course4 ?student$$GraduateStudent11_GraduateStudent13
P1 ?course$$Course5 ?student$$GraduateStudent1_GraduateStudent5_GraduateStudent7
P1 ?course$$Course6
?student$$GraduateStudent4_GraduateStudent8_GraduateStudent11_GraduateSt udent12
P1 ?course$$Course7 ?student$$GraduateStudent3_GraduateStudent7_GraduateStudent11

Ο mapper διαβάζει το String που περιέχεται στο value κάθε ζευγαριού και το σπάει μέχρι να εντοπίσει τη μεταβλητή course. Όταν την βρεί παίρνει τη λίστα με τα bindings που την ακολουθούν και για κάθε ένα από αυτά παράγει ένα ζεύγος key/value. Αυτό το ζεύγος περιέχει ως κλειδί την τιμή του κάθε binding και ως value τα bindings όλων των υπόλοιπων μεταβλητών που βρίσκονταν στο value εισόδου. Στο value τοποθετούμε και το id του BGP pattern ή join από το οποίο προέρχονται τα δεδομένα. Η έξοδος, λοιπόν, των mapper για την παραπάνω είσοδο θα ήταν:

Key	**Value**
?course$$Course4	P0
?course$$Course6	P0
?course$$Course7	P0
?course$$Course4	P1 ?student$$GraduateStudent11_GraduateStudent13
?course$$Course5	P1 ?student$$GraduateStudent1_GraduateStudent5_GraduateStudent7
?course$$Course6	P1 ?student$$GraduateStudent4_GraduateStudent8_GraduateStudent11_GraduateSt udent12
?course$$Course7	P1 ?student$$GraduateStudent3_GraduateStudent7_GraduateStudent11

Πίνακας 13: Map output

4.2.3.2) Reduce

Τα ζεύγη key/value που παράγονται από τους mappers ταξινομούνται και ομαδοποιούνται με βάση το κλειδί τους, από το πλαίσιο του MapReduce. Έτσι οι reducers λαμβάνουν ως είσοδο για κάθε κλειδί, μια λίστα με τα values που αντιστοιχούν σε αυτό. Η είσοδος, λοιπόν, των reducers φαίνεται στον παρακάτω πίνακα.

Key	Value
?course$$Course4	P0
	P1 ?student$$GraduateStudent11_GraduateStudent13
?course$$Course5	P1 ?student$$GraduateStudent1_GraduateStudent5_GraduateStudent7
?course$$Course6	P0
	P1 ?student$$GraduateStudent4_GraduateStudent8_GraduateStudent11_GraduateSt udent12
?course$$Course7	P0
	P1 ?student$$GraduateStudent3_GraduateStudent7_GraduateStudent11

Πίνακας 14: Reduce input

Για να εκτελέσουμε το join πρέπει να βρούμε ποιές από τις τιμές της μεταβλητής περιέχονταν και στα δυο BGP queries. Οι reducers, λοιπόν, διαβάζουν τη λίστα με τα values και μετράνε τα διαφορετικά id που περιέχονται σε αυτή. Αν ο αριθμός των διαφορετικών id είναι ίδιος με τον αριθμό των BGP εισόδου τότε δημιουργούν ένα ζευγάρι key/value εξόδου. Σε περίπτωση που ο αριθμός είναι μικρότερος τα συγκεκριμένα δεδομένα αγνοούνται και δεν παράγεται ζεύγος εξόδου. Αν το join χρησιμοποιεί αποτελέσματα ενός άλλου join σαν είσοδο τότε αυτά έχουν το δικό τους id και μετράνε σαν να είχαμε ένα ακόμα BGP ερώτημα.

Όταν οι reducers πρέπει να δημιουργήσουν ένα ζευγάρι εξόδου, το μόνο που πρέπει να κάνουν είναι να ενώσουν τα string που περιέχονται τόσο στο key όσο και σε όλα τα values. Από τα string αφαιρούνται τα id και προσθέτεται ένα καινούργιο id στην αρχή τους. Τα αποτελέσματα μετά από τη φάση του reduce θα είναι:

J0 ?course$$Course4 ?student$$GraduateStudent11_GraduateStudent13
J0 ?course$$Course6
?student$$GraduateStudent4_GraduateStudent8_GraduateStudent11_GraduateSt udent12
J0 ?course$$Course7 ?student$$GraduateStudent3_GraduateStudent7_GraduateStudent11

4.2.4) SingleInputJoin

Ο δεύτερος αλγόριθμος που θα χρησιμοποιήσουμε για την εκτέλεση των join, ονομάζεται SingleInputJoin. Όπως φαίνεται και από το όνομά του, ο αλγόριθμος παίρνει είσοδο μόνο από ένα από τα BGP queris του. Η βασική ιδέα στην οποία βασίζεται είναι το γεγονός ότι κάποιο από τα BGP queries του join μπορεί να παρουσιάζει μεγάλο selectivity, δηλαδή, ο αριθμός των δεδομένων που του αντιστοιχούν είναι μικρός. Μπορούμε λοιπόν να εκμεταλλευτούμε αυτή την ιδιότητα και να χρησιμοποιήσουμε ως είσοδο του join μόνο τα δεδομένα αυτού του query. Διαθέτοντας, λοιπόν, τα bindings της join μεταβλητής από αυτό το query, τα χρησιμοποιούμε για να ελέγξουμε αν ικανοποιούν τα υπόλοιπα BGP queries. Με αυτό τον τρόπο καταφέρνουμε να μειώσουμε το μέγεθος των δεδομένων που επεξεργαζόμαστε και να πετύχουμε καλύτερη απόδοση στην εκτέλεση του join.

Στη συνέχεια, θα αναλύσουμε τον συγκεκριμένο αλγόριθμο εκτέλεσης join μέσω ενός συγκεκριμένου παραδείγματος. Έστω, λοιπόν, ότι θέλουμε να εκτελέσουμε το παρακάτω join

ub:GraduateStudent11 ub:takesCourse ?course
?student ub:takesCourse ?course

Μπορούμε εύκολα να καταλάβουμε ότι το πρώτο BGP query έχει μεγάλο selectivity αφού μας επιστρέφει μόνο τα μαθήματα που παρακολουθεί ένας συγκεκριμένος φοιτητής. Το δεύτερο query, αντίθετα, έχει πολύ χαμηλό selectivity. Τα δεδομένα του είναι πάρα πολλά αφού επιστρέφει για κάθε μάθημα όλους τους φοιτητές που το παρακολουθούν. Μπορούμε, λοιπόν, να χρησιμοποιήσουμε για την εκτέλεσή του το SingleInputJoin.

Τα δεδομένα εισόδου για το πρώτο query μπορούν να ανακτηθούν από το index SP_O. Για το παράδειγμά μας τα δεδομένα που βρίσκονται στα index SP_O και PO_S φαίνονται στους παρακάτω πίνακες.

RowId	ColumnId
GraduateStudent11_takesCourse	Course4
GraduateStudent11_takesCourse	Course6
GraduateStudent11_takesCourse	Course7

Πίνακας 15: Δεδομένα SP_O index

RowId	ColumnId
takesCourse_Course4	GraduateStudent11
takesCourse_Course4	GraduateStudent13
takesCourse_Course5	GraduateStudent1
takesCourse_Course5	GraduateStudent5
takesCourse_Course5	GraduateStudent7
takesCourse_Course6	GraduateStudent4
takesCourse_Course6	GraduateStudent8
takesCourse_Course6	GraduateStudent11
takesCourse_Course6	GraduateStudent12
takesCourse_Course7	GraduateStudent3
takesCourse_Course7	GraduateStudent7
takesCourse_Course7	GraduateStudent11

Πίνακας 16: Δεδομένα PO_S index

4.2.4.1) Map

Οι διεργασίες map λαμβάνουν ως είσοδο μόνο τα δεδομένα που αντιστοιχούν στο πρώτο query, τα οποία βρίσκονται στο index SP_O. Παρακάτω φαίνονται τα ζευγάρια key/value εισόδου για το join ως προς τη μεταβλητή course.

P0 ?course$$Course4_Course6_Course7

Ο mapper διαβάζει το String που περιέχεται στο value του ζευγαριού και το σπάει μέχρι να εντοπίσει τη μεταβλητή course. Όταν την βρεί παίρνει τη λίστα με τα bindings που την ακολουθούν και για κάθε ένα από αυτά παράγει ένα ζεύγος key/value. Αυτό το ζεύγος περιέχει ως κλειδί την τιμή του κάθε binding και ως value τα bindings όλων των υπόλοιπων μεταβλητών που βρίσκονταν στο value εισόδου. Στο value τοποθετούμε και το id του BGP pattern ή join από το οποίο προέρχονται τα δεδομένα. Η έξοδος, λοιπόν, των mapper για την παραπάνω είσοδο θα ήταν:

Key	Value
?course$$Course4	P0
?course$$Course6	P0
?course$$Course7	P0

Πίνακας 17: Map output

4.2.4.2) Reduce

Τα ζεύγη key/value που παράγονται από τους mappers ταξινομούνται και ομαδοποιούνται με βάση το κλειδί τους, από το πλαίσιο του MapReduce. Έτσι οι reducers λαμβάνουν ως είσοδο για κάθε κλειδί, μια λίστα με τα values που αντιστοιχούν σε αυτό. Η είσοδος, λοιπόν, των reducers φαίνεται στον παρακάτω πίνακα.

Key	Value
?course$$Course4	P0
?course$$Course6	P0
?course$$Course7	P0

Πίνακας 18: Reduce input

Για να εκτελέσουμε το join πρέπει να βρούμε ποιό από τα bindings της μεταβλητής course ικανοποιεί και το δεύτερο BGP query. Για κάθε key, λοιπόν, οι reducers ψάχνουν στο index PO_S χρησιμοποιώντας το συγκεκριμένο κλειδί ως object. Αν υπάρχει αντίστοιχη γραμμή στον πίνακα PO_S τότε παράγουν ένα ζευγάρι key/value εξόδου. Σε περίπτωση που το index δεν διαθέτει τέτοια εγγραφή, οι reducers δεν δημιουργούν καμία έξοδο.

Για να δημιουργήσουν, οι reducers, ένα ζευγάρι εξόδου, αρχικά διαβάζουν τα δεδομένα που βρίσκονται στο index και από αυτά δημιουργούν τα αντίστοιχα binding για τη μεταβλητή student. Στη συνέχεια ενώνουν τα bindings που δημιουργήθηκαν από τα index με αυτά που υπήρχαν στο value του ζευγαριού εισόδου τους. Από το string που δημιουργείται αφαιρούνται τα id και προσθέτεται ένα καινούργιο id στην αρχή τους. Τα αποτελέσματα μετά από τη φάση του reduce θα είναι:

J0 ?course$$Course4 ?student$$GraduateStudent11_GraduateStudent13
J0 ?course$$Course6
?student$$GraduateStudent4_GraduateStudent8_GraduateStudent11_GraduateS tudent12
J0 ?course$$Course7 ?student$$GraduateStudent3_GraduateStudent7_GraduateStudent11

4.2.5) Σύγκριση των αλγορίθμων εκτέλεσης των join

Καθένας από τους παραπάνω αλγορίθμους παρουσιάζει τόσο πλεονεκτήματα όσο και μειονεκτήματα. Όσον αφορά τον αλγόριθμο του SingleInputJoin αυτός έχει το πλεονέκτημα ότι μειώνει πάρα πολύ το μέγεθος των δεδομένων που χρειάζεται να επεξεργαστούμε. Όμως αυτό δεν ισχύει πάντοτε. Όπως αναφέραμε για να είναι αποδοτικός αυτός ο αλγόριθμος θα πρέπει να υπάρχει κάποιο BGP query με πολύ υψηλό selectivity. Σε περιπτώσεις, όμως, που δεν υπάρχει κάποιο τέτοιο query η απόδοση του συγκεκριμένου αλγορίθμου πέφτει δραματικά. Η μείωση της απόδοσης γίνεται ακόμα πιο αισθητή σε περιπτώσεις που υπάρχει query με υψηλό selectivity αλλά εμείς επιλέξουμε κάποιο άλλο ως είσοδο.

Αντίθετα, το FullInputJoin λαμβάνει πάντα ως είσοδο τα δεδομένα όλων των query και έτσι παρουσιάζει μεγαλύτερη ευστάθεια στα διαφορετικά είδη των join.

Πιο συγκεκριμένα θα εξετάσουμε αυτές τις διαφορετικές περιπτώσεις με αντίστοιχα παραδείγματα. Στην πρώτη περίπτωση υπάρχει ένα query με υψηλό selectivity το οποίο χρησιμοποιούμε και ως είσοδο για το SingleInputJoin. Στον παρακάτω πίνακα φαίνεται ο αριθμός των bindings που αντιστοιχούν στο κάθε BGP query. Βλέπουμε, δηλαδή, ότι στο πρώτο query αντιστοιχούν 5 διαφορετικά bindings για τη μεταβλητή course. Στο δεύτερο query αντιστοιχούν 500 διαφορετικά course κάθε ένα από τα οποία έχει κατά μέσο όρο 30 μαθητές.

BGP query	?course	?student
ub:GraduateStudent11 ub:takesCourse ?course	5	
?student ub:takesCourse ?course	500	30

Πίνακας 19: Μέγεθος των BGP

Θα εξετάσουμε αναλυτικά την απόδοση των δυο αλγορίθμων για την εκτέλεση του join.

FullInputJoin: Στο FullInputJoin θα διαβάσουμε όλα τα δεδομένα από τα index στην αρχή του join. Άρα θα χρειαστούμε συνολικό χρόνο για ανάκτηση των δεδομένων:

$$index + 5*read + index +500*30read= 2*index +15005*read$$

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Τα δεδομένα εισόδου βρίσκονται διαδοχικά αποθηκευμένα στο index και άρα χρειάζονται συνολικά μόνο δυο αναζητήσεις στο index.

SingleInputJoin: Αν χρησιμοποιήσουμε ως είσοδο το πρώτο query θα χρειαστούμε συνολικό χρόνο

map: index + 5*read
reduce: 5*(index+30*read)
συνολικά: 6*index + 155*read

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Στη φάση του map input διαβάζουμε μόνο τα δεδομένα του πρώτου query. Στη φάση του reduce για κάθε binding διαβάζουμε μία γραμμή του index. Για να διαβάσουμε αυτή τη

γραμμή χρειαζόμαστε μια αναζήτηση στο index και 30 αναγνώσεις από τον πίνακα. Βλέπουμε, λοιπόν, ότι η χρήση του SingleInputJoin πραγματοποιεί πολύ λιγότερες ανακτήσεις δεδομένων και άρα έχει καλύτερη απόδοση από το FullInputJoin.

Στη δεύτερη περίπτωση έχουμε τα δυο BGP query έχουν εξίσου μεγάλο selectivity. Στον παρακάτω πίνακα φαίνεται ο αριθμός των bindings που αντιστοιχούν στο κάθε BGP query. Βλέπουμε, δηλαδή, ότι στο πρώτο query αντιστοιχούν 500 διαφορετικά bindings για τη μεταβλητή course. Στο δεύτερο query αντιστοιχούν 500 διαφορετικά course κάθε ένα από τα οποία έχει κατά μέσο όρο 30 μαθητές.

BGP query	?course	?student
ub:GraduateStudent11 ub:takesCourse ?course	500	
?student ub:takesCourse ?course	500	30

Πίνακας 20: Μέγεθος των BGP

Θα εξετάσουμε αναλυτικά την απόδοση των δυο αλγορίθμων για την εκτέλεση του join.

FullInputJoin: Στο FullInputJoin θα διαβάσουμε όλα τα δεδομένα από τα index στην αρχή του join. Άρα θα χρειαστούμε συνολικό χρόνο για ανάκτηση των δεδομένων:

index + 500*read + index +500*30read= 2*index +15500*read

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Τα δεδομένα εισόδου βρίσκονται διαδοχικά αποθηκευμένα στο index και άρα χρειάζονται συνολικά μόνο δυο αναζητήσεις στο index.

SingleInputJoin: Αν χρησιμοποιήσουμε ως είσοδο το πρώτο query θα χρειαστούμε συνολικό χρόνο

map: index + 500*read
reduce: 500*(index+30*read)
συνολικά: 501*index + 15500*read

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Στη φάση του map input διαβάζουμε μόνο τα δεδομένα του πρώτου query. Στη φάση του reduce για κάθε binding διαβάζουμε μια γραμμή του index. Για να διαβάσουμε αυτή τη γραμμή χρειαζόμαστε μια αναζήτηση στο index και 30 αναγνώσεις από τον πίνακα.

Βλέπουμε, λοιπόν, ότι σε αυτή την περίπτωση το SingleInputJoin πραγματοποιεί 499 περισσότερες αναζητήσεις στο index. Αυτό έχει ως αποτέλεσμα να παρουσιάζει χειρότερη απόδοση από το FullInputJoin. Μάλιστα, οι αναζητήσεις αυτές στο index γίνονται ταυτόχρονα κατά τη φάση του reduce με αποτέλεσμα να υπερφορτώνουν τους servers της HBase και να οδηγούν σε ακόμα μικρότερη απόδοση. Τέτοιες μαζικές προσβάσεις στην HBase όχι μόνο σειριοποιούνται αλλά μπορεί να οδηγήσουν ακόμα και σε πτώση των server της.

Η τρίτη περίπτωση αποτελεί την χειρότερη περίπτωση για το SingleInputJoin αφού σε αυτή κάνουμε λάθος επιλογή του BGP query εισόδου. Στον παρακάτω πίνακα φαίνεται ο αριθμός των bindings που αντιστοιχούν στο κάθε BGP query. Βλέπουμε, δηλαδή, ότι στο πρώτο query αντιστοιχούν 500 διαφορετικά bindings για τη μεταβλητή course. Στο δεύτερο query αντιστοιχούν 5 διαφορετικά course κάθε ένα από τα οποία έχει κατά μέσο όρο 30

μαθητές.

BGP query	?course	?student
ub:GraduateStudent11 ub:takesCourse ?course	500	
?student ub:takesCourse ?course	5	30

Πίνακας 21: Μέγεθος των BGP

Θα εξετάσουμε αναλυτικά την απόδοση των δυο αλγορίθμων για την εκτέλεση του join.

FullInputJoin: Η απόδοση του FullInputJoin, όπως έχουμε εξηγήσει, παραμένει σταθερή και δεν εξαρτάται από την επιλογή του query εισόδου. Άρα και σε αυτή την περίπτωση θα διαβάσουμε όλα τα δεδομένα από τα index στην αρχή του join. Δηλαδή, θα χρειαστούμε συνολικό χρόνο για ανάκτηση των δεδομένων:

$$index + 500*read + index + 5*30read = 2*index + 650*read$$

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Τα δεδομένα εισόδου βρίσκονται διαδοχικά αποθηκευμένα στο index και άρα χρειάζονται συνολικά μόνο δυο αναζητήσεις στο index.

SingleInputJoin: Αν χρησιμοποιήσουμε, λανθασμένα, ως είσοδο το πρώτο query θα χρειαστούμε συνολικό χρόνο

map: index + 500*read
reduce: 500*(index+30*read)
συνολικά: 501*index + 15500*read

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Στη φάση του map input διαβάζουμε μόνο τα δεδομένα του πρώτου query. Στη φάση του reduce για κάθε binding διαβάζουμε μια γραμμή του index. Για να διαβάσουμε αυτή τη γραμμή χρειαζόμαστε μια αναζήτηση στο index και 30 αναγνώσεις από τον πίνακα.

Βλέπουμε, λοιπόν, ότι σε αυτή την περίπτωση το SingleInputJoin πραγματοποιεί 499 περισσότερες αναζητήσεις στο index και 14850 περισσότερες αναγνώσεις από τους πίνακες της HBase. Όπως μπορούμε εύκολα να καταλάβουμε, η απόδοση του SingleInputJoin είναι δραματική. Δεν έχουμε μόνο περισσότερες προσβάσεις στο index της HBase αλλά έχουμε και πολλαπλές αναγνώσεις των ίδιων δεδομένων από τον πίνακα. Και εδώ, οι αναζητήσεις αυτές στο index γίνονται ταυτόχρονα κατά τη φάση του reduce με αποτέλεσμα να υπερφορτώνουν τους servers της HBase και να οδηγούν σε ακόμα μικρότερη απόδοση και πιθανή πτώση των server της.

Συμπεραίνουμε, λοιπόν, ότι οι δυο αλγόριθμοι που παρουσιάσαμε είναι συμπληρωματική και πρέπει να χρησιμοποιήσουμε ένα συνδυασμό και τον δυο ώστε το σύστημά μας να έχει καλή απόδοση σε όλα τα είδη των ερωτημάτων. Γι' αυτό το σκοπό δημιουργήσαμε το PartialInputJoin που αποτελεί έναν συνδυασμό των δυο αλγορίθμων. Στην παρακάτω ενότητα θα αναλύσουμε το PartialInputJoin και θα δείξουμε τον τρόπο με τον οποίο συνδυάζει τις ιδιότητες και των δυο παραπάνω αλγορίθμων.

4.2.6) PartialInputJoin

Ο αλγόριθμος αυτός συνδυάζει τα πλεονεκτήματα των αλγορίθμων FullInputJoin και SingleInputJoin. Όπως φαίνεται και από το όνομά του μπορούμε να επιλέξουμε ως είσοδό του έναν μεταβλητό αριθμό από τα query εισόδου. Αν επιλέξουμε ένα query ο αλγόριθμος ισοδυναμεί με τον SingleInputJoin, ενώ αν επιλέξουμε όλα τα query σαν input ο αλγόριθμος είναι ίδιος με τον FullInputJoin. Μια επιπλέον δυνατότητα που προσφέρεται από αυτόν είναι η δυνατότητα επιλογής οποιουδήποτε αριθμού input queries. Αυτό μας επιτρέπει να έχουμε την καλύτερη δυνατή απόδοση σε όλους τους τύπους των διαφορετικών join. Βέβαια, η απόδοση του αλγορίθμου εξαρτάται σε μεγάλο βαθμό από την σωστή ή όχι επιλογή των ερωτημάτων εισόδου.

Στη συνέχεια, θα αναλύσουμε τον συγκεκριμένο αλγόριθμο εκτέλεσης join μέσω ενός συγκεκριμένου παραδείγματος. Έστω, λοιπόν, ότι θέλουμε να εκτελέσουμε το παρακάτω join

?student ub:takesCourse ub:Course10
?student ub:takesCourse ub:Course11
?student rdf:type ub:Student

Μπορούμε εύκολα να καταλάβουμε ότι τα δυο πρώτα BGP queries έχουν και τα δυο μεγάλο selectivity. Όμως ο συνδυασμός των δυο αυτών ερωτημάτων έχει ακόμα μεγαλύτερο selectivity. Το τρίτο query δεν παρουσιάζει σχεδόν καθόλου επιλεκτικότητα αφού επιστρέφει όλους τους φοιτητές που βρίσκονται αποθηκευμένοι στη βάση μας. Θα επιλέξουμε λοιπόν ως είσοδο του PartialInputJoin τα δυο πρώτα ερωτήματα.

Τα δεδομένα εισόδου για τα δυο πρώτα query μπορούν να ανακτηθούν από το index PO_S Για το παράδειγμά μας τα δεδομένα που βρίσκονται στο index PO_S φαίνονται στον παρακάτω πίνακα.

RowId	ColumnId
takesCourse_Course10	GraduateStudent1
takesCourse_Course10	GraduateStudent5
takesCourse_Course10	GraduateStudent7
takesCourse_Course10	GraduateStudent11
takesCourse_Course10	GraduateStudent13
takesCourse_Course11	GraduateStudent3
takesCourse_Course11	GraduateStudent4
takesCourse_Course11	GraduateStudent5
takesCourse_Course11	GraduateStudent7
takesCourse_Course11	GraduateStudent9
takesCourse_Course11	GraduateStudent10
takesCourse_Course11	GraduateStudent15

Πίνακας 22: PO_S index

4.2.6.1) Map

Οι διεργασίες map λαμβάνουν ως είσοδο μόνο τα δεδομένα που αντιστοιχούν στα δυο πρώτα query, τα οποία βρίσκονται στο index PO_S. Παρακάτω φαίνονται τα ζευγάρια key/value εισόδου για το join ως προς τη μεταβλητή student.

P0 ?student$$GraduateStudent1_GraduateStudent5_GraduateStudent7_GraduateStudent11_GraduateStudent13
P1 ?student$$GraduateStudent3_GraduateStudent4_GraduateStudent5_GraduateStudent7_GraduateStudent9_ GraduateStudent10_ GraduateStudent15

Ο mapper διαβάζει το String που περιέχεται στο value του ζευγαριού και το σπάει μέχρι να εντοπίσει τη μεταβλητή student. Όταν την βρεί παίρνει τη λίστα με τα bindings που την ακολουθούν και για κάθε ένα από αυτά παράγει ένα ζεύγος key/value. Αυτό το ζεύγος περιέχει ως κλειδί την τιμή του κάθε binding και ως value τα bindings όλων των υπόλοιπων μεταβλητών που βρίσκονταν στο value εισόδου. Στο value τοποθετούμε και το id του BGP pattern ή join από το οποίο προέρχονται τα δεδομένα. Η έξοδος, λοιπόν, των mapper για την παραπάνω είσοδο θα ήταν:

Key	Value
?student$$GraduateStudent1	P0
?student$$GraduateStudent5	P0
?student$$GraduateStudent7	P0
?student$$GraduateStudent11	P0
?student$$GraduateStudent13	P0
?student$$GraduateStudent3	P1
?student$$GraduateStudent4	P1
?student$$GraduateStudent5	P1
?student$$GraduateStudent7	P1
?student$$GraduateStudent9	P1
?student$$GraduateStudent10	P1
?student$$GraduateStudent15	P1

Πίνακας 23: Map output

4.2.6.2) Reduce

Τα ζεύγη key/value που παράγονται από τους mappers ταξινομούνται και ομαδοποιούνται με βάση το κλειδί τους, από το πλαίσιο του MapReduce. Έτσι οι reducers λαμβάνουν ως είσοδο για κάθε κλειδί, μια λίστα με τα values που αντιστοιχούν σε αυτό. Η είσοδος, λοιπόν, των reducers φαίνεται στον παρακάτω πίνακα.

Key	Value
?student$$GraduateStudent1	P0

?student$$GraduateStudent3	P1
?student$$GraduateStudent4	P1
?student$$GraduateStudent5	P0
	P1
?student$$GraduateStudent7	P0
	P1
?student$$GraduateStudent9	P1
?student$$GraduateStudent10	P1
?student$$GraduateStudent11	P0
?student$$GraduateStudent13	P0
?student$$GraduateStudent15	P1

Πίνακας 24: Reduce input

Για να εκτελέσουμε το join πρέπει να βρούμε ποιό από τα bindings της μεταβλητής student ικανοποιεί όλα τα BGP query. Για κάθε key, λοιπόν, οι reducers αρχικά διαβάζουν τη λίστα με τα values και μετράνε τα διαφορετικά id που περιέχονται σε αυτή. Αν ο αριθμός των διαφορετικών id είναι ίδιος με τον αριθμό των BGP εισόδου, δηλαδή 2, τότε ο reducer προχωράει στη φάση της αναζήτησης στην HBase. Σε περίπτωση που ο αριθμός είναι μικρότερος τα συγκεκριμένα δεδομένα αγνοούνται και δεν παράγεται ζεύγος εξόδου. Από την πρώτη αυτή φάση διατηρούνται τα παρακάτω δεδομένα.

Key	**Value**
?student$$GraduateStudent5	P0
	P1
?student$$GraduateStudent7	P0
	P1

Πίνακας 25: Ενδιάμεσα δεδομένα reducer

Στη δεύτερη φάση πρέπει να βρούμε ποιό από αυτά τα bindings της μεταβλητής student ικανοποιεί και το τρίτο BGP query. Για κάθε key, λοιπόν, οι reducers ψάχνουν στο index PO_S χρησιμοποιώντας το συγκεκριμένο κλειδί ως subject. Αν υπάρχει αντίστοιχη γραμμή στον πίνακα PO_S τότε παράγουν ένα ζευγάρι key/value εξόδου. Σε περίπτωση που το index δεν διαθέτει τέτοια εγγραφή, οι reducers δεν δημιουργούν καμία έξοδο.

Για να δημιουργήσουν, οι reducers, ένα ζευγάρι εξόδου, αρχικά διαβάζουν τα δεδομένα που βρίσκονται στο index και από αυτά δημιουργούν τα αντίστοιχα bindings. Στη συνέχεια ενώνουν τα bindings που δημιουργήθηκαν από τα index με αυτά που υπήρχαν σε όλα τα value του συγκεκριμένου κλειδιού. Από το string που δημιουργείται αφαιρούνται τα id και προσθέτεται ένα καινούργιο id στην αρχή τους. Τα αποτελέσματα μετά από τη φάση του reduce θα είναι:

J0 ?student$$GraduateStudent5
J0 ?student$$GraduateStudent7

4.2.6.3) Απόδοση του PartialInputJoin

Σε αυτή την ενότητα θα εξετάσουμε την απόδοση του PartialInputJoin χρησιμοποιώντας ως παράδειγμα το join που εξετάστηκε παραπάνω. Στον παρακάτω πίνακα φαίνεται ο αριθμός των bindings που αντιστοιχούν στο κάθε BGP query. Βλέπουμε, δηλαδή, ότι στο πρώτο query αντιστοιχούν 40 διαφορετικά bindings για τη μεταβλητή student, ενώ για το δεύτερο αντιστοιχούν 35 bindings. Στο τρίτο query αντιστοιχούν 1000 διαφορετικά student.

BGP query	?student
?student ub:takesCourse ub:Course10	40
?student ub:takesCourse ub:Course11	35
?student rdf:type ub:Student	1000

Πίνακας 26: Μέγεθος BGP

Θα εξετάσουμε αναλυτικά την απόδοση των δυο αρχικών αλγορίθμων και θα την συγκρίνουμε με αυτή του PartialInputJoin.
FullInputJoin: Στο FullInputJoin θα διαβάσουμε όλα τα δεδομένα από τα index στην αρχή του join. Άρα θα χρειαστούμε συνολικό χρόνο για ανάκτηση των δεδομένων:

index + 40*read + index +35*read + index +1000*read= 3*index +1075*read

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Τα δεδομένα εισόδου βρίσκονται διαδοχικά αποθηκευμένα στο index και άρα χρειάζονται συνολικά μόνο τρεις αναζητήσεις στο index.

SingleInputJoin: Αν χρησιμοποιήσουμε ως είσοδο το δεύτερο query, που έχει και το μεγαλύτερο selectivity, θα χρειαστούμε συνολικό χρόνο

map: index + 35*read
reduce: 35*2*(index+read)
συνολικά: 71*index + 105*read

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Στη φάση του map input διαβάζουμε μόνο τα δεδομένα του δεύτερου query. Στη φάση του reduce για κάθε binding ελέγχουμε την ύπαρξη ενός cell του index για κάθε επιπλέον query, δηλαδή, ελέγχουμε την ύπαρξη 2 cell. Για να ελέγξουμε την ύπαρξη ενός cell αρκεί μια αναζήτηση στο index και μια ανάγνωση από τον πίνακα.

PartialInputJoin: Σε αυτή την περίπτωση θα χρησιμοποιήσουμε ως είσοδο τα δυο πρώτα query. Ο συνδυασμός των δυο αυτών ερωτημάτων μας μειώνει πάρα πολύ το μέγεθος των δεδομένων που πρέπει να επεξεργαστούμε. Για τον υπολογισμό θεωρούμε ότι υπάρχουν 5 φοιτητές που παρακολουθούν και τα δυο μαθήματα. Για να εκτελέσουμε αυτό το join θα χρειαστούμε συνολικό χρόνο

map: index + 40*read + index + 35*read
reduce: 5*(index+read)

συνολικά: 6*index + 80*read

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Στη φάση του map input διαβάζουμε τα δεδομένα του πρώτου και του δεύτερου query. Στη φάση του reduce για κάθε binding ελέγχουμε αρχικά αν ικανοποιεί και τα δυο query εισόδου. Τα binding που ικανοποιούν τα query αυτά είναι όπως θεωρήσαμε 5. Στη συνέχεια για κάθε ένα από αυτά τα binding ελέγχουμε αν υπάρχει το αντίστοιχο cell στον πίνακα με τα index. Για να ελέγξουμε την ύπαρξη ενός cell αρκεί μια αναζήτηση στο index και μια ανάγνωση από τον πίνακα. Βλέπουμε, λοιπόν, ότι με χρήση του PartialInputJoin πετυχαίνουμε την καλύτερη δυνατή απόδοση. Στα πλαίσια που οι επιλογές εισόδου του συστήματός μας είναι οι βέλτιστες μπορούμε με το PartialInputJoin να πετύχουμε και την καλύτερη δυνατή απόδοση.

4.2.6.4) Πρόσθετες δυνατότητες των αλγορίθμων εκτέλεσης των join

Σε αυτή την ενότητα θα αναλύσουμε την βέλτιστη δυνατή αξιοποίηση ενός MapReduce job για την εκτέλεση των join αλγορίθμων που παρουσιάστηκαν. Αρχικά πρέπει να αναφέρουμε ότι για την εκτέλεση ενός MapReduce job χρειάζεται αρκετός χρόνος για την αρχικοποίηση του, τον διαμοιρασμό των δεδομένων εισόδου του καθώς και για τον τερματισμό του. Αυτός ο χρόνος είναι σημαντικός και πρέπει να λαμβάνεται σοβαρά υπόψη. Για αυτό θα πρέπει σε κάθε MapReduce job να εκτελούμε όσο το δυνατόν περισσότερες εργασίες.

Έστω λοιπόν ότι θέλουμε να εκτελέσουμε το παρακάτω SPARQL ερώτημα:

```
SELECT  ?x ?y ?z
WHERE   {
        ?x rdf:type ub:GraduateStudent .
        ?y rdf:type ub:University .
        ?z rdf:type ub:Department .
        ?x ub:memberOf ?z .
        ?z ub:subOrganizationOf ?y .
        ?x ub:undergraduateDegreeFrom ?y
}
```

Το συγκεκριμένο query είναι αρκετά πολύπλοκο και απαιτεί την εκτέλεση πολλαπλών join για την απάντησή του. Θα χρειαστούμε, λοιπόν, περισσότερα από ένα MapReduce jobs. Για να εξοικονομήσουμε το χρόνο που χάνεται στην αρχικοποίηση και στον τερματισμό αυτών των εργασιών θα πρέπει σε κάθε μια από αυτές να εκτελούμε όσο το δυνατόν περισσότερα join.

Μια αρχική παρατήρηση είναι το γεγονός ότι μπορούμε με ένα MapReduce job να κάνουμε join όλα τα BGP που περιέχουν μια συγκεκριμένη μεταβλητή. Αν θέλουμε, δηλαδή, να εκτελέσουμε το join ως προς τη μεταβλητή x μπορούμε να συνδυάσουμε και τα τρία BGP στα οποία εμφανίζεται:

?x rdf:type ub:GraduateStudent
?x ub:memberOf ?z
?x ub:undergraduateDegreeFrom ?y

Με μια δεύτερη παρατήρηση διαπιστώνουμε ότι στο ίδιο MapReduce job μπορούμε να εκτελέσουμε και ένα join ως προς κάποια άλλη μεταβλητή. Αυτό μπορεί να γίνει εύκολα αφού το μόνο που χρειάζεται είναι ο mapper να ξέρει για κάθε key/value του ποιά είναι η μεταβλητή σύμφωνα με την οποία γίνεται το join. Τα key/value εισόδου των mappers διαθέτουν και το id του query από το οποίο προέρχονται. Ο mapper, λοιπόν, αρκεί να διαθέτει μια δομή η οποία αποθηκεύει ποιά μεταβλητή χρησιμοποιείται γίνεται join σε κάθε query. Η υπόλοιπη διαδικασία του MapReduce join δεν χρειάζεται καμία αλλαγή. Το μόνο που πρέπει να προσέξουμε είναι το γεγονός ότι κάθε query μπορεί να χρησιμοποιείται μόνο μια φορά σε ένα MapReduce job. Στο παραπάνω, λοιπόν, παράδειγμα θα μπορούσαμε να εκτελέσουμε στο ίδιο MapReduce job και ένα join ως προς τη μεταβλητή ?z. Τα queris που έχουν την μεταβλητή ?z και δεν έχουν επιλεχθεί ήδη είναι το τρίτο και το πέμπτο. Άρα μπορούμε να εκτελέσουμε ταυτόχρονα τα παρακάτω join

?x rdf:type ub:GraduateStudent
?x ub:memberOf ?z
?x ub:undergraduateDegreeFrom ?y
?z rdf:type ub:Department
?z ub:subOrganizationOf ?y

Βλέπουμε, λοιπόν, ότι σε ένα MapReduce job μπορούμε να εκτελέσουμε ταυτόχρονα πολλά διαφορετικά join. Παρουσιάζεται, έτσι, η ανάγκη προσδιορισμού ενός αλγορίθμου που θα έχει την ικανότητα να αναθέτει αποδοτικά τις εργασίες join στα MapReduce jobs. Ο αλγόριθμος αυτός θα λαμβάνει υπόψη του όλα τα παραπάνω και θα φροντίζει να γίνεται βάλτιστη αξιοποίηση των πόρων του MapReduce. Στην παρακάτω ενότητα θα περιγράψουμε

4.2.7) Join Planner

4.2.7.1) Εισαγωγή

Το βασικότερο κομμάτι ενός συστήματος που επεξεργάζεται SPARQL ερωτήματα είναι ο αλγόριθμος με τον οποίο επιλέγει τη σειρά εκτέλεσης των απαραίτητων join. Σε αυτή την ενότητα, λοιπόν, θα παρουσιάσουμε τον αλγόριθμο που θα προσδιορίζει το βέλτιστο πλάνο εκτέλεσης των MapReduce join στο σύστημά μας. Αυτός ο αλγόριθμος πρέπει να λαμβάνει υπόψη του τόσο το selectivity κάθε input query, όσο και την δυνατότητα ελαχιστοποίησης του συνολικού αριθμού των MapReduce jobs. Όπως αναφέραμε και στην προηγούμενη ενότητα, για την εκτέλεση κάθε ξεχωριστού MapReduce job ξοδεύουμε αρκετό χρόνο στην αρχικοποίηση του συστήματος και στο διαμοιρασμό των δεδομένων. Ο χρόνος αυτός είναι υπολογίσιμος σε σχέση με το συνολικό χρόνο διεξαγωγής του join, άρα θα πρέπει πάντα να προσπαθούμε να ελαχιστοποιήσουμε τον αριθμό των MapReduce εργασιών που χρειαζόμαστε για την ολοκλήρωση του SPARQL ερωτήματος.

Στη συνέχεια της ενότητας θα παρουσιάσουμε ένα μοντέλο προσδιορισμού του κόστους κάθε join καθώς και τον αλγόριθμο που θα βασίζεται σε αυτό και θα επιλέγει κάθε φορά το βέλτιστο πλάνο εκτέλεσης των join. Ο αλγόριθμος που θα χρησιμοποιήσουμε βασίζεται στον αλγόριθμο που χρησιμοποιείται στην εργασία [16].

4.2.7.2) Μοντέλο κόστους εκτέλεσης join

Για να απαντήσουμε ένα ερώτημα SPARQL μπορεί να χρειαστούμε περισσότερα

από ένα MapReduce job. Συνεπώς, η εκτίμηση του κόστους για την επεξεργασία ενός ερωτήματος, απαιτεί ξεχωριστή εκτίμηση κόστους για κάθε επιμέρους εργασία MapReduce. Το κόστος εκτέλεσης μιας εργασίας MapReduce παρουσιάζεται συνοπτικά στην παρακάτω εξίσωση.

Job = IJ + MI_no*(mread + map) + MO_no*write +Sort+ RI_no*(rread + reduce) +RO_no*write + FJ

όπου
IJ: Initialize job, δηλδή, ο χρόνος που χρειάζεται για την αρχικοποίηση του συστήματος.
MI_no: Map Input number, δηλαδή, ο αριθμός των ζευγαριών key/value που δίνονται ως είσοδος στη map φάση.
mread: Ο χρόνος ανάκτησης ενός ζευγαριού key/value από τους πίνακες HBase.
map: Ο χρόνος επεξεργασίας ενός ζευγαριού key/value μέσω της συνάρτησης map.
MO_no: Map Output number, δηλαδή, ο αριθμός των ζευγαριών key/value που παράγονται από τη φάση του map.
Write: Ο χρόνος που απαιτείται για την εγγραφή ενός ζευγαριού key/value στα αρχεία εξόδου του HDFS.
Sort: Ο χρόνος που χρειάζεται για την ταξινόμηση των ζευγαριών key/value που δημιουργήθηκαν από τη map φάση.
RI_no: Reduce Input number, δηλαδή, ο αριθμός των διαφορετικών κλειδιών που λαμβάνουν οι reducers. Ο αριθμός αυτός διαφέρει από τον αριθμό των ζευγαριών εξόδου της map φάσης, αφού τα ζευγάρια αυτά ομαδοποιούνται για κάθε διαφορετική τιμή του κλειδιού.
rread: Ο χρόνος ανάγνωσης ενός ζευγαριού από τα ενδιάμεσα αρχεία που δημιουργούνται μετά τη φάση του map.
Reduce: Ο χρόνος επεξεργασίας για ένα συγκεκριμένο κλειδί μέσω της συνάρτησης reduce.
RO_no: Reduce Output number, δηλαδή, ο αριθμός των ζευγαριών εξόδου της φάσης του reduce.
FJ: Finalize job, δηλαδή ο χρόνος που χρειάζεται για τον τερματισμό της εργασίας MapReduce.

Το παραπάνω μοντέλο κόστους είναι ιδιαίτερα αναλυτικό και περιλαμβάνει όλες τις βασικές παραμέτρους που επηρεάζουν το χρόνο εκτέλεσης μιας εργασίας MapReduce. Όμως, δεν είμαστε πάντα σε θέση να γνωρίζουμε όλες τις παραπάνω παραμέτρους κατά τη διάρκεια προσδιορισμού του βέλτιστου πλάνου εκτέλεσης. Γι' αυτό το λόγο θα πρέπει να απλοποιήσουμε το μοντέλο μας και να βασιστούμε περισσότερο σε παραμέτρους οι οποίες μπορούν εύκολα να προσδιοριστούν.

Όπως αναφέραμε σε προηγούμενη ενότητα, οι εργασίες MapReduce θα εκτελούν τον αλγόριθμο του PartialInputJoin. Πρέπει λοιπόν αρχικά να μελετήσουμε την πολυπλοκότητα αυτού του αλγορίθμου για όλους τους δυνατούς συνδυασμούς εισόδου του. Βασική παράμετρος του αλγορίθμου αυτού είναι η επιλογή του αριθμού των BGP query εισόδου. Στον παρακάτω πίνακα φαίνεται ο αριθμός των bindings που αντιστοιχούν στο κάθε BGP query, για ένα παράδειγμα εκτέλεσης του αλγορίθμου PartialInputJoin. Βλέπουμε, δηλαδή, ότι στο πρώτο query αντιστοιχούν n1 διαφορετικά bindings για τη μεταβλητή στην οποία γίνεται το join, ενώ το query αυτό δεν διαθέτει bindings για κάποια άλλη μεταβλητή. Τα bindings που ονομάζουμε other_variables αντιστοιχούν στις υπόλοιπες μεταβλητές που έχει κάθε query. Η τιμή o, που περιέχεται στον πίνακα, είναι η μέση τιμή των bindings των υπολοίπων μεταβλητών, που αντιστοιχεί σε ένα binding για τη join μεταβλητή. Για παράδειγμα, αν για κάθε binding της μεταβλητής student αντιστοιχούν κατά

μέσο όρο 10 μαθήματα, τα οποία παρακολουθεί, τότε η τιμή του ο, για αυτό το query, θα είναι 10. Αντίστοιχα, παρουσιάζονται και τα δεδομένα εισόδου για τα υπόλοιπα queries του join. Τέλος, ως είσοδος για τον αλγόριθμο επιλέγονται τα δυο πρώτα queries.

BGP query	join_variable	other_variables
BGP1	n1	_
BGP2	n2	o2
BGP3	n3	o3
BGP4	n4	_

Πίνακας 27: BGP variables

Ο χρόνος που θα χρειαστεί για να εκτελεστεί το συγκεκριμένο join είναι:

map: index + n1*read + index + (n2*o2)*read
reduce: $(n1 \cap n2)$*(index+o3*read +index +read)

όπου index: ο χρόνος αναζήτησης μιας συγκεκριμένης γραμμής μέσω του index της HBase
read: ο χρόνος για να διαβάσουμε μια εγγραφή από τον πίνακα
Στη φάση του map input διαβάζουμε τα δεδομένα του πρώτου και του δεύτερου query. Στη φάση του reduce για κάθε binding ελέγχουμε αρχικά αν ικανοποιεί και τα δυο query εισόδου. Τα binding που ικανοποιούν και τα δυο query αυτά είναι ο αριθμός $(n1 \cap n2)$. Στη συνέχεια για κάθε ένα από αυτά τα binding ελέγχουμε αν υπάρχει η αντίστοιχη εγγραφή στον πίνακα με τα index για τα υπόλοιπα 2 queries. Στο παραπάνω μοντέλο δεν λάβαμε υπόψη μας τους χρόνους αρχικοποίησης και τερματισμού της εργασίας MapReduce αλλά επικεντρωθήκαμε μόνο στον χρόνο που εξαρτάται από τα πραγματικά δεδομένα εισόδου και την επεξεργασία τους. Για λόγους ευκολίας μπορούμε να απλοποιήσουμε το παραπάνω μοντέλο γράφοντας την απόδοση ως τάξη μεγέθους των μεταβλητών n και o:

map: O(n1 + n2*o2)
reduce: O($(n1 \cap n2)$* (1 + o3))

Το μέγεθος $(n1 \cap n2)$δεν μπορεί να υπολογιστεί άμεσα πριν την εκτέλεση του join, οπότε, για τον προσδιορισμό του θα χρησιμοποιήσουμε το άνω φράγμα του n1+n2. Αν στις παραπάνω σχέσεις θέσουμε και o1=o4=1 τότε προκύπτει ο γενικός τύπος απόδοσης

map: O(n1*o1 + n2*o2)
reduce: O((n1+n2)* (o3 + o4))

Μπορούμε, λοιπόν, να χρησιμοποιήσουμε το παρακάτω μοντέλο απόδοσης για το PartialInputJoin:

$$Join = O\left(\sum_{i \in Inp} n_i o_i\right) + O\left(\sum_{i \in Inp} n_i * \sum_{j \notin Inp} o_j\right)$$

Όπου Inp το σύνολο των αρχείων εισόδου του join.

Ο τύπος αυτός, όμως, εκφράζει μόνο την απόδοση η οποία εξαρτάται από το μέγεθος εισόδου του. Όπως έχουμε αναφέρει, η αρχικοποίηση και ο τερματισμός ενός MapReduce job, προσθέτουν ένα αρκετά σημαντικό σταθερό κόστος στην εκτέλεση των join. Θα πρέπει, λοιπόν, να προσθέσουμε μια ακόμα σταθερά στον τύπο της απόδοσης, η

οποία θα εκφράζει αυτά τα σταθερά κόστη που συνεπάγεται η εκτέλεση της ξεχωριστής εργασίας. Άρα το συνολικό μοντέλο κόστους που θα χρησιμοποιήσουμε θα είναι:

$$Join = O\left(\sum_{i \in Inp} n_i o_i\right) + O\left(\sum_{i \in Inp} n_i * \sum_{j \notin Inp} o_j\right) + offset$$

4.2.7.3) Αλγόριθμος επιλογής του βέλτιστου πλάνου εκτέλεσης των join

Μπορούμε να χρησιμοποιήσουμε το μοντέλο κόστους που περιγράφηκε στην προηγούμενη ενότητα για να βρούμε το βέλτιστο πλάνο εκτέλεσης των join. Ως βέλτιστο πλάνο θεωρούμε αυτό που θα έχει το μικρότερο συνολικά κόστος εκτέλεσης βάση του μοντέλου κόστους μας. Σε αυτή την ενότητα θα δείξουμε ότι είναι δύσκολο υπολογιστικά να βρούμε το βέλτιστο αυτό πλάνο. Αρχικά θα ορίσουμε το πρόβλημά μας και δείξουμε την πολυπλοκότητά του.

Ορίζουμε το πρόβλημά μας ως ένα πρόβλημα αναζήτησης μικρότερης διαδρομής σε ένα γράφο. Έστω G(V, E), ένας κατευθυνόμενος γράφος με βάρη στις ακμές του, όπου κάθε κορυφή του αντιστοιχεί σε μια κατάσταση του ερωτήματος και κάθε ακμή του αντιστοιχεί στην εκτέλεση ενός συγκεκριμένου MapReduce join. Το βάρος κάθε ακμής προέρχεται από το μοντέλο κόστους του συγκεκριμένου join. Στο γράφο, αυτό, υπάρχει μια κορυφή που αντιστοιχεί στην αρχική κατάσταση του ερωτήματος, δηλαδή στην κατάσταση στην οποία δεν έχει εκτελεστεί κανένα join, και μια κορυφή που αντιστοιχεί στην τελική του κατάσταση, δηλαδή στην κατάσταση στην οποία έχουν εκτελεστεί όλα τα απαραίτητα join. Το πρόβλημά μας δηλαδή αντιστοιχεί σε ένα πρόβλημα εντοπισμού του μονοπατιού ελάχιστου κόστους από την αρχική κορυφή στην τελική κορυφή.

Αυτό το πρόβλημα είναι ενα κλασσικό πρόβλημα τεχνίτης νοημοσύνης αφού ο γράφος τον οποίο πρέπει να ερευνήσουμε μεγαλώνει εκθετικά με βάση τον αριθμό των διαφορετικών μεταβλητών που πρέπει να γίνουν join. Υπάρχουν πολλοί αλγόριθμοι που διευκολύνουν την επίλυση τέτοιου είδους προβλημάτων, όμως το πρόβλημα δεν σταματάει να είναι εκθετικής τάξης μεγέθους. Η χρησιμοποίηση ενός τέτοιου αλγορίθμου για την επιλογή του πλάνου δεν θα ήταν καλή επιλογή, αφού θα σπαταλούσαμε χρόνο, στην επιλογή του πλάνου, που δεν θα μπορούσαμε να κερδίσουμε από την βελτίωση της απόδοσης των join.

4.2.7.4) Άπληστος αλγόριθμος επιλογής του πλάνου εκτέλεσης των join

Όπως δείξαμε το πρόβλημα της επιλογής του βέλτιστου πλάνου είναι ένα εκθετικά δύσκολο πρόβλημα. Θα χρησιμοποιήσουμε, λοιπόν, έναν άπληστο αλγόριθμο για την επιλογή του πλάνου εκτέλεσης των join. Οι άπληστοι αλγόριθμοι δεν οδηγούν σε μια επιλογή που είναι βέλτιστη, αλλά με χρήση της κατάλληλης συνάρτησης επιλογής μπορούμε να δημιουργήσουμε ένα πλάνο που είναι αρκετά κοντά σε αυτή τη βέλτιστη επιλογή.

Βασικό χαρακτηριστικό των άπληστων αλγορίθμων είναι το γεγονός ότι σε κάθε βήμα διαλέγουμε την καλύτερη δυνατή επιλογή βάση μιας συνάρτησης αξιολόγησης. Οι βέλτιστες αυτές επιμέρους επιλογές δεν μας οδηγούν πάντα σε ένα συνολικά βέλτιστο μονοπάτι αλλά μπορούμε να εξασφαλίσουμε ένα σχετικά καλό συνολικό αποτέλεσμα. Ο κύριος λόγος που δεν ισχύει η αρχή της βελτιστότητας σε αυτό το πρόβλημα είναι το σταθερό κόστος εκτέλεσης ενός MapReduce job. Αυτό σημαίνει ότι μονοπάτια με

μικρότερο μήκος θα έχουν συνήθως καλύτερη συνολική απόδοση από μεγαλύτερα μονοπάτια.

Πριν προσδιορίσουμε τη συνάρτηση που θα χρησιμοποιούμε για την άπληστη επιλογή πρέπει να αναφέρουμε μερικά ακόμα προβλήματα που επηρεάζουν την επιλογή του πλάνου. Το βασικότερο από αυτά τα προβλήματα είναι ο πρακτικός προσδιορισμός της σταθερής καθυστέρησης κάθε MapReduce join. Το μοντέλο της απόδοσης των join μετράει αυτή την απόδοση σε τάξεις μεγέθους των bindings των διαφόρων queries του join. Για να έχουμε λοιπόν ένα ενιαίο μοντέλο που να περιέχει και την σταθερή καθυστέρηση, θα πρέπει να εκφράσουμε την σταθερή αυτή καθυστέρηση με βάση έναν αριθμό από bindings. Πρέπει δηλαδή να βρούμε τον αριθμό των bindings που θα χρειάζονταν για την επεξεργασία τους χρόνο ίσο με αυτή τη σταθερή καθυστέρηση. Αυτό δεν θα είναι ιδιαίτερα δύσκολο αφού θα επιλέξουμε μια προσεγγιστική τιμή που θα είναι προσαρμοσμένη κάθε φορά στο cluster στο οποίο τρέχουμε τις εργασίες.

Άλλο ένα πρόβλημα είναι το γεγονός ότι δεν μπορούμε να γνωρίζουμε τον ακριβή αριθμό των bindings εξόδου ενός join πριν την εκτέλεσή του. Αυτό σημαίνει ότι δεν θα μπορούσαμε να προβλέψουμε αποτελεσματικά την απόδοση ενός join που χρησιμοποιεί δεδομένα που προέρχονται από κάποιο άλλο. Αυτό θα ήταν ένα πολύ σοβαρό πρόβλημα σε περίπτωση που χρησιμοποιούσαμε κάποιον αλγόριθμο υπολογισμού του βέλτιστου μονοπατιού, αφού ο πλήρης γράφος του ερωτήματος δεν μπορεί να είναι διαθέσιμος εξαρχής. Η χρήση ενός άπληστου αλγορίθμου μας βοηθάει να ξεπεράσουμε τέτοιου είδους προβλήματα. Σε κάθε βήμα επιλέγουμε οριστικά ποιό join πρέπει να εκτελεστεί. Μετά την επιλογή μπορούμε να εκτελέσουμε το συγκεκριμένο join και να καταγράψουμε το μέγεθος των δεδομένων εξόδου του, ώστε να μπορούμε να εφαρμόσουμε ξανά το κριτήριο της άπληστης επιλογής στα νέα δεδομένα.

Σε αυτό το σημείο, πρέπει να θέσουμε τους βασικούς στόχους που θέλουμε να επιτύχουμε με τον άπληστο αλγόριθμο που θα χρησιμοποιήσουμε. Γνωρίζουμε ότι ο στόχος είναι η εύρεση του πλάνου που θα έχει το μικρότερο, συνολικά, χρόνο εκτέλεσης. Με μια πρώτη ματιά, λοιπόν, θα σκεφτόμασταν να επιλέγουμε κάθε φορά το join που θα έχει την καλύτερη πρόβλεψη απόδοσης. Αν εφαρμόσουμε αυτή την επιλογή, όμως, θα είναι σαν να μην λαμβάνουμε υπόψη μας το σταθερό κομμάτι που αντιστοιχεί στην αρχικοποίηση και τον τερματισμό ενός MapReduce join. Το σταθερό αυτό κομμάτι θα περιέχεται σε όλα τα δυνατά join και άρα δε θα υπολογίζεται. Έτσι βλέπουμε ότι θα δημιουργούσαμε πλάνα, τα οποία θα εκτελούσαν περισσότερες MapReduce εργασίες με αποτέλεσμα το συνολικό πλάνο να απέχει πολύ από το βέλτιστο. Μάλιστα έχουμε διαπιστώσει ότι ο χρόνος του σταθερού κομματιού της εκτέλεσης του MapReduce, αντιστοιχεί σε επεξεργασία πάρα πολύ μεγάλου όγκου από bindings.

Πρέπει, λοιπόν, να δημιουργήσουμε μια άπληστη επιλογή η οποία θα έχει δυο στόχους. Πρώτος στόχος της θα είναι η δημιουργία ενός συνολικού πλάνου, που θα έχει όσο το δυνατόν λιγότερες εργασίες MapReduce και δεύτερος στόχος της θα είναι να επιλέγει εργασίες, που θα έχουν όσο το δυνατόν μικρότερο χρόνο εκτέλεσης κάθε φορά. Οι δυο αυτοί στόχοι είναι εξίσου σημαντικοί και άρα, πρέπει να δημιουργήσουμε μια συνάρτηση επιλογής, που όχι μόνο να τους λαμβάνει υπόψη της, αλλά και να εξισορροπεί τις θετικές και αρνητικές επιπτώσεις τους στην επιλογή του πλάνου.

Αρχικά θα πρέπει να βρούμε ένα κριτήριο που να μας επιτρέπει να επιλέγουμε πλάνα που θα χρειάζονται τον ελάχιστο αριθμό βημάτων. Όπως έχουμε αναφέρει και σε προηγούμενη ενότητα ένα MapReduce job μπορεί να εκτελέσει ταυτόχρονα join διαφορετικών μεταβλητών, αρκεί κάθε επιμέρους query του συνολικού ερωτήματος να γίνεται join μόνο ως προς μια μεταβλητή του. Έστω, λοιπόν, ότι θέλουμε να εκτελέσουμε το παρακάτω SPARQL ερώτημα

```
SELECT ?X, ?Y, ?Z WHERE {
	?X rdf:type ub:GraduateStudent.
	?Y rdf:type ub:University.
	?Z rdf:type ub:Department.
	?X ub:memberOf ?Z.
	?Z ub:subOrganizationOf ?Y.
	?X ub:undergraduateDegreeFrom ?Y
}
```

Αν θέλουμε να εκτελέσουμε το join ως προς τη μεταβλητή X σημαίνει ότι δεν μπορούμε, ταυτόχρονα, να εκτελέσουμε join που περιέχουν τα BGP 1, 4, 6. Αυτά τα BGP διαθέτουν, εκτός από τη μεταβλητή X, τις μεταβλητές Y, Z. Άρα διαπιστώνουμε ότι η ολική εκτέλεση του join ως προς X αποκλείει την ταυτόχρονη, ολική εκτέλεση των join ως προς τις μεταβλητές Y και Z. Το αποτέλεσμα του join θα έχει δυο μεταβλητές με βάση τις οποίες θα πρέπει να γίνει join. Αυτός, ο αριθμός αποτελεί μια καλή προσέγγιση των join που πρέπει να υλοποιηθούν ακόμα για την ολοκλήρωση του ερωτήματος. Θα ονομάζουμε, λοιπόν, ως e-count(v) τον αριθμό των join μεταβλητών που θα υπάρχουν στο αποτέλεσμα που προκύπτει αν γίνει το ολικό join ως προς τη μεταβλητή v. Για παράδειγμα, αν στο παραπάνω SPARQL ερώτημα εκτελέσουμε πλήρως το join ως προς τη μεταβλητή X, δηλαδή εκτελέσουμε το join(X, XY, XZ), θα έχουμε ως αποτέλεσμα ένα pattern της μορφής (YZ). Το pattern, αυτό έχει δυο μεταβλητές που πρέπει να γίνουν join, άρα το e-count(X) είναι 2.

Μπορούμε, λοιπόν, χρησιμοποιώντας το e-count σαν άπληστη συνάρτηση επιλογής των join, να επιλέξουμε ένα πλάνο που θα έχει μικρό αριθμό από MapReduce joins. Μάλιστα, στο άρθρο [16], έχει αποδειχθεί ότι μια τέτοια άπληστη επιλογή οδηγεί σε πλάνα, που έχουν λογαριθμικό αριθμό από MapReduce joins σε σχέση με το συνολικό αριθμό των BGP queries του ερωτήματος. Πιο συγκεκριμένα, έχει αποδειχθεί ότι αν N είναι ο αριθμός των BGP queries ενός SPARQL ερωτήματος και K ο αριθμός των join variables, ο αριθμός των MapReduce join που θα εκτελεστούν είναι:

$$J = \begin{cases} 0, & N = 0, \\ 1, & N = 1 \text{ or } K = 1, \\ \min(\lceil 1.71 \log_2 N \rceil, K), & N, K > 1. \end{cases}$$

Join variables είναι οι μεταβλητές που βρίσκονται τουλάχιστον σε δυο διαφορετικά BGP και άρα πρέπει να γίνουν join.

Παρόλα αυτά, η χρήση του e-count ως συνάρτηση άπληστης επιλογής ικανοποιεί μόνο το στόχο της ελαχιστοποίησης των συνολικών join, αφού δε λαμβάνει καθόλου υπόψη το selectivity των join. Για να επιτύχουμε, λοιπόν, την εξισορρόπηση των δυο στόχων θα χρησιμοποιήσουμε ως συνάρτηση άπληστης επιλογής έναν γραμμικό συνδυασμό των δυο συναρτήσεων που αναφέρθηκαν. Θα χρησιμοποιήσουμε δηλαδή έναν γραμμικό συνδυασμό της συνάρτησης e-count και της συνάρτησης απόδοσης ενός join. Όπως παρατηρήσαμε και παραπάνω η συνάρτηση e-count μπορεί να χρησιμοποιηθεί ως προσέγγιση των join που απομένουν για την απάντηση του ερωτήματος. Άρα μπορούμε να χρησιμοποιήσουμε την παρακάτω συνάρτηση για την άπληστη επιλογή

$$gready(v) = offset * e_count(v) + \alpha\pi\acute{o}\delta o\sigma\eta_join(v)$$

όπου offset είναι μια σταθερά, η οποία επιλέγεται να είναι περίπου ίση με τον αριθμό των

bindings που μπορούν να επεξεργαστούν σε χρόνο περίπου ίσο με το χρόνο αρχικοποίησης και τερματισμού μιας εργασίας MapReduce. Από παραδείγματα που έχουμε κάνει φαίνεται πως αυτή η σταθερά είναι αρκετά μεγάλη και μπορεί να φτάνει σε κάποιες περιπτώσεις ακόμα και σε τάξεις εκατομμυρίων bindings.

4.2.7.5) Αναλυτική περιγραφή του άπληστου αλγορίθμου επιλογής των join

Σε αυτή την ενότητα θα παρουσιάσουμε αναλυτικά και με χρήση ψευδοκώδικα τον άπληστο αλγόριθμο επιλογής των join. Στη συνέχεια παρουσιάζουμε τον ψευδοκώδικα του αλγορίθμου.

```
Q := Remove_non-joining_variables(Q)
while Q != Empty do
        J := 1 //Total number of jobs
        V := {v1, ..., vk}        //Όλες οι μεταβλητές ταξινομημένες με αύξουσα σειρά της τιμής της
                                  //άπληστης συνάρτησης
        Jobj := Empty             //Λίστα που περιέχει τα join που θα εκτελεστούν στην τρέχουσα εργασία
        tmp := Empty              //Προσωρινή λίστα με τα αποτελέσματα του join
        for i = 1 to K do
                if Can-Eliminate(Q, vi )=true then
                        // μπορούμε να κάνουμε ολικό ή μερικό join ως προς τη μεταβλητή
                        tmp := tmp U Join_result(TP(Q, vi ))
                        Q := Q - TP(Q,vi)
                        Jobj := Jobj U join(TP(Q,vi))
                end if
        end for
        Q :=Q U tmp
        J := J +1
        executeJoin(Jobj)
 end while
```

Για την πλήρη περιγραφή του αλγορίθμου θα εκτελούμε τις αντίστοιχες ενέργειες χρησιμοποιώντας ως παράδειγμα το query:

```
SELECT ?X, ?Y, ?Z WHERE {
        ?X rdf:type ub:GraduateStudent.
        ?Y rdf:type ub:University.
        ?Z rdf:type ub:Department.
        ?X ub:memberOf ?Z.
        ?Z ub:subOrganizationOf ?Y.
        ?X ub:undergraduateDegreeFrom ?Y
}
```

Ο αλγόριθμος ξεκινάει με την απαλοιφή των nonjoining variables από το SPARQL query, αυτό σημαίνει ότι πρέπει να αφαιρέσουμε από τα triple pattern τις μεταβλητές που εμφανίζονται μόνο μια φορά σε όλο το query. Τέτοιες μεταβλητές δεν χρησιμοποιούνται για join και άρα δεν πρέπει να περιέχονται στη δομή λίστα των μεταβλητών προς επιλογή. Στο

παράδειγμά μας όλες οι μεταβλητές είναι joining variables και πρέπει να περιληφθούν στη λίστα Q, η οποία γίνεται Q={X, Y, Z, XZ, ZY, XY}. Στη συνέχεια, εκτελούμε έναν βρόχο while σε κάθε επανάληψη του οποίου εκτελούμε και ένα MapReduce join. Ο βρόχος, αυτός, σταματάει όταν δεν υπάρχουν πλέον patterns στη lista Q που χρειάζεται να γίνουν join.

Σε κάθε επανάληψη του βρόχου, ταξινομούμε όλες τις joining μεταβλητές με βάση την άπληστη συνάρτηση που επιλέξαμε στην προηγούμενη ενότητα. Για να υπολογίσουμε την τιμή της συνάρτησης πρέπει να κρατάμε στατιστικά στοιχεία για τα binding που περιέχουν τα διαφορετικά queries καθώς και να δημιουργούμε τα απαραίτητα στατιστικά στοιχεία για τα patterns που δημιουργούνται από εκτέλεση κάποιου join. Οι ταξινομημένες μεταβλητές αποθηκεύονται σε μια λίστα με αύξουσα σειρά. Για να εφαρμόσουμε το βήμα αυτό στο παράδειγμά μας θα χρειαστεί να γνωρίζουμε τα στατιστικά στοιχεία για τα επιμέρους queries του. Έστω, λοιπόν, ότι διαθέτουμε τα παρακάτω στατιστικά

BGP query	bindings
BGP1	X-100000
BGP2	Y-100
BGP3	Z-1000

Πίνακας 28: Μέγεθος BGP

BGP query	join_variable	other_variables
BGP4	X-100000	Z-1
BGP5	Z-1000	Y-1
BGP6	X-100000	Y-1

Πίνακας 29: SP_O index

BGP query	join_variable	other_variables
BGP4	Z-1000	X-100
BGP5	Y-100	Z-10
BGP6	Y-100	X-1000

Πίνακας 30: PO_S index

Στους παραπάνω πίνακες φαίνονται με λεπτομέρεια όλα τα απαραίτητα στατιστικά που θα χρησιμοποιήσουμε. Πρέπει, ακόμα, να ορίσουμε μια τιμή για το offset του MapReduce job. Έστω ότι η τιμή του offset είναι 100000. Στη συνέχεια ακολουθεί αναλυτικός υπολογισμός της άπληστης συνάρτησης για όλες τις μεταβλητές του ερωτήματος.

X: Βλέπουμε ότι για την μεταβλητή X δεν υπάρχει κάποιο BGP που να έχει μεγάλο selectivity, οπότε όπως αναλύσαμε και σε προηγούμενη ενότητα επιλέγουμε ως είσοδο του join όλα τα BGP. Η απόδοση λοιπόν του join ως προς X θα είναι:

map: 100000 + 100000 + 100000
reduce: 0
συνολικά: 300000

Το e-count(X) είναι ίσο με 2 αφού αποκλείει τις μεταβλητές Y και Z. Άρα συνολικά θα έχουμε ως τιμή της άπληστης συνάρτησης:

2*100000 + 300000 = 500000

Y: Βλέπουμε ότι για την μεταβλητή Y δεν υπάρχει κάποιο BGP που να έχει μεγαλύτερο selectivity από τα υπόλοιπα, οπότε όπως αναλύσαμε και σε προηγούμενη ενότητα επιλέγουμε ως είσοδο του join όλα τα BGP. Η απόδοση λοιπόν του join ως προς X θα είναι:

map: 100+ 100 * 10 + 100 *10
reduce: 0
συνολικά: 2100

Το e-count(Y) είναι ίσο με 2 αφού αποκλείει τις μεταβλητές X και Z. Άρα συνολικά θα έχουμε ως τιμή της άπληστης συνάρτησης:

2*100000 + 2100 = 202100

Z: Βλέπουμε ότι για την μεταβλητή Z δεν υπάρχει κάποιο BGP που να έχει μεγαλύτερο selectivity από τα υπόλοιπα, οπότε όπως αναλύσαμε και σε προηγούμενη ενότητα επιλέγουμε ως είσοδο του join όλα τα BGP. Η απόδοση λοιπόν του join ως προς X θα είναι:

map: 1000 + 1000 * 1 + 1000 *100
reduce: 0
συνολικά: 101100

Το e-count(Z) είναι ίσο με 2 αφού αποκλείει τις μεταβλητές Y και X. Άρα συνολικά θα έχουμε ως τιμή της άπληστης συνάρτησης:

2*100000 + 101100 = 301100

Άρα, οι μεταβλητές ταξινομούνται με την σειρά {Y, Z, X}. Στη συνέχεια εκτελούμε ένα βρόχο ο οποίος ελέγχει με ταξινομημένη σειρά όλες τις μεταβλητές. Για κάθε μεταβλητή, με τη συνθήκη Can-Eliminate(Q, v_i) ελέγχουμε αν στο Q υπάρχουν δυο ή περισσότερα patterns που την περιέχουν. Αν υπάρχουν τότε μπορούμε να κάνουμε ολικό ή μερικό join ως προς τη μεταβλητή. Για να το κάνουμε αυτό χρησιμοποιούμε τον παρακάτω κώδικα:

tmp := tmp U Join_result(TP(Q, v_i))
Q := Q - TP(Q,v_i)
Job_j := Job_j U join(TP(Q,v_i))

Η συνάρτηση TP(Q, v_i) μας επιστρέφει όλα τα pattern του Q στα οποία περιέχεται η συγκεκριμένη μεταβλητή. Αρχικά λοιπόν βάζουμε στην προσωρινή λίστα το αποτέλεσμα του join που θα έχει είσοδο όλα τα patterns της μεταβλητής. Στη συνέχεια, αφαιρούμε τα patterns, αυτά, από το query αφού θα επεξεργάζονται ήδη και δεν μπορούν να ξαναεπιλεχθούν. Τέλος, προσθέτουμε στη δομή Job, όπου αποθηκεύουμε τις εργασίες του MapReduce join, το join που μόλις επιλέξαμε.

Στο παράδειγμά μας, πρώτα θα ελέγξουμε την μεταβλητή Y. Η μεταβλητή αυτή μπορεί να γίνει join και τα patterns που επιστρέφει η συνάρτηση TP είναι τα Y, ZY, XY. Το αποτέλεσμα του Join_result(Y, ZY,XY) είναι ZX. Στη συνέχεια από το Q αφαιρούνται τα παραπάνω patterns και προσθέτουμε το join στην αντίστοιχη δομή. Η επόμενη μεταβλητή είναι το Z, που μπορεί να γίνει μερικώς join αφού η συνάρτηση TP επιστρέφει τα Z, XZ. Το

αποτέλεσμα του Join_result(Z, XZ) είναι ZX, το Z παραμένει αφού δεν έγινε ολική αφαίρεσή του. Στη συνέχεια από το Q αφαιρούνται τα παραπάνω patterns και προσθέτουμε το join στην αντίστοιχη δομή. Τέλος εξετάζουμε το X το οποίο δεν μπορεί να γίνει join αφού υπάρχει μόνο ένα διαθέσιμο pattern του.

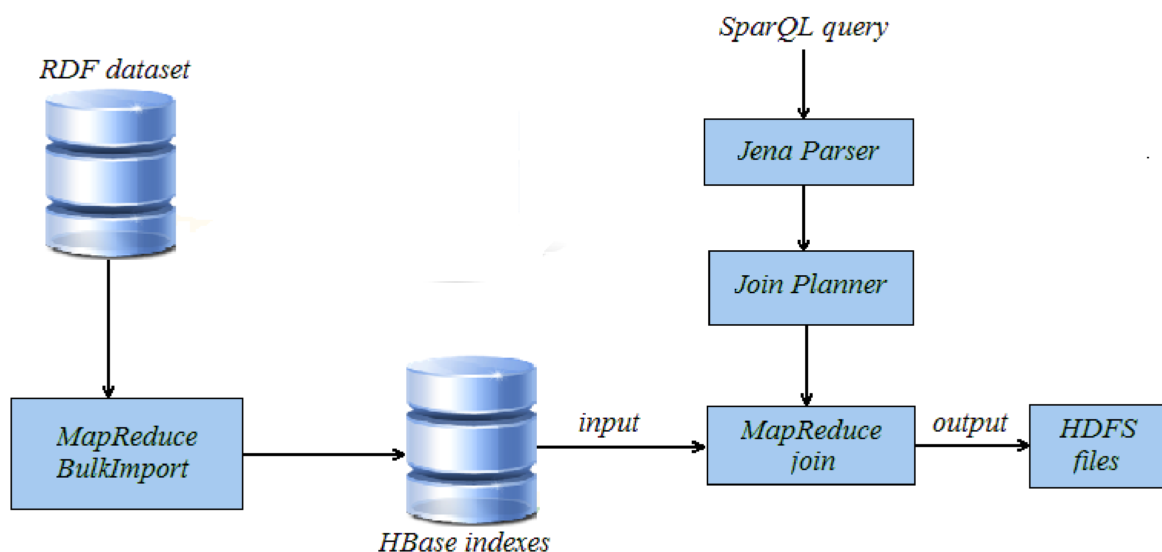
Άρα στο πρώτο Mapreduce job, θα εκτελέσουμε 2 join, το πρώτο ως προς τη μεταβλητή Y με είσοδο τα patterns Y, ZY, XY και το δεύτερο ως προς Z με είσοδο τα patterns Z, XZ. Στη συνέχεια εκτελούμε το MapReduce job και ανανεώνουμε τη δομή Q, η οποία γίνεται Q={X, ZX, ZX}. Ο αλγόριθμος συνεχίζει μέχρι να μην υπάρχει κανένα pattern στο Q, τα αποτελέσματα του ερωτήματος βρίσκονται στο αρχείο εξόδου που προέκυψε από το τελευταίο MapReduce job.

Από το παραπάνω παράδειγμα, διαπιστώνουμε ότι ο συγκεκριμένος αλγόριθμος επιτυγχάνει μια αρκετά καλή χρησιμοποίηση των MapReduce joins και πετυχαίνει να δημιουργήσει ένα πλάνο εκτέλεσης των join που είναι αρκετά κοντά στο βέλτιστο. Ακόμα βλέπουμε ότι ο αλγόριθμος τρέχει ταυτόχρονα με την εκτέλεση των join, και προσθέτει για κάθε join μια πολυπλοκότητα της τάξης του O(K*log(K)) για την ταξινόμηση των μεταβλητών, όπου K ο αριθμός των joining variables. Ο αριθμός K είναι σχετικός μικρός για όλα τα SPARQL ερωτήματα, οπότε μπορούμε να πούμε ότι ο αλγόριθμος αυτός προσθέτει μια αμελητέα καθυστέρηση σε σχέση με το χρόνο εκτέλεσης του κάθε join.

5) Υλοποίηση Συστήματος

5.1) Εισαγωγή

Σε αυτή την ενότητα θα παρουσιάσουμε τον τρόπο με τον οποίο υλοποιήσαμε το σύστημά μας με χρήση διαφόρων τεχνολογιών όπως οι MapReduce, Hbase, Jena. Θα μελετήσουμε διεξοδικά όλα τα επιμέρους συστήματα, που δημιουργήσαμε, και θα εξηγήσουμε αναλυτικά τον κώδικα που φτιάξαμε για κάθε ένα από αυτά. Τέλος θα αξιολογήσουμε τον κώδικα αυτό και θα προσπαθήσουμε να αναδείξουμε τα πλεονεκτήματα καθώς και τα μειονεκτήματά του. Παρακάτω βλέπουμε τη βασική αρχιτεκτονική του συστήματός μας και τα κύρια κομμάτια, που πρέπει να υλοποιήσουμε στο σύστημά μας. Τα κομμάτια του συστήματος θα αναλυθούν στη συνέχεια σε αντίστοιχες ενότητες.



Σχήμα 10: Αρχιτεκτονική του συστήματος

5.2) Δημιουργία πινάκων index και εισαγωγή των δεδομένων με χρήση του MapReduce

Ένα από τα βασικότερα κομμάτια του συστήματός μας είναι τα index που χρησιμοποιούμε για την αποθήκευση των triples της βάσης μας. Όπως αναφέραμε στην προηγούμενη ενότητα υπάρχουν δυο διαφορετικά είδη index. Μετά από μελέτη και σύγκριση αυτών των index επιλέξαμε να χρησιμοποιήσουμε τα index SP_O, OS_P, PO_S. Τα 3 αυτά index είναι αρκετά ώστε να μπορούμε να απαντήσουμε αποδοτικά όλα τα διαφορετικά query patterns. Πρέπει λοιπόν να δημιουργήσουμε ένα MapReduce job, το οποίο θα παίρνει στην είσοδό του αρχεία με τα triples της βάσης μας και θα δημιουργεί τους πίνακες των index.

5.2.1) Συμπίεση δεδομένων με χρήση hash function

Όπως έχουμε αναφέρει τα RDF triples γράφονται με τη μορφή URI, η οποία μας επιτρέπει να έχουμε αναφορές που καθορίζονται από συγκεκριμένους, παγκόσμια αναγνωρισμένους, φορείς και να μην χρησιμοποιούμε απλά string. Η χρήση των URI, όπως εξηγήσαμε και στην εισαγωγή της εργασίας, προσφέρει πολλά πλεονεκτήματα στο μοντέλο του RDF. Οι βάσεις, λοιπόν, που θέλουμε να εισάγουμε στα index είναι και αυτές γραμμένες με μορφή URI.

Για να αποθηκεύσουμε, τέτοιες βάσεις, στα index θα έπρεπε να χρησιμοποιούμε κάθε φορά ως τιμή το πλήρες URI. Τα URI, όμως, αποτελούνται από μεγάλα string, τα οποία χρειάζονται και μεγάλο χώρο αποθήκευσης. Είναι, λοιπόν, αναγκαίο να χρησιμοποιήσουμε κάποιο τρόπο συμπίεσης των URI ώστε να καταφέρουμε να μειώσουμε τον όγκο που θα καταλαμβάνουν οι πίνακες των index μας. Ακόμα, στο σύστημά μας θα δημιουργήσουμε 3 index δηλαδή κάθε resource θα αποθηκευτεί τουλάχιστον 3 φορές στους πίνακες. Θα μπορούσαμε, λοιπόν, αντί να χρησιμοποιούμε κάθε φορά το πλήρες URI, να χρησιμοποιούμε έναν δείκτη στο πραγματικό string του URI. Έτσι μπορούμε να εξοικονομήσουμε μεγάλο χώρο αποθήκευσης και να καταφέρουμε να αποθηκεύσουμε αποδοτικά τη βάση μας.

Για να επιτύχουμε τον παραπάνω στόχο, πρέπει να αποθηκεύουμε μια μόνο φορά το κάθε διαφορετικό URI, και για όλες τις αναφορές σε αυτό να χρησιμοποιούμε έναν δείκτη σε αυτή την εγγραφή. Θα δημιουργήσουμε λοιπόν έναν ακόμα πίνακα HBase, που θα τον ονομάσουμε names, και θα περιέχει σε κάθε γραμμή του το όνομα ενός resource. Σε αυτό το σημείο πρέπει να ορίσουμε τον τρόπο με τον οποίο θα κάνουμε την δεικτοδότηση καθώς και τις ιδιότητες που πρέπει να έχουν οι δείκτες που θα χρησιμοποιήσουμε.

Αρχικά παρατηρούμε ότι, για να είναι αποδοτική μια τέτοια λύση, πρέπει να διαθέτουμε έναν πολύ γρήγορο αλγόριθμο για την εύρεση του δείκτη αν έχουμε στη διάθεσή μας το URI. Οι βάσεις, που θα θέλουμε να αποθηκεύσουμε στα index, θα έχουν μορφή URI, άρα για κάθε resource της βάσης θα πρέπει να βρίσκουμε αν το έχουμε αποθηκεύσει και να ανακτούμε το δείκτη του. Αυτή η διαδικασία θα γίνεται για κάθε resource της βάσης και άρα είναι απαραίτητο να είναι εξαιρετικά γρήγορη, ώστε να πετύχουμε αποδοτική εισαγωγή στους πίνακες.

Ακόμα, οι δείκτες που θα χρησιμοποιήσουμε πρέπει να έχουν συγκεκριμένες ιδιότητες για να μπορέσουν να χρησιμοποιηθούν στα index. Όπως αναφέραμε στο κεφάλαιο του σχεδιασμού, η λειτουργία των index, βασίζεται κατά κύριο λόγο στην λεξικογραφική ταξινόμηση που πραγματοποιείται στην HBase. Αρχικά, για να δημιουργήσουμε σωστά index πρέπει, όλοι οι δείκτες που αντιστοιχούν στο ίδιο resource, να είναι λεξικογραφικά ίσοι. Αυτό είναι μια απαραίτητη ιδιότητα, αφού θέλουμε οι εγγραφές του index που αντιστοιχούν στο ίδιο resource να ομαδοποιούνται μαζί. Διαπιστώνουμε, δηλαδή, ότι πρέπει να υπάρχει μια 1 προς 1 σχέση μεταξύ των URI και των δεικτών. Για να εκμεταλλευτούμε και τα υπόλοιπα πλεονεκτήματα της λεξικογραφικής ταξινόμησης, πρέπει οι δείκτες μας να αποθηκεύονται σε μια μορφή που έχει δυνατότητες λεξικογραφικής σύγκρισης. Για παράδειγμα, μπορούμε να χρησιμοποιήσουμε ως δείκτες αριθμούς τύπου integer.

Ένας αποδοτικός τρόπος, για να υλοποιήσουμε όλα τα παραπάνω, είναι η χρήση μιας hash function για την δημιουργία των δεικτών άμεσα από τα URI strings. Το βασικότερο πλεονέκτημα της χρήσης μιας hash function είναι το γεγονός ότι μπορούμε πολύ γρήγορα να βρούμε την τιμή του hash για ένα συγκεκριμένο URI. Έχουμε επιλέξει να χρησιμοποιήσουμε την συνάρτηση Jenkin's hash η οποία δέχεται σαν είσοδο έναν πίνακα από bytes και επιστρέφει έναν ακέραιο αριθμό τύπου integer. Κάθε φορά, λοιπόν, που θέλουμε να βρούμε το δείκτη που αντιστοιχεί σε ένα URI αρκεί να μετατρέψουμε το string του URI σε bytes και στη συνέχεια να το περάσουμε από τη συνάρτηση hash. Αυτή η

διαδικασία είναι εξαιρετικά γρήγορη και μπορούμε να την χρησιμοποιήσουμε για την εισαγωγή των δεδομένων στα index.

Η συνάρτηση του Jenkin's hash μας εξασφαλίζει ότι δυο διαφορετικά URI θα έχουν και διαφορετική τιμή hash. Η τιμή του hash εξαρτάται από όλα τα bit του string και μπορεί να είναι εντελώς διαφορετική ακόμα και στην περίπτωση που δυο string διαφέρουν μόνο κατά ένα χαρακτήρα. Η πιθανότητα δυο διαφορετικά string να αντιστοιχούν στο ίδιο hash, χρησιμοποιώντας αυτή τη συνάρτηση, είναι μια στα 2^{32} string. Η πιθανότητα αυτή είναι αρκετά καλή για τη χρησιμοποίηση στο σύστημά μας και μας εξασφαλίζει ότι δεν θα έχουμε κάποια σύμπτωση των δεικτών. Βέβαια, σε περίπτωση που θέλουμε να μειώσουμε ακόμα αυτήν την πιθανότητα μπορούμε εύκολα να αλλάξουμε τη συνάρτηση hash και να χρησιμοποιήσουμε κάποια άλλη όπως είναι η md5 που προσφέρει μικρότερη πιθανότητα σφάλματος. Η χρήση, όμως, μιας καλύτερης συνάρτησης, όπως η md5, συνεπάγεται και αύξηση στο μέγεθος των τιμών των hash άρα και σε συνολική αύξηση του όγκου των δεδομένων της βάσης μας. Για παράδειγμα, με χρήση του Jenkin's hash έχουμε τιμές hash των 4 byte, ενώ με χρήση του md5 τιμές hash των 16 byte. Η χρήση δεικτών md5, λοιπόν, συνεπάγεται ότι η βάση μας θα έχει τετραπλάσιο συνολικό όγκο σε σχέση με τη χρήση των δεικτών Jenkin's. Η μείωση του όγκου που μας προσφέρει το Jenkin's hash είναι το κυριότερο κριτήριο επιλογής του. Έτσι κι αλλιώς μπορούμε κατά τη διαδικασία εισαγωγής των δεδομένων στη βάση να ελέγχουμε τέτοιου είδους σφάλματα και αν κρίνεται απαραίτητο να μεταβαίνουμε σε κάποιο άλλο είδος hash function.

Όπως, γνωρίζουμε, οι συναρτήσεις hash δεν είναι αντιστρέψιμες, δηλαδή δεν υπάρχει τρόπος να βρούμε μια αντίστροφη συνάρτηση, που να μας μετατρέπει την τιμή hash στο αντίστοιχο URI από το οποίο προήλθε. Η αντιστοίχηση όμως των τιμών hash στα URI είναι και αυτή μια σημαντική λειτουργία που πρέπει να υπάρχει στο σύστημά μας. Για την επίτευξη μιας τέτοιας αντιστοίχησης θα δημιουργήσουμε έναν ακόμα πίνακα HBase, στον οποίο θα αποθηκεύσουμε όλους τους συνδυασμούς hash-URI. Ο πίνακας, αυτός, θα έχει μία γραμμή για κάθε τέτοιο συνδυασμό, το rowid της οποία θα είναι η τιμή του hash. Κάθε γραμμή θα έχει μια μόνο στήλη, η τιμή της οποίας θα είναι το πλήρες string, του αντίστοιχου URI. Κάθε φορά που θέλουμε να βρούμε το URI που αντιστοιχεί σε έναν δείκτη, απλά θα παίρνουμε την αντίστοιχη τιμή από τον πίνακα.

Υλοποιώντας τα παραπάνω έχουμε δημιουργήσει μια 1 προς 1 αντιστοίχηση μεταξύ URI και δεικτών και ταυτόχρονα έχουμε καταφέρει να πετύχουμε σημαντική συμπίεση του όγκου της βάσης μας. Χωρίς την παραπάνω τροποποίηση θα χρησιμοποιούσαμε στους πίνακες index, ως τιμές, το πλήρες URI. Ένα τέτοιο URI string θα χρειαζόταν τουλάχιστον 50 byte για την αποθήκευσή του, ενώ οι τιμές hash χρειάζονται μόνο 4 byte η καθεμία. Ειδικά στο σύστημά μας, που θα διαθέτει 3 index, κάθε τέτοιο string θα παρουσιαζόταν τουλάχιστον 3 φορές στη βάση μας. Με χρήση, λοιπόν, των hash τιμών αποθηκεύουμε μια φορά κάθε ένα από τα URI και στα index χρησιμοποιούμε τις τιμές hash. Καταφέρνουμε, λοιπόν, να πετύχουμε μια εξαιρετικά μεγάλη συμπίεση των δεδομένων μας.

Θα μπορούσαμε να υλοποιήσουμε κάτι τέτοιο και με χρήση ενός παραδοσιακού αλγορίθμου συμπίεσης, οι περισσότεροι όμως από αυτούς απαιτούν την επεξεργασία όλων των δεδομένων και δεν θα είχαν καλή απόδοση αν συμπιέζαμε κάθε τιμή ξεχωριστά. Αυτό θα ήταν μεγάλο πρόβλημα για το σύστημά μας και θα οδηγούσε σε μεγάλες καθυστέρησης κατά την εισαγωγή των δεδομένων στα index.

Ένα ακόμα πλεονέκτημα, της χρήσης της συνάρτησης hash, είναι η δημιουργία δεικτών σταθερού μεγέθους και τύπου integer. Η χρήση δεικτών τύπου integer διευκολύνει ιδιαίτερα την επεξεργασία της λεξικογραφικής ταξινόμησής τους. Η λεξικογραφική σύγκριση δυο αριθμών integer ταυτίζεται με την αριθμητική τους σύγκριση, γεγονός που τους καθιστά πολύ πρακτικούς. Ακόμα, με την χρήση τιμών σταθερού μεγέθους, καταφέρνουμε να δημιουργήσουμε ένα πιο σταθερό και αξιόπιστο σχήμα αποθήκευσης.

5.2.2) Api Import vs Bulk Import

Σε αυτή την ενότητα, θα εξετάσουμε το σχεδιασμό μιας MapReduce εργασίας η οποία θα μπορέσει να δημιουργήσει αποδοτικά τις απαραίτητες δομές για την λειτουργία του συστήματός μας. Η εργασία αυτή, θα λαμβάνει ως είσοδο αρχεία που θα περιέχουν τα RDF triples της βάσης μας και θα δημιουργεί από αυτά 4 πίνακες HBase. Ο πρώτος πίνακας θα είναι αυτός, που περιγράφηκε στην προηγούμενη ενότητα και περιέχει όλες τις αντιστοιχίες hash-URI. Οι υπόλοιποι 3 πίνακες θα είναι τα index της βάσης μας τα οποία θα είναι συγκεκριμένα τα SP_O, OS_P και PO_S. Οι ιδιότητες και το μοντέλο των πινάκων, που θα περιέχουν αυτά τα index, παρουσιάστηκαν στο αντίστοιχο κεφάλαιο του σχεδιασμού.

Εδώ, λοιπόν, θα αναζητήσουμε τον πλέον αποδοτικό τρόπο δημιουργίας αυτών των πινάκων με χρήση του πλαισίου MapReduce. Στόχος μας είναι να δημιουργήσουμε έναν κώδικα, ο οποίος θα είναι σε θέση να αντιμετωπίσει αποτελεσματικά μεγάλες σε όγκο βάσεις δεδομένων σε αποδεκτό χρόνο. Η χρήση του MapReduce, θα μας βοηθήσει να παραλληλοποιήσουμε την εκτέλεση των απαραίτητων εργασιών, ώστε να μπορούμε να βελτιώσουμε την απόδοση του συστήματός μας με προσθήκη επιπλέον πόρων.

Βασική εργασία, που πρέπει να εκτελεστεί στο συγκεκριμένο MapReduce job, είναι η δημιουργία πινάκων HBase και εισαγωγή δεδομένων σε αυτούς. Υπάρχουν δυο τρόποι, με τους οποίους μπορούμε να εκτελέσουμε μια τέτοια εργασία.

Ο πρώτος τρόπος είναι να δημιουργήσουμε τους πίνακες πριν την εργασία MapReduce και στη συνέχεια μέσα στις συναρτήσεις map και reduce να εισάγουμε δεδομένα σε αυτούς μέσω του Api της HBase. Το Api της HBase μας προσφέρει μεθόδους, όπως η put, με τις οποίες μπορούμε να εισάγουμε ένα συγκεκριμένο cell ή μια γραμμή στον πίνακά μας. Η χρήση, αυτής της μεθοδολογίας δεν θα ήταν ιδιαίτερα αποδοτική, αφού το Api της HBase βασίζεται κυρίως στον master κόμβο του cluster της HBase. Όταν κάποιος κόμβος του cluster θέλει να εκτελέσει μια εγγραφή μέσω του Api, πρέπει να επικοινωνήσει με τον HMaster ο οποίος είναι ο κεντρικός κόμβος της HBase και στη συνέχεια να στείλει την αντίστοιχη εγγραφή σε κάποιον από τους κόμβους της HBase. Αυτή η ανάγκη επικοινωνίας με τον κεντρικό κόμβο είναι και το αδύνατο σημείο της τεχνικής αυτής. Όπως αναφέραμε η εισαγωγή των δεδομένων στα index θα γίνεται ταυτόχρονα από όλους τους υπολογιστές του cluster μας. Αυτό σημαίνει ότι οι υπολογιστές αυτοί θα βομβαρδίζουν συνεχώς τον Hmaster με αιτήματα εισαγωγής νέων δεδομένων. Χρησιμοποιώντας μια τέτοια τεχνική, διαπιστώνουμε ότι τα πλεονεκτήματα της παράλληλης εκτέλεσης χάνονται, αφού οι εγγραφές σειριοποιούνται στον Hmaster. Εκτός από το μειονέκτημα της σειριοποίησης, μπορούμε εύκολα να διαπιστώσουμε ότι ο master θα έχει μεγάλο φόρτο εργασίας, με αποτέλεσμα την περεταίρω καθυστέρηση των αιτημάτων εισαγωγής στους πίνακες.

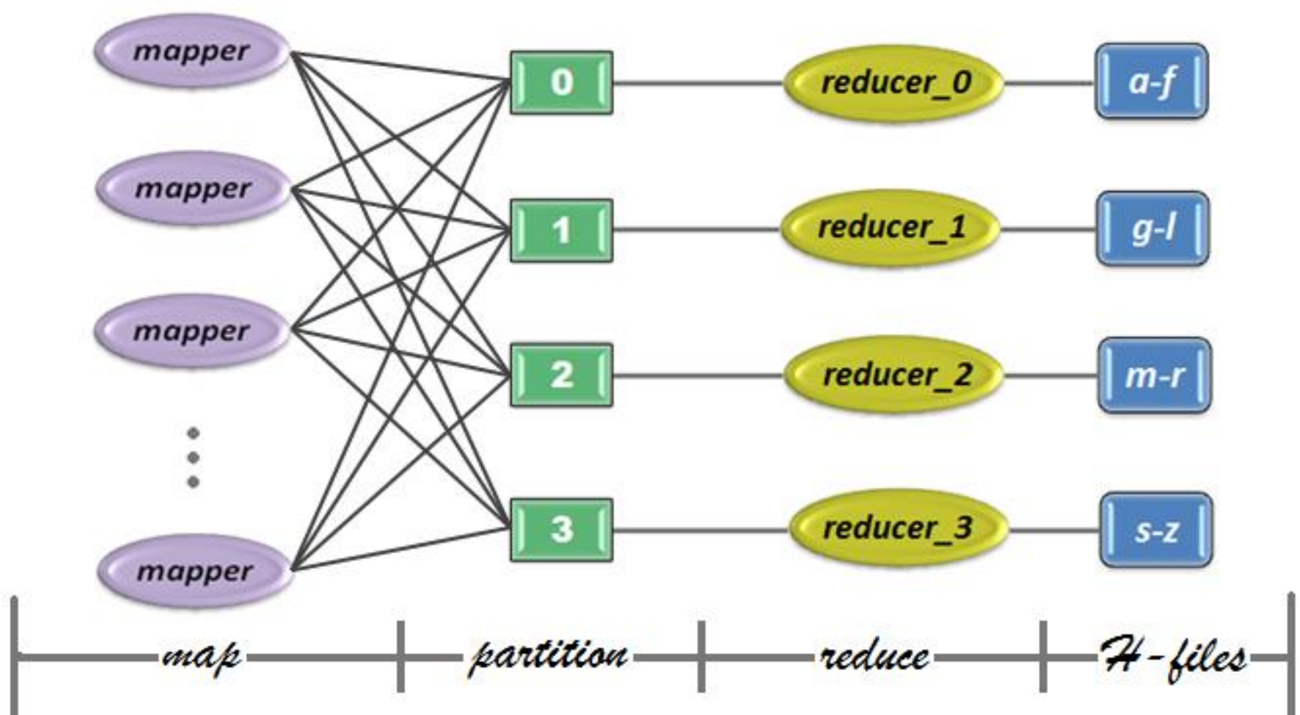
Πειράματα, που έχουμε εκτελέσει, με χρήση εισαγωγής μέσω του Api έδειξαν ότι η απόδοση ενός τέτοιου συστήματος δεν είναι καθόλου ικανοποιητική. Μάλιστα, σε περιπτώσεις πειραμάτων με μεγάλο αριθμό από εισαγωγές, δεν έχουμε μόνο δραματική πτώση της απόδοσης, αλλά υπάρχει και πιθανότητα πτώσης των servers της HBase με αποτέλεσμα την πλήρη αποτυχία της εργασίας. Βλέπουμε, λοιπόν, ότι η τεχνική της εισαγωγής μέσω Api παρουσιάζει πολλά μειονεκτήματα και δεν θα αποτελούσε καλή λύση για την υλοποίηση του συστήματός μας.

Η δεύτερη τεχνική που μπορούμε να χρησιμοποιήσουμε για την δημιουργία των πινάκων ονομάζεται BulkImport. Η βασική λογική, αυτής της τεχνικής, είναι η αποφυγή της χρήσης της HBase κατά τη διάρκεια εκτέλεσης του MapReduce job. Για να πετύχουμε κάτι τέτοιο θα πρέπει η MapReduce εργασία μας, να παράγει ολοκληρωμένα Hfiles. Τα Hfiles είναι η βασική δομή αποθήκευσης των δεδομένων που χρησιμοποιείται από την HBase.

Ένα Hfile περιέχει τις ταξινομημένες γραμμές μιας περιοχής του πίνακα και τα δεδομένα τους. Ακόμα, περιέχει όλες τις απαραίτητες δομές για την δεικτοδότηση των rowid και την αποδοτική αναζήτηση εγγραφών μέσα σε αυτό.

Αντί, λοιπόν, να χρησιμοποιούμε κάθε φορά το Api για την εισαγωγή ενός cell στον πίνακα, μπορούμε να παράγουμε ένα ζευγάρι εξόδου της μορφής key/value. Το key του ζευγαριού θα αντιστοιχεί στο rowid της γραμμής, ενώ το value θα περιέχει το συνδυασμό "collumn_family:qualifier, value" του cell που θέλουμε να αποθηκεύσουμε. Για την εγγραφή των ζευγαριών αυτών σε Hfile χρησιμοποιούμε ένα OutputFormat που περιέχεται στη HBase και ονομάζεται HfileOutputFormat. Το HfileOutputFormat αναλαμβάνει τη δημιουργία όλων των δομών και την αποθήκευση των ζευγαριών μας σε ολοκληρωμένα Hfile. Μετά το τέλος της εργασίας του MapReduce, χρησιμοποιούμε κατάλληλο κώδικα που παρέχεται στο πακέτο της HBase για την σύνδεση αυτών των Hfile με τον αντίστοιχο πίνακα της HBase.

Η μόνη προϋπόθεση για την χρήση του HfileOutputFormat είναι ότι πρέπει οι reducers να παράγουν τα ζευγάρια key/value εξόδου, με λεξικογραφική σειρά ως προς τα κλειδιά τους. Αυτό είναι απαραίτητο για τη δημιουργία των Hfiles αφού, όπως αναφέραμε, τα Hfiles αποθηκεύουν τις εγγραφές με λεξικογραφική σειρά. Το βασικό στοιχείο είναι ότι τα ζευγάρια αυτά πρέπει να ακολουθούν μια συνολική ταξινόμηση για όλους τους reducers και να μην είναι απλά ταξινομημένα σε σχέση με τα υπόλοιπα ζευγάρια του ίδιου reducer.



Σχήμα 11: Ολική διάταξη

Όπως βλέπουμε από το παραπάνω σχήμα, κάθε reducer παράγει ένα Hfile το οποίο πρέπει να περιέχει μια περιοχή των κλειδιών του πίνακα HBase. Έστω, λοιπόν ότι χρησιμοποιούμε τα κλειδιά των εγγραφών που θέλουμε να τοποθετήσουμε στον πίνακα ως κλειδιά των ενδιάμεσων ζευγαριών του MapReduce. Η ταξινόμηση που πραγματοποιεί το MapReduce στα ενδιάμεσα, αυτά, κλειδία δεν θα μας εξασφάλιζε και την πλήρη ταξινόμηση που απαιτείται. Το μόνο που κάνει η ταξινόμηση του MapReduce είναι να ταξινομεί τα κλειδιά στο εσωτερικό του κάθε partition. Οι mappers δημιουργούν τα ενδιάμεσα ζευγάρια key/value και χρησιμοποιούν τη συνάρτηση του Partitoner για να

επιλέξουν το partition, στο οποίο θα τοποθετήσουν το συγκεκριμένο ζευγάρι. Στη συνέχεια εκτελείται ταξινόμηση εντός των ξεχωριστών partitions. Ο προεπιλεγμένος Partitioner, που χρησιμοποιείται στο MapReduce εκτελεί τη συνάρτηση hash(key)modR όπου R ο αριθμός των διαφορετικών partitions. Χρησιμοποιώντας αυτή τη συνάρτηση έχουμε σε κάθε partition ζευγάρια που μπορεί να προέρχονται από οποιαδήποτε περιοχή των κλειδιών του πίνακα.

Για να πετύχουμε, λοιπόν, ολική ταξινόμηση των ζευγαριών εξόδου θα πρέπει να δημιουργήσουμε τον δικό μας Partitioner, ο οποίος θα αναλαμβάνει να τοποθετεί τα ενδιάμεσα ζευγάρια key/value στις περιοχές στις οποίες ανήκουν. Βέβαια, η δημιουργία του Partitioner απαιτεί την πλήρη γνώση του εύρους και του είδους των κλειδιών που θα χρησιμοποιηθούν. Αυτό είναι το μεγαλύτερο μειονέκτημα της τεχνικής του BulkImport. Για παράδειγμα, αν τα κλειδιά ήταν απλές λέξεις με λατινικούς χαρακτήρες και θέλαμε να χρησιμοποιήσουμε 4 reducers, θα μπορούσαμε να χωρίσουμε τις λέξεις στις 4 περιοχές a-f, g-l, m-r, s-z. Εύκολα μπορούμε τώρα να δημιουργήσουμε έναν Partioner, ο οποίος το μόνο που πρέπει να κάνει είναι να κοιτάει σε ποιά περιοχή ανήκει το πρώτο γράμμα κάθε κλειδιού. Με τη χρήση ενός τέτοιου Partitioner, καταφέρνουμε να επιτύχουμε ολική ταξινόμηση και να εισάγουμε σωστά τα δεδομένα μας στον πίνακα.

Η απόδοση, όμως, του παραπάνω Partitioner μπορεί να παρουσιάσει αρκετά προβλήματα τα οποία δεν θα παρουσιάζονταν με χρήση του προεπιλεγμένου Partitioner του MapReduce. Ο προεπιλεγμένος Partitioner έχει σχεδιαστεί ώστε να κατανέμει εξίσου τα κλειδιά στα διαφορετικά partitions. Αυτό σημαίνει ότι όλοι οι reducers έχουν περίπου ίδια ποσότητα από δεδομένα για επεξεργασία. Η ισοκατανομή επιτυγχάνεται με τη χρήση της hash function, η οποία έχει την ιδιότητα να παράγει τιμές, με ίση πιθανότητα, σε όλο το διάστημα των τιμών της. Με την ισοκατανομή πετυχαίνουμε μεγάλη παραλληλοποίηση της εκτέλεσης και έτσι κερδίζουμε σε απόδοση και δυνατότητα κλιμάκωσης του συστήματος.

Αντίθετα, ο Partitioner, που χρησιμοποιήσαμε προηγουμένως, δεν μας εξασφαλίζει την ιδιότητα της ισοκατανομής. Είναι πιθανό να υπάρχουν πολύ περισσότερα κλειδιά που βρίσκονται στο διάστημα a-f, από ότι κλειδιά που βρίσκονται στο διάστημα s-z. Με χρήση αυτού του Partitioner, μπορεί να προκύψουν εντελώς διαφορετικά σε όγκο partitions, με αποτέλεσμα κάποιοι reducers να υπερφορτώνονται και κάποιοι να μένουν ανενεργοί. Αυτό οδηγεί σε μη αξιοποίηση των διαθέσιμων κόμβων και φυσικά σε μειωμένη απόδοση.

Πρέπει, λοιπόν, να είμαστε πολύ προσεκτικοί κατά τη διάρκεια του σχεδιασμού του Partitioner που θα χρησιμοποιήσουμε και σαφώς να έχουμε καλή γνώση των κλειδιών που αποθηκεύουμε. Εδώ αξίζει να αναφέρουμε το γεγονός ότι η χρήση των integer hash τιμών που θα χρησιμοποιήσουμε σαν κλειδιά μας δίνει το πλεονέκτημα της ισοκατανομής καθώς και της εύκολης επιλογής των partitions. Το θέμα αυτό θα αναλυθεί περισσότερο στην ενότητα που θα μελετήσουμε τον Partitoner του συστήματός μας.

Τέλος συμπεραίνουμε, ότι με χρήση της τεχνικής του BulkImport μπορούμε να επιτύχουμε καλύτερη απόδοση στη δημιουργία των πινάκων σε σχέση με την τεχνική του Api Import. Η τεχνική του BulkImport όμως, είναι σαφώς πιο περίπλοκη προγραμματιστικά και θα πρέπει να εξασφαλίσουμε την ολική ταξινόμηση και την ισοκατανομή των δεδομένων για να πετύχουμε την επιθυμητή απόδοση.

5.2.3) Map

Σε αυτή την ενότητα, θα παρουσιάσουμε τον κώδικα και την λειτουργία που εκτελείται κατά την map φάση της εισαγωγής των δεδομένων στους πίνακες. Αρχικά, πρέπει να καθορίσουμε τον τρόπο με τον οποίο θα εισάγουμε τα δεδομένα της βάσης μας. Η βάσεις που θέλουμε να εισάγουμε έχουν τη μορφή των RDF triple και είναι αποθηκευμένες

σε αρχεία κειμένου που σε κάθε γραμμή περιέχουν και μια διαφορετική RDF τριάδα. Για παράδειγμα, μερικές γραμμές από ένα τέτοιο αρχείο έχουν την παρακάτω μορφή.

<http://www.Department0.University0.edu/Lecturer0>
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#emailAddress>
"Lecturer0@Department0.University0.edu" .
<http://www.Department0.University0.edu/Lecturer0>
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#name> "Lecturer0" .
<http://www.Department0.University0.edu/Lecturer0>
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#worksFor>
<http://www.Department0.University0.edu> .

Όπως βλέπουμε ακολουθείται η URI αναφορά στα resources και τα literals αναπαριστώνται με ένα string μέσα σε “”. Κάθε γραμμή του αρχείου τελειώνει με τον χαρακτήρα “.” μετά τον οποίο έχουμε την αλλαγή γραμμής. Ο τρόπος με τον οποίο είναι αποθηκευμένα τα RDF δεδομένα μας, μας επιτρέπει την χρησιμοποίηση του TextInputFormat του Hadoop για εισαγωγή των δεδομένων. Το TextInputFormat παίρνει ώς είσοδο αρχεία κειμένου και για κάθε γραμμή τους παράγει και ένα ζεύγος key/value. Σαν κλειδί του ζεύγους χρησιμοποιείται το byte offset της γραμμής στο αρχείο κειμένου και σαν value έχουμε ολόκληρο το περιεχόμενο της γραμμής. Ο τρόπος, αυτός, ταιριάζει απόλυτα με τα αρχεία εισόδου μας και έτσι δεν είναι απαραίτητο να φτιάξουμε το δικό μας InputFormat.

Άρα με χρήση του συγκεκριμένου τρόπου εισόδου θα τρέχουμε μια συνάρτηση map για κάθε RDF triple στη βάση μας. Η συνάρτηση map παίρνει ως value την γραμμή του κειμένου που περιέχει το RDF triple. Αρχικά, λοιπόν, σπάμε τη γραμμή ώστε να προσδιορίσουμε τα strings που αντιστοιχούν στα subject, predicate και object αντίστοιχα. Κάθε ένα από αυτά τα string περνάει από τη συνάρτηση Jenkin's hash και λαμβάνουμε την τιμή του hash του, την οποία θα χρησιμοποιήσουμε για εισαγωγή στα index.

Σε αυτό το σημείο, θα πρέπει να θυμίσουμε ότι στόχος είναι η δημιουργία τεσσάρων πινάκων, ενός πίνακα που θα περιέχει τις αντιστοιχίες hash-URI και των 3 index. Ακόμα θα πρέπει να τονίσουμε ότι μετατρέπουμε τα κλειδιά σε bytes, με στόχο τη μείωση τόσο των ενδιάμεσων δεδομένων του MapReduce, όσο και τη μείωση των δεδομένων που αποθηκεύουμε στα index. Στα παρακάτω παραδείγματα θα χρησιμοποιούμε τιμές κλειδιών όπως s_hash(subject)_hash(predicate)_hash(object). Αυτό το κλειδί αποτελείται από 13 bytes. Το πρώτο byte κάθε κλειδιού δείχνει τον πίνακα στον οποίο αναφέρεται. Έχουμε 4 περιπτώσεις, που θα παρουσιάζονται με τα γράμματα n, s, p, o και θα κωδικοποιούνται με ένα byte. Το n αντιστοιχεί στον πίνακα με τους συνδυασμούς hash-URI και τον οποίο θα ονομάζουμε στο εξής names, το s στο index SP_O, το p στο index PO_S και το o στο index OS_P. Κάθε hash τιμή χρειάζεται 4 byte για την αποθήκευσή της, αφού είναι τύπου integer και άρα η ονομασία hash(subject) θα αντιστοιχεί στα 4 αυτά byte.

Για τη δημιουργία όλων των πινάκων χρειάζεται κάθε mapper να παράγει 6 ενδιάμεσα ζευγάρια key/value. Τα 3 πρώτα ζευγάρια θα χρησιμοποιούνται για τη δημιουργία του πίνακα names. Για κάθε ένα από τα resources subject, predicate και object παράγουμε ένα ζεύγος key/value με κλειδί της μορφής n_hash(subject)_subject και κενό value. Στη θέση του subject τοποθετούμε το πλήρες string που αντιστοιχεί στο συγκεκριμένο resource. Τα υπόλοιπα 3 ζευγάρια key/value, χρησιμοποιούνται για τη δημιουργία των 3 index. Στον παρακάτω πίνακα φαίνονται τα ζευγάρια key/value που αντιστοιχούν σε κάθε περίπτωση.

Πίνακας	Key	Value
names	n_hash(subject)_subject	-
names	n_hash(predicate)_predicate	-
names	n_hash(object)_object	-
SP_O	s_hash(subject)_hash(predicate)_hash(object)	-
PO_S	p_hash(predicate)_hash(object)_hash(subject)	-
OS_P	o_hash(object)_hash(subject)_hash(predicate)	-

Πίνακας 31: Map output

Βλέπουμε, ότι τα κλειδιά που θα χρησιμοποιηθούν έχουν ως πρώτο byte, ένα byte, που αναφέρεται στον πίνακα που θα δημιουργήσουν. Η χρήση αυτού του byte γίνεται με σκοπό τη εύκολη δημιουργία του Partitioner, ο οποίος θα πρέπει να οδηγήσει αυτά τα κλειδιά σε συγκεκριμένα partitons ανάλογα με τον πίνακα που θα δημιουργήσουν. Θα μπορούσαμε, ακόμα να παρατηρήσουμε το γεγονός ότι δεν χρησιμοποιούμε καθόλου τα πεδία value των ζευγαριών. Αυτό γίνεται λόγω του σχεδιασμού των πινάκων των index. Όπως έχουμε πει τα index μας θα έχουν ως rowid τα δυο resources και ως qualifier το τρίτο resource. Αυτό σημαίνει ότι χρειαζόμαστε ταξινόμηση για το συνολικό συνδυασμό subject-predicate-object και άρα δεν μπορούμε να τοποθετήσουμε κάποιο από αυτά στο value. Αν δημιουργούσαμε index της μορφής S_PO θα μπορούσαμε να τοποθετήσουμε το τελευταίο resource στο value.

Στη συνέχεια παραθέτουμε τον πλήρη κώδικα της συνάρτησης του map που υλοποιήσαμε.

```
public static class Map extends Mapper<LongWritable, Text, ImmutableBytesWritable, ImmutableBytesWritable> {
		private byte[] subject;
		private byte[] predicate;
		private byte[] object;
		private byte[] non;
		private ImmutableBytesWritable new_key = new ImmutableBytesWritable();
		private ImmutableBytesWritable v = new ImmutableBytesWritable();
		private static Hash h = JenkinsHash.getInstance();

		public void map(LongWritable key, Text value, Context context) throws IOException {
			non=Bytes.toBytes("");
			String line = value.toString();
			String s,p,o;
			StringTokenizer tokenizer = new StringTokenizer(line);

			s=tokenizer.nextToken(" ");
			if(s.contains("\"")){
				if(!s.endsWith("\""))
					s+= tokenizer.nextToken("\"")+"\"";
			}
			subject=Bytes.toBytes(s);
			p=tokenizer.nextToken(" ");
			if(p.contains("?")){
```

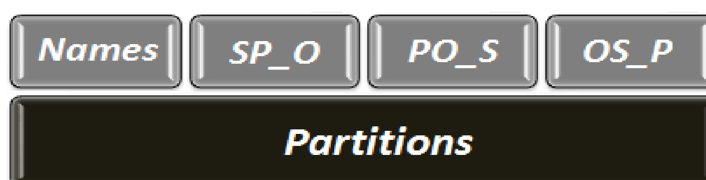
```
        if(!p.endsWith("\""))
                p+= tokenizer.nextToken("\"")+"\"";
}
predicate=Bytes.toBytes(p);
o=tokenizer.nextToken(" ");
if(o.contains("\"")){
        if(!o.endsWith("\""))
                o+= tokenizer.nextToken("\"")+"\"";
}
object=Bytes.toBytes(o);

try {

byte[] si = getHash(s);
byte[] pi = getHash(p);
byte[] oi = getHash(o);
//δημιουργία πίνακα names byte[0]=1
byte[] k = new byte[subject.length+4+1];
        k[0] = (byte)1;
for (int i = 0; i < 4; i++) {
        k[i+1]=si[i];
}
for (int i = 0; i < subject.length; i++) {
                k[i+4+1]=subject[i];
}
new_key.set(k, 0, k.length);
context.write(new_key,new ImmutableBytesWritable(non,0,0));
k = new byte[predicate.length+4+1];
k[0] = (byte)1;
for (int i = 0; i < 4; i++) {
        k[i+1]=pi[i];
}
for (int i = 0; i < predicate.length; i++) {
                k[i+4+1]=predicate[i];
}
new_key.set(k, 0, k.length);
context.write(new_key,new ImmutableBytesWritable(non,0,0));
k = new byte[object.length+4+1];
k[0] = (byte)1;
for (int i = 0; i < 4; i++) {
        k[i+1]=oi[i];
}
for (int i = 0; i < object.length; i++) {
                k[i+4+1]=object[i];
}
new_key.set(k, 0, k.length);
context.write(new_key,new ImmutableBytesWritable(non,0,0));

//δημιουργία index SP_O byte[0]=4
k = new byte[4+4+4+1];
k[0] = (byte)4;
for (int i = 0; i < 4; i++) {
        k[i+1]=si[i];
}
```

```
			for (int i = 0; i < 4; i++) {
				k[i+4+1]=pi[i];
			}
			for (int i = 0; i < 4; i++) {
				k[i+4+4+1]=oi[i];
			}
			new_key.set(k, 0, k.length);
			context.write(new_key, new ImmutableBytesWritable(non,0,0));

			//δημιουργία index OS_P byte[0]=2
			k = new byte[4+4+4+1];
			k[0] = (byte)2;
			for (int i = 0; i < 4; i++) {
				k[i+1]=oi[i];
			}
			for (int i = 0; i < 4; i++) {
				k[i+4+1]=si[i];
			}
			for (int i = 0; i < 4; i++) {
				k[i+4+4+1]=pi[i];
			}
			new_key.set(k, 0, k.length);
			context.write(new_key, new ImmutableBytesWritable(non,0,0));

			//δημιουργία index PO_S byte[0]=3
			k = new byte[4+4+4+1];
			k[0] = (byte)3;
			for (int i = 0; i < 4; i++) {
				k[i+1]=pi[i];
			}
			for (int i = 0; i < 4; i++) {
				k[i+4+1]=oi[i];
			}
			for (int i = 0; i < 4; i++) {
				k[i+4+4+1]=si[i];
			}
			new_key.set(k, 0, k.length);
			context.write(new_key, new ImmutableBytesWritable(non,0,0));

			} catch (InterruptedException e) {
				e.printStackTrace();
			}

		}

		private byte[] getHash(String string) {
			byte[] bst=Bytes.toBytes(string);
			Integer hashVal = Math.abs(h.hash(bst, bst.length, 0));
			byte[] b = Bytes.toBytes(hashVal);
			return b;
		}

	}
```

5.2.4) Partitioner

Όπως εξηγήσαμε, σε προηγούμενη ενότητα, για τη σωστή εκτέλεση της τεχνικής του BulkImport πρέπει να δημιουργήσουμε τον δικό μας Partitioner, ο οποίος αναλαμβάνει να δώσει ολική ταξινόμηση στα δεδομένα εισόδου της reduce φάσης. Ακόμα κατά τη δημιουργία του Partitioner πρέπει να φροντίζουμε ώστε να επιτυγχάνεται η απαραίτητη ισοκατανομή των δεδομένων στα partitions.

Ο Partitioner μας πρέπει να χειρίζεται αποτελεσματικά τα κλειδιά που παράγονται από τη φάση του map και να τα στέλνει κάθε φορά στο σωστό partition. Αρχικά, θα δημιουργήσουμε έναν Partitioner, ο οποίος δεν θα έχει γνώση των δεδομένων που θα περιέχονται στις βάσεις εισόδου. Στο παρακάτω σχήμα φαίνεται ο διαχωρισμός των partitions που θα χρησιμοποιηθεί.



Σχήμα 12: Basic Partitions

Όπως είπαμε στην προηγούμενη ενότητα, το πρώτο byte κάθε κλειδιού δηλώνει τον πίνακα στον οποίο αντιστοιχεί. Μπορούμε, λοιπόν, να χωρίσουμε τα διαθέσιμα partitions ανά πίνακα, όπως φαίνεται και στο σχήμα. Αρχικά, ελέγχουμε το πρώτο byte του κλειδιού και ανάλογα με την τιμή του, επιλέγουμε την αντίστοιχη ομάδα από partitions. Στη συνέχεια πρέπει να επιλέξουμε σε ποιό από τα partitions της συγκεκριμένης ομάδας θα τοποθετήσουμε το κλειδί. Μετά το πρώτο byte, που αντιστοιχεί στον πίνακα, ακολουθεί, σε όλους τους τύπους των κλειδιών, μια τιμή hash. Ανάλογα, λοιπόν, με αυτή την τιμή hash πρέπει να προσδιορίσουμε σε ποιό από τα partitions του πίνακα πρέπει να τοποθετηθεί το κλειδί. Η τιμή των hash είναι ένας θετικός ακέραιος αριθμός και άρα γνωρίζουμε ότι τα κλειδιά μας θα βρίσκονται στην περιοχή 0-MAX_INT. Ακόμα η χρήση των hash, ως κλειδιών, μας επιτρέπει να θεωρήσουμε ότι υπάρχει ισοκατανομή στα δεδομένα και άρα μπορούμε να ισομοιράσουμε την περιοχή 0-MAX_INT στα partitions που αντιστοιχούν στον κάθε πίνακα. Το μόνο που έχουμε να κάνουμε λοιπόν είναι να βρούμε σε ποιά περιοχή ανήκει το κάθε hash και να το στείλουμε στο αντίστοιχο partition. Με τον τρόπο, αυτό πετυχαίνουμε ολική ταξινόμηση και αρκετά καλή κατανομή των δεδομένων στα partitions.

Στην εργασία μας, όμως, γνωρίζουμε πλήρως τον τύπο των δεδομένων που θα χρησιμοποιήσουμε για τα πειράματά μας. Καλό είναι, λοιπόν, να δημιουργήσουμε έναν Partitioner ο οποίος λαμβάνει υπόψη του τα δεδομένα και πετυχαίνει καλύτερη ισοκατανομή των δεδομένων. Οι RDF βάσεις που θα χρησιμοποιήσουμε σαν είσοδο θα προέρχονται από το LUBM generator. Το LUBM είναι ένας κώδικας που δημιουργεί RDF datasets που αναφέρουνται σε στοιχεία πανεπιστημίων. Με χρήση αυτού του generator μπορούμε να παράγουμε όσο μεγάλα datasets επιθυμούμε για να τεστάρουμε το σύστημά μας. Σε αυτό το σημείο πρέπει να εξετάσουμε αν για τα συγκεκριμένα δεδομένα πετυχαίνουμε καλή ισοκατανομή στα partions.

Αρχικά για τον πίνακα names τα πρώτα hash του κλειδιού, τα οποία θα χρησιμοποιήσουμε για επιλογή του partition, προέρχονται από όλα τα διαφορετικά resourse και literals που υπάρχουν στη βάση μας. Αυτό μας εξασφαλίζει ότι θα υπάρχει μια μεγάλη ποικιλία από διαφορετικά hash και άρα θα έχουμε επαρκή ισοκατανομή.

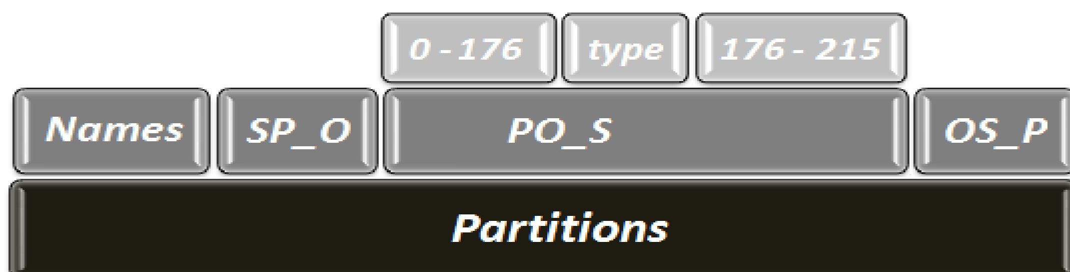
Ο πίνακας SP_O θα έχει ως πρώτο κλειδί τις τιμές των hash που αντιστοιχούν σε

όλα τα διαφορετικά subject της βάσης μας. Στα δεδομένα που παράγονται από το LUBM, υπάρχει μεγάλη ποικιλία από διαφορετικά subjects και άρα η ύπαρξη αυτής της ποικιλίας μας εξασφαλίζει μια επαρκή ισοκατανομή. Ακόμα δεν υπάρχει κάποιο συγκεκριμένο subject που να χρησιμοποιείται πολύ περισσότερες φορές από τα υπόλοιπα. Η ύπαρξη ενός τέτοιου subject θα οδηγούσε σε υπερφόρτωση του partition που το περιέχει. Οι ίδιες ιδιότητες ισχύουν και για τα object, άρα η ισοκατανομή εξασφαλίζεται επαρκώς, για τους πίνακες SP_O και OS_P, από τον αρχικό μας Partitioner.

Τα resources, όμως, που χρησιμοποιούνται ως predicates είναι πολύ περιορισμένα. Από τα στατιστικά στοιχεία του LUBM βλέπουμε ότι υπάρχουν μόνο 17 διαφορετικά predicates για όλη τα triples της βάσης. Ο μικρός αριθμός των διαφορετικών predicate δεν μπορεί να μας εξασφαλίσει καλή κατανομή. Ακόμα μπορούμε να δούμε ότι υπάρχουν κάποια από τα predicate αυτά που εμφανίζονται σε πολύ περισσότερα triples από ότι τα υπόλοιπα. Διαπιστώνουμε, λοιπόν, ότι στην περίπτωση του PO_S ο προηγούμενος Partitioner δεν μας παρέχει ικανοποιητική ισοκατανομή.

Ειδικότερα αν εξετάσουμε τα διαφορετικά predicates του LUBM βλέπουμε ότι υπάρχει μεγάλη χρησιμοποίηση του predicate rdf:type με το οποίο ορίζεται ο τύπος κάθε διαφορετικής οντότητας της βάσης μας. Όλα τα κλειδιά, που αντιστοιχούν στο συγκεκριμένο predicate, θα τοποθετούνται από τον Partitioner στο ίδιο partition και άρα ο συγκεκριμένος reducer θα έχει μεγάλη φόρτο εργασίας. Για να αποφύγουμε αυτό το φαινόμενο θα δημιουργήσουμε έναν καινούργιο Partitioner, ο οποίος θα χρησιμοποιείται για εισαγωγή των LUBM δεδομένων.

Ο νέος Partitioner θα έχει την ίδια λειτουργικότητα με τον παλιό σε ότι αφορά τους πίνακες names, SP_O και OS_P. Για την επεξεργασία των κλειδιών του πίνακα PO_S, χωρίζουμε τα partitions, που του αντιστοιχούν, σε τρία σύνολα. Η τιμή του κλειδιού, που αντιστοιχεί στο predicate rdf:type είναι 1768118995, άρα χωρίζουμε τα partitions του PO_S με βάση αυτό το κλειδί. Το πρώτο σύνολο περιέχει τα partitions που θα επεξεργαστούν τις τιμές των hash που είναι μικρότερες από την παραπάνω τιμή. Το δεύτερο σύνολο, επεξεργάζεται τα κλειδιά που περιέχουν την τιμή type, ενώ το τρίτο τα κλειδιά που είναι μεγαλύτερα. Μπορούμε να καθορίσουμε το μέγεθος των τριών αυτών συνόλων με βάση το φόρτο που έχει το καθένα. Στην περίπτωση που το predicate δεν είναι το rdf:type η κατανομή των κλειδιών στα δυο σύνολα γίνεται κανονικά με βάση το hash του predicate. Όταν το predicate του κλειδιού είναι το rdf:type λαμβάνουμε από αυτό και την τιμή του δεύτερου hash, το οποίο αντιστοιχεί στο object του triple. Όπως έχουμε αναφέρει, υπάρχουν πολλά διαφορετικά object και άρα μπορούμε να χρησιμοποιήσουμε την τιμή του hash αυτού, για κατανομή εντός του αντίστοιχου συνόλου από partitions. Έτσι, μπορούμε να πετύχουμε καλύτερη ισοκατανομή των δεδομένων.



Σχήμα 13: LUBM Partitions

Μια ακόμα βελτιστοποίηση, που μπορούμε να εφαρμόσουμε στον Partitioner, είναι η χρήση μεταβλητών για τον προσδιορισμό των partitions που αντιστοιχούν σε κάθε σύνολο. Ο αριθμός των partition ανά σύνολο, μπορεί να λαμβάνεται ως ποσοστό των συνολικών partitions. Μπορούμε, λοιπόν να πειραματιστούμε με τα επιμέρους ποσοστά των

συνόλων και να πετύχουμε την καλύτερη δυνατή κατανομή των δεδομένων.

Τέλος, ο Partitioner θα πρέπει να δημιουργεί τα offset των πινάκων που αναλύσαμε στο κεφάλαιο του σχεδιασμού. Το offset χρησιμοποιείται για τον αποδοτικό διαμοιρασμό των δεδομένων σε διαφορετικές γραμμές του πίνακα. Ο Partitioner μας διαθέτει μια σταθερά, MAX_HBASE_ROWS, η τιμή της οποίας θα ορίζει πόσες διαφορετικές γραμμές θα δημιουργούνται στον πίνακα, για τον συγκεκριμένο συνδυασμό κλειδιού. Τα συνολικά δεδομένα που αντιστοιχούν σε κάθε συνδυασμό κλειδιού πρέπει να μοιράζονται στις διαφορετικές αυτές γραμμές. Μια καλή τιμή για την σταθερά MAX_HBASE_ROWS πρέπει να είναι ακέραιο πολλαπλάσιο των κόμβων του cluster μας. Με μια τέτοια τιμή θα εξασφαλίσουμε ότι τα δεδομένα ενός query θα μοιράζονται εξίσου σε όλους τους κόμβους, αφού όλοι θα παίρνουν τον ίδιο αριθμό από γραμμές για επεξεργασία.

Σε αυτό το σημείο πρέπει να βρούμε έναν τρόπο να χωρίσουμε τα δεδομένα στις γραμμές μέσω του Partitioner. Ο πιο απλός και αποδοτικός τρόπος, για να το πετύχουμε, είναι να επιλέγουμε για κάθε κλειδί έναν τυχαίο αριθμό στο διάστημα 0-MAX_HBASE_ROWS και να τον τοποθετήσουμε μετά το δεύτερο hash του κλειδιού. Η τυχαιότητα της επιλογής είναι η ιδιότητα που μας εξασφαλίσει την καλή κατανομή των δεδομένων στις επιμέρους γραμμές. Για τα κλειδιά που αντιστοιχούν στους πίνακες names και OS_P δεν επιλέγουμε την τυχαία τιμή αφού οι γραμμές τους δεν περιέχουν μεγάλο αριθμό δεδομένων και άρα δεν πρέπει να χωρίζονται. Στον παρακάτω πίνακα βλέπουμε την μετατροπή των κλειδιών και με r συμβολίζουμε το byte που αντιστοιχεί στον τυχαίο αριθμό.

Old Key	**New Key**
n_hash(subject)_subject	Δεν αλλάζει
n_hash(predicate)_predicate	Δεν αλλάζει
n_hash(object)_object	Δεν αλλάζει
s_hash(subject)_hash(predicate)_hash(object)	s_hash(subject)_hash(predicate)_r_hash(object)
p_hash(predicate)_hash(object)_hash(subject)	p_hash(predicate)_hash(object)_r_hash(subject)
o_hash(object)_hash(subject)_hash(predicate)	Δεν αλλάζει

Πίνακας 32: Προσθήκη του offset

Ακολουθεί ο κώδικας του Partitioner που δημιουργήσαμε

```
public class MyNewTotalOrderPartitioner<VALUE> extends
Partitioner<ImmutableBytesWritable, VALUE>{
	private Random g1= new Random();

	public static final int MAX_HBASE_ROWS = 8;

	@Override
	public int getPartition(ImmutableBytesWritable key, VALUE value,
			int numPartitions) {
		int partition=0;
		byte[] k = key.get();
		byte[] k1 = new byte[4];
		for (int i = 0; i < 4; i++) {
			k1[i]=k[i+1];
		}
		byte pin = (byte) k[0]; //το byte που αντιστοιχεί στον πίνακα
		char[] kch = Bytes.toChars(k1);
		char[] s1 = new char[2];
```

```
for (int i = 0; i < 2 ; i++) {
                s1[i]=kch[i];
        }
        int id1 = Bytes.toInt(Bytes.toBytes(s1));//integer που αντιστοιχεί στο πρώτο
hash του key
        int id2; //integer που αντιστοιχει στο δεύτερο hash του key
        if(pin==(byte) 3){
                byte[] key2 = new byte[4];
                for (int i = 0; i < 4; i++) {
                        key2[i]=k[i+5];
                }
                kch = Bytes.toChars(key2);
        for (int i = 0; i < 2 ; i++) {
                        s1[i]=kch[i];
                }
                id2 = Bytes.toInt(Bytes.toBytes(s1));
        }
        //partition που αντιστοιχούν στους πίνακες εκτός του PO_S
        int def_prt= (int) Math.round(numPartitions*0.20);
        //partition που αντιστοιχούν στον πίνακα PO_S
        int pos_prt= numPartitions-(3*def_prt);

        if(pin==(byte) 1){//names
                p=0;
                int first = id1/10000000;
                float d= (float)215/(float)def_prt;
                partition = (int) Math.floor(first/d) +p;
        }
        else if(pin==(byte) 2){//OS_P
                p=def_prt;
                int first = id1/10000000;
                float d= (float)215/(float)def_prt;
                partition = (int) Math.floor(first/d) +p;
        }
        else if(pin==(byte) 3){//PO_S
                //επιλογή τυχαίου offset και προσθήκη του στο κλειδί
                Random g2 = new Random( g1.nextInt() );
                int i = g2.nextInt(MAX_HBASE_ROWS);
                byte[] k2 = new byte[k.length+1];
                for (int ii = 0; ii < 4+4+1; ii++) {
                        k2[ii]=k[ii];
                }
                k2[9]=(byte)i;
                for (int ii = 0; ii < 4; ii++) {
                        k2[10+ii]=k[9+ii];
                }
                key.set(k2,0,k2.length);

                p=2*def_prt;
                int no_type_part1 = (int) Math.round(pos_prt*0.20);
                int no_type_part2 = (int) Math.round(pos_prt*0.20);
                int type_part = pos_prt- no_type_part1- no_type_part2;
                int no_t_no1= 176;
                int no_t_no2= 40;
```

```
				int type=1768118995;
				else if(id1 <type){
					int first = id1/10000000;
					float d= (float)no_t_no1/(float)no_type_part1;
					partition = (int) Math.floor(first/d) +p;
				}
				else if(id1 == type){
					//επιλογή με βάση το δεύτερο hash id2
					int first = id2/10000000 ;
					float d= (float)215/(float)takes_part;
					partition = (int) Math.floor(first/d) +p + no_type_part1;
				}
				else if(id1>type){
					int first = id1/10000000 -176;
					float d= (float)no_t_no2/(float)no_type_part2;
					partition = (int) Math.floor(first/d) +p +pos_prt-
no_type_part2;
				}
			}
			else if(pin==(byte) 4){//SP_0
				//επιλογή τυχαίου offset και προσθήκη του στο κλειδί
				Random g2 = new Random( g1.nextInt() );
				int i = g2.nextInt(MAX_HBASE_ROWS);
				byte[] k2 = new byte[k.length+1];
				for (int ii = 0; ii < 4+4+1; ii++) {
					k2[ii]=k[ii];
				}
				k2[9]=(byte)i;
				for (int ii = 0; ii < 4; ii++) {
					k2[10+ii]=k[9+ii];
				}
				key.set(k2,0,k2.length);

				p=2*def_prt+pos_prt;
				int first = id1/10000000;
				float d= (float)215/(float)def_prt;
				partition = (int) Math.floor(first/d) +p;
			}
		}
		return partition;
}
```

5.2.5) Reducer

Μετά την επιλογή του κατάλληλου partition, κάθε reducer πρέπει να επεξεργαστεί ένα partition και να αποθηκεύσει τα δεδομένα του στον αντίστοιχο πίνακα. Πρέπει στον reducer, λοιπόν, να ορίσουμε τον τρόπο αντιμετώπισης κάθε διαφορετικού ζευγαριού key/value. Ο reducer για κάθε κλειδί εισόδου του θα δημιουργεί και ένα cell στον αντίστοιχο πίνακα HBase μέσω του HfileOutputFormat. Τα δεδομένα μας έχουν εξασφαλίσει την ολική ταξινόμηση και άρα μπορούμε άμεσα από κάθε κλειδί να δημιουργούμε την αντίστοιχη έξοδο. Όλες οι διαφορετικές περιπτώσεις φαίνονται στον παρακάτω πίνακα.

Reduce Key	rowid	Qualifier
n_hash(subject)_subject	n_hash(subject)	Subject
n_hash(predicate)_predicate	n_hash(predicate)	Predicate
n_hash(object)_object	n_hash(object)	Object
s_hash(subject)_hash(predicate)_r_hash(object)	s_hash(subject)_hash(predicate)_r	hash(object)
p_hash(predicate)_hash(object)_r_hash(subject)	p_hash(predicate)_hash(object)_r	hash(subject)
o_hash(object)_hash(subject)_hash(predicate)	o_hash(object)_hash(subject)	hash(predicate)

Πίνακας 33: Reduce output

Όταν επεξεργαζόμαστε στοιχεία του πίνακα names θα χρησιμοποιούμε σαν rowid την τιμή του hash και σαν value το πλήρες sting που αντιστοιχεί στο συγκεκριμένο resource. Εκτός όμως από την εισαγωγή των συνδυασμών στον πίνακα, θέλουμε να ελέγχουμε αν υπάρχουν δυο διαφορετικά resource που να αντιστοιχούν στο ίδιο hash. Αυτό όπως έχουμε αναφέρει έχει μια μικρή πιθανότητα να συμβεί αφού χρησιμοποιήσαμε το Jenkin's hash. Πρέπει, λοιπόν, στη φάση του reduce να ελέγχουμε για τέτοια σφάλματα και να ενημερώνουμε ώστε να γίνει η κατάλληλη αλλαγή της συνάρτησης hash και να χρησιμοποιηθεί κάποια άλλη όπως η md5. Αυτό μπορεί να επιτευχθεί χρησιμοποιώντας μια εξωτερική static μεταβλητή στην κλάση του reducer μας. Κάθε φορά που επεξεργαζόμαστε ένα ζευγάρι key/value μέσω της συνάρτησης reduce, αποθηκεύουμε την τιμή του hash στην μεταβλητή αυτή η οποία θα μπορεί να διαβαστεί σε επόμενη κλήση της συνάρτησης hash. Τα κλειδιά εισόδου του reducer είναι ταξινομημένα και άρα τα διαδοχικά hash θα είναι και αυτά ταξινομημένα. Αρκεί, λοιπόν, να ελέγχουμε κάθε φορά την τιμή της μεταβλητής και την τιμή του hash του κλειδιού. Αν οι δυο τιμές βρεθούν ίδιες θα έχουμε ένα σφάλμα. Τα κλειδιά που θα αντιστοιχούν σε resource με το ίδιο hash θα έχουν ταξινομηθεί σε διαδοχικές θέσεις στα ζευγάρια key/value και άρα μπορούμε να εξασφαλίσουμε πως αν ο έλεγχος, αυτός, δεν επιβεβαιωθεί ποτέ, δεν θα έχουμε κανένα σφάλμα στη βάση μας.

Όσον αφορά τους πίνακες των index, το μόνο που πρέπει να κάνουν οι reducers είναι να σπάσουν το κλειδί εισόδου σε δυο κομμάτια. Το πρώτο κομμάτι περιέχει τα byte μέχρι και το τυχαίο offset, ενώ το δεύτερο περιέχει την τελευταία τιμή hash. Ως rowid χρησιμοποιούμε το πρώτο κομμάτι και ως qualifier το δεύτερο και αυτό αρκεί για να δημιουργήσουμε τα index μας.

Παρακάτω φαίνεται ο αντίστοιχος κώδικας του Reducer

```
public static class Reduce extends Reducer<ImmutableBytesWritable,
ImmutableBytesWritable, ImmutableBytesWritable, KeyValue> {

        public void reduce(ImmutableBytesWritable key,
Iterable<ImmutableBytesWritable> values, Context context) throws IOException {

                byte[] k = key.get();
                byte pin = (byte) k[0]; //το byte που αντιστοιχεί στον πίνακα
                KeyValue emmitedValue = null;

                if(pin==(byte) 1){//names
                        byte[] k1 = new byte[4+1];
                        for (int i = 0; i < k1.length; i++) {
                                k1[i]=k[i];
```

```
				}
				byte[] k2 = new byte[k.length-5];
				for (int i = 0; i < k2.length; i++) {
					k2[i]=k[i+5];
				}
				ImmutableBytesWritable emmitedKey = new
						ImmutableBytesWritable(k1, 0, k1.length);


					emmitedValue = new KeyValue(emmitedKey.get(), Bytes
						.toBytes("A"), Bytes
						.toBytes("i"), k2);

			try {
					context.write(emmitedKey, emmitedValue);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
			else{
				if(pin==(byte) 2){//OS_P δεν έχουμε το επιπλέον byte του offset
					if(k.length!=13){
						System.exit(1);
					}
					byte[] newKey= new byte[9];
					for (int i = 0; i < newKey.length; i++) {
						newKey[i]=k[i];
					}
					ImmutableBytesWritable emmitedKey = new
					ImmutableBytesWritable(newKey, 0, newKey.length);
					byte[] val = new byte[4];
					for (int i = 0; i < val.length; i++) {
						val[i]=k[9+i];
					}
					emmitedValue = new KeyValue(emmitedKey.get(), Bytes
							.toBytes("A"), val, null);
					try {
					context.write(emmitedKey, emmitedValue);
					} catch (InterruptedException e) {
						e.printStackTrace();
					}
				}
				else{//SP_O, PO_S με offset
					if(k.length!=14){
						System.exit(1);
					}
					byte[] newKey= new byte[10];
					for (int i = 0; i < newKey.length; i++) {
						newKey[i]=k[i];
					}
					ImmutableBytesWritable emmitedKey = new
					ImmutableBytesWritable(newKey, 0, newKey.length);
					byte[] val = new byte[4];
```

```
                        for (int i = 0; i < val.length; i++) {
                              val[i]=k[10+i];
                        }
                        emmitedValue = new KeyValue(emmitedKey.get(), Bytes
                                    .toBytes("A"), val, null);
                        try {
                        context.write(emmitedKey, emmitedValue);
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                  }

            }

      }
```

5.2.6) Σύνδεση των Hfile με τους πίνακες

Αφού για τον reducer, χρησιμοποιήσαμε ως έξοδο το HfileOutputFormat τα αρχεία εξόδου μας θα έχουν τη μορφή των Hfile. Με αυτή, την τεχνική, δημιουργείται για κάθε reducer ένα ξεχωριστό Hfile που ανήκει σε διαφορετικό region του πίνακα. Για να ολοκληρωθεί η διαδικασία δημιουργίας των πινάκων πρέπει τα Hfile αυτά να συνδεθούν με την HBase και τον κατάλληλο πίνακα. Για το σκοπό, αυτό, υπάρχει ένα ειδικό αρχείο bat, που περιέχεται στο πακέτο της HBase και είναι υπεύθυνο για την φόρτωση των δεδομένων σε έναν ήδη υπάρχων πίνακα της HBase. Αρχικά, λοιπόν, δημιουργούμε τον πίνακά μας και στη συνέχεια, τρέχοντας μια απλή εντολή του φορτώνουμε τα Hfile που δημιουργήθηκαν από την εργασία MapReduce. Η εκτέλεση της εισαγωγής έχει σχεδιαστεί ώστε να είναι εξαιρετικά γρήγορη. Η παραπάνω διαδικασία μπορεί να γίνει και προγραμματιστικά μέσα από κώδικα Java, ο οποίος χρησιμοποιεί τις έτοιμες συναρτήσεις της HBase.

5.3) Εκτέλεση των join με χρήση MapReduce

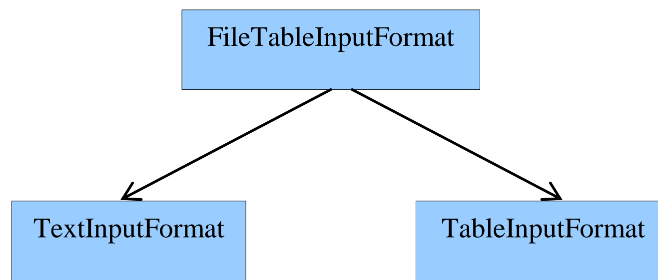
Σε αυτή την ενότητα θα μελετήσουμε όλα τα απαραίτητα συστήματα που δημιουργήσαμε για την εκτέλεση των join με χρήση του προγραμματιστικού πλαισίου του MapReduce. Η εκτέλση των join θα γίνεται σύμφωνα με το σχεδιασμό που παρουσιάστηκε στο προηγούμενο κεφάλαιο. Τα join θα εκτελούνται σύμφωνα με τον αλγόριθμο του PartialInputJoin που αναλύθηκε διεξοδικά. Τα δεδομένα εισόδου θα λαμβάνονται κατευθείαν από τους πίνακες των index της HBase και τα αποτελέσματα θα αποθηκεύονται με κατάλληλη μορφή σε αρχεία του HDFS.

5.3.1) InputFormat για εισαγωγή από πίνακες index

Βασικό, στοιχείο της δημιουργίας ενός τέτοιου συστήματος είναι ο προσδιορισμός του τρόπου με τον οποίο θα γίνεται η ανάκτηση των δεδομένων εισόδου από τους πίνακες των index. Το MapReduce χρησιμοποιεί συγκεκριμένες κλάσεις που χρησιμοποιούνται για την είσοδο των δεδομένων και ονομάζονται InputFormats. Αυτές οι κλάσεις προσφέρουν τις εξής λειτουργίες:

- Μια μέθοδο με την οποία ο κώδικας πελάτη να ορίζει τα δεδομένα που θα χρησιμοποιηθούν σαν είσοδος
- Χωρίζουν τα δεδομένα αυτά σε InputSplits καθένα από τα οποία θα αποτελέσει την είσοδο ενός mapper
- Ορίζει έναν RecordReader για την ανάγνωση των InputSplits και την δημιουργία των ζευγαριών key/value

Ένα MapReduce join έχει, δυο διαφορετικούς τύπους εισόδου. Μπορεί να πάρει δεδομένα από τα index που αποθηκεύονται σε πίνακες HBase ή δεδομένα από τα αρχεία εξόδου ενός άλλου MapReduce join. Αυτοί είναι δυο εντελώς διαφορετικοί τρόποι εισόδου που μερικές φορές χρειάζεται να χρησιμοποιηθούν και ταυτόχρονα. Πρέπει, λοιπόν, να δημιουργήσουμε έναν ενιαίο InputFormat που θα συνδυάζει τους δύο, αυτούς τύπους εισόδου. Γνωρίζουμε ότι για αρχεία κειμένου χρησιμοποιείται το TextInputFormat, ενώ για είσοδο από πίνακες HBase το TableInputFormat. Το InputFormat που θα δημιουργήσουμε, FileTableInputFormat, θα επεκτείνει και τα δυο αυτά InputFormat και θα χρησιμοποιεί κατάλληλα τις επιμέρους λειτουργίες τους.



Σχήμα 14: Input Format

Οι επιμέρους λειτουργίες των TextInputFormat και TableInputFormat δεν είναι όλες κατάλληλες και αποδοτικές για το σύστημα που θέλουμε να υλοποιήσουμε. Παρακάτω, θα εξετάσουμε τις επιμέρους, αυτές, λειτουργίες και όπου είναι απαραίτητο θα τις τροποποιούμε για να ταιριάζουν στο σύστημά μας.

5.3.1.1) Προσδιορισμός των δεδομένων εισόδου

Οι κλάσεις TextInputFormat και TableInputFormat προσφέρουν ικανοποιητικούς τρόπους με τους οποίους μπορούμε να προσδιορίσουμε τα δεδομένα εισόδου μας σε κάθε περίπτωση. Στο TextInputFormat μπορείς να δηλώσεις το όνομα του αρχείου εισόδου και στο TableInputFormat ο χρήστης προσδιορίζει ένα αντικείμενο scan που περιέχει όλα τα απαραίτητα στοιχεία. Οι δυο, αυτοί, τρόποι προσδιορισμού μπορούν να χρησιμοποιηθούν άμεσα από το FileTableInputFormat. Το μόνο που πρέπει να κάνουμε είναι, κάθε φορά που ο χρήστης στέλνει ένα αίτημα εισόδου στο FileTableInputFormat, να το προωθούμε στην αντίστοιχη λειτουργία του TextInputFormat ή του TableInputFormat.

5.3.1.2) Χωρισμός των δεδομένων σε Input Splits

Για να πετύχουμε καλή απόδοση στην είσοδο των δεδομένων του MapReduce, πρέπει να διαθέτουμε μια τεχνική, με την οποία θα χωρίζουμε τα δεδομένα εισόδου μας σε επιμέρους κομμάτια. Ο διαχωρισμός αυτός πρέπει να γίνεται χωρίς να είναι απαραίτητη η

ανάκτηση των δεδομένων. Για παράδειγμα, έστω ότι έχουμε ένα τεράστιο αρχείο κειμένου σαν είσοδο της εργασίας μας. Αυτό το αρχείο κειμένου βρίσκεται αποθηκευμένο στο HDFS και άρα είναι χωρισμένο σε block που έχουν ένα συγκεκριμένο μέγιστο μέγεθος. Ένας αποδοτικός τρόπος χωρισμού αυτού του αρχείου και κατανομής του στους mappers είναι, ο χωρισμός του στα block του HDFS. Κάθε mapper λαμβάνει ως είσοδο ένα block του HDFS και το επεξεργάζεται τοπικά. Ο διαμοιρασμός, αυτός είναι εξαιρετικά αποδοτικός αφού δεν χρειάζεται καμία επεξεργασία των πραγματικών δεδομένων για την επίτευξή του.

Τα κομμάτια στα οποία χωρίζουμε την είσοδο ονομάζονται InputSplits και πρέπει να προσδιορίζονται επαρκώς μέσα από ένα αντικείμενο που μπορεί να σταλεί μέσω του δικτύου. Για παράδειγμα τα block του αρχείου κειμένου προσδιορίζονται από το id του μπλοκ, το όνομα του αρχείου, τα byte offset των δεδομένων του κτλ. Κατά την εκκίνηση της εργασίας, ο master κόμβος, φτιάχνει τα αντικείμενα των InputSplits και στη συνέχεια τα στέλνει στους εργάτες, ένα για κάθε mapper. Το InputSplit δεν περιέχει τα πραγματικά δεδομένα αλλά μια περιγραφή για το που μπορεί ο slave να τα βρει.

Τα δεδομένα εισόδου ενός MapReduce join είναι δεδομένα που αντιστοιχούν σε συγκεκριμένα BGP patterns. Τα δεδομένα, αυτά, αποτελούνται από ένα σύνολο γραμμών των πινάκων της HBase, που περιέχουν τα index. Πρέπει, λοιπόν, να βρούμε έναν αποδοτικό τρόπο για τον χωρισμό ενός τέτοιου συνόλου γραμμών. Στη συνέχεια θα εξετάσουμε πως μπορούμε να χωρίσουμε αποδοτικά την είσοδο σε όλες τις διαφορετικές περιπτώσεις BGP ερωτημάτων.

Οι τέσσερες διαφορετικές περιπτώσεις εισόδου είναι η είσοδος με μια, δυο ή τρεις μεταβλητές από τα index και η είσοδος από ένα αρχείου εξόδου του join. Πρώτα, θα εξετάσουμε την περίπτωση που τα δεδομένα εισόδου προέρχονται από το αποτέλεσμα ενός άλλου, προηγούμενου, join. Τα αποτελέσματα ενός MapReduce join γράφονται σε αρχεία κειμένου που αποθηκεύονται στο HDFS. Κάθε γραμμή των αρχείων αυτών αντιστοιχεί σε ένα σύνολο από binding και μπορεί να χρησιμοποιηθεί άμεσα ως είσοδος σε ένα MapReduce join. Μπορούμε, λοιπόν, να χρησιμοποιήσουμε το TextInputFormat και να καταφέρουμε εύκολα να ανακτήσουμε τα δεδομένα εισόδου.

Στις υπόλοιπες περιπτώσεις η είσοδος γίνεται από τους πίνακες της HBase. Για εισαγωγή από πίνακες HBase, υπάρχει ένα έτοιμο InputFormat που ονομάζεται TableInputFormat. Το TableInputFormat παίρνει, ως είσοδο, ένα σύνολο από γραμμές της HBase και τις χωρίζει στα διαφορετικά regions. Όπως καταλαβαίνουμε δεν μπορούμε να χρησιμοποιήσουμε έναν τέτοιο τρόπο εισόδου αφού σε κάποιες περιπτώσεις η είσοδός μας αποτελείται από μια μόνο γραμμή που πρέπει να χωριστεί. Ακόμα και όταν έχουμε ένα σύνολο γραμμών, ο χωρισμός τους στα διαφορετικά regions δεν θα μας αρκούσε. Αυτό συμβαίνει επειδή φορτώσαμε τα δεδομένα μέσω του BulkImport. Σύμφωνα με τον αλγόριθμο, που χρησιμοποιήσαμε, τα rowid με την ίδια πρώτη τιμή hash, τοποθετούνται από τον Partitioner στο ίδιο partition και άρα γράφονται στο ίδιο region της HBase. Στην περίπτωση, λοιπόν, που θέλουμε να ανακτήσουμε τα δεδομένα διαθέτοντας μόνο την πρώτη τιμή hash, το σύνολο των γραμμών μας ανήκει στο ίδιο region και άρα δεν χωρίζεται. Άρα πρέπει να δημιουργήσουμε το δικό μας InputFormat για την εισαγωγή από τα index.

Η πρώτη περίπτωση είναι να έχουμε ένα BGP query με μια μεταβλητή και δυο σταθερές. Σε αυτή την περίπτωση, ο χωρισμός γίνεται με βάση το offset που έχει η κάθε γραμμή. Θα υπάρχουν το πολύ MAX_HBASE_ROWS γραμμές, στον πίνακα, που αντιστοιχούν στο συγκεκριμένο query. Όπως είπαμε, φροντίζουμε το MAX_HBASE_ROWS να είναι ακέραιο πολλαπλάσιο του αριθμού των κόμβων του δικτύου μας. Το μόνο που έχουμε να κάνουμε σε αυτή την περίπτωση είναι να μοιράσουμε τις διαφορετικές γραμμές στους κόμβους μας. Κάθε κόμβος παίρνει ν γραμμές, όπου ν το αποτέλεσμα της διαίρεσης του MAX_HBASE_ROWS με τον αριθμό των κόμβων. Σαν InputSplit μπορούμε να χρησιμοποιήσουμε ένα scan με startRow και stopRow που

αντιστοιχούν στο σύνολο αυτών των γραμμών.

Θα εξετάσουμε, τώρα, την περίπτωση ενός BGP query με δυο μεταβλητές και μια σταθερά. Τα rowid των γραμμών της HBase αποτελούνται από δυο τιμές hash. Σε αυτή την περίπτωση, γνωρίζουμε την πρώτη τιμή hash και ως είσοδο έχουμε όλες τις γραμμές του πίνακα που έχουν αυτή, ως πρώτη τιμή. Οι γραμμές, αυτές, θα είναι αποθηκευμένες διαδοχικά στον πίνακα, λόγω της λεξικογραφικής ταξινόμησης. Στα rowid μετά την πρώτη τιμή hash ακολουθεί η δεύτερη τιμή, η οποία είναι ένας θετικός ακέραιος. Άρα, βλέπουμε ότι γνωρίζουμε το σύνολο στο οποίο θα ανήκουν οι γραμμές εισόδου μας. Χωρίζουμε, λοιπόν το διάστημα των [0, MAX_INT] σε r σύνολα, όπου r ο αριθμός των κόμβων του cluster μας. Κάθε κόμβος επεξεργάζεται ένα από αυτά τα σύνολα. Και σε αυτή την περίπτωση μπορούμε να χρησιμοποιήσουμε σαν InputSplit ένα scan με startRow και stopRow που αντιστοιχούν στο σύνολο αυτών των γραμμών.

Τέλος έχουμε την περίπτωση ενός BGP query με τρεις μεταβλητές και καμία σταθερά. Εδώ, ως είσοδος πρέπει να χρησιμοποιηθεί όλη η βάση μας και άρα θα διαβαστεί ένας ολόκληρος πίνακας. Ο καλύτερος τρόπος χωρισμού, για έναν ολόκληρο πίνακα, θα ήταν ο χωρισμός του στα διαφορετικά regions. Αυτό γίνεται, εύκολα, με χρήση του TableInputFormat. Τα InputSplits που χρησιμοποιούνται και εδώ βασίζονται σε ένα scan με startRow και stopRow.

Αφού εξετάσαμε όλες τις περιπτώσεις είδαμε ότι δεν χρειάζεται να τροποποιήσουμε καθόλου τις λειτουργίες του TextInputFormat. Όσον αφορά το TableInputFormat πρέπει να μετατρέψουμε την μέθοδο getSplits ώστε σε κάθε περίπτωση να εκτελεί τον χωρισμό που προσδιορίστηκε παραπάνω. Τέλος είδαμε ότι, για την υλοποίηση του συστήματός μας, χρειαζόμαστε το συνδιασμό 2 διαφορετικών InputSplits. Το πρώτο είναι το FileInputSplit που χρησιμοποιείται για είσοδο από αρχεία και το δεύτερο το TableInputSplit που χρησιμοποιείται για είσοδο από πίνακες HBase. Το πρόβλημα σε αυτή την περίπτωση είναι ότι πρέπει σαν InputSplit να χρησιμοποιούμε αποκλειστικά ένα κοινό αντικείμενο. Για να ξεπεράσουμε αυτό το εμπόδιο θα δημιουργήσουμε ένα νέο InputSplit το FileTableInputSplit το οποίο θα είναι απλά ένα interface, το οποίο θα γίνεται implement και από τα 2 επιθυμητά InputSplit. Το interface του FileTableInputSplit θα περιέχει εκτός των άλλων μια μεταβλητή, η οποία θα τίθεται στην κατάλληλη τιμή από τα FileSplit και TableSplit. Όταν ο mapper, λαμβάνει ένα FileTableInputSplit θα κοιτάει την τιμή της μεταβλητής και ανάλογα θα αναγνωρίζει το split ως FileSplit ή TableSplit. Στο FileTableInputSplit θα τοποθετούμε και επιπλέον πληροφορίες που αφορούν το id του join που εκτελούμε, το όνομα της joining variable και άλλες απαραίτητες πληροφορίες για την δημιουργία των ζευγαριών key/value εισόδου.

5.3.1.3) RecordReader

Ο RecordReader είναι μια κλάση, η οποία είναι υπεύθυνη για την ανάγνωση του InputSplit και την μετατροπή των δεδομένων σε ζευγάρια key/value που θα χρησιμοποιηθούν για είσοδο στους mappers. Τα InputFormat FileSplit και TableSplit παρέχουν τους δικούς τους, διαφορετικούς, RecordReaders. Αρχικά πρέπει να δούμε πως θα γίνεται η επιλογή του κατάλληλου RecordReader. Για τον προσδιορισμό της εργασίας, ο κόμβος εργάτης λαμβάνει μόνο το αντικείμενο του FileTableInputSplit. Διαβάζοντας την τιμή του πεδίου που αντιστοιχεί στον τύπο του split, ο εργάτης, μπορεί να προσδιορίσει τον τύπο του split και να δημιουργήσει τον κατάλληλο RecordReader.

Στην πρώτη περίπτωση θα εξετάσουμε την είσοδο από αρχεία κειμένου. Τα αρχεία κειμένου που θα έχουμε, ως είσοδο, θα προέρχονται από κάποιο άλλο join και άρα θα έχουν κατάλληλη μορφή για άμεση επεξεργασία τους από τον LineRecordReader. Η κάθε γραμμή

των αρχείων, αυτών, θα μπορεί να χρησιμοποιηθεί άμεσα ως value για είσοδο στον mapper και άρα δεν χρειάζεται καμία αλλαγή στον RecordReader.

Στην δεύτερη περίπτωση έχουμε την ανάγνωση δεδομένων από πίνακα HBase μέσω του TableRecordReader. O TableRecordReader διαβάζει με τη σειρά τα cell που ανήκουν στο split και για κάθε cell δημιουργεί ένα ζευγάρι key/value. Το ζευγάρι αυτό περιέχει ως key το συνδυασμό rowid, column_family, qualifier, timestamp και ως value την τιμή του αντίστοιχου cell. Στο κεφάλαιο του σχεδιασμού αναφέραμε ότι τα ζευγάρια key/value, που θα χρησιμοποιήσουμε, θα έχουν τα bindings στο πεδίο value, η μορφή του οποίου θα ακολουθεί τον παρακάτω γενικό κανόνα:

jpat var1$$bindings var2$$bindings .... varN$$bindings

όπου var1...N είναι οι διαφορετικές μεταβλητές που συνδέονται με το συγκεκριμένο join. Στη θέση του bindings τοποθετούμε μια ή περισσότερες τιμές της αντίστοιχης μεταβλητής. Το jpat είναι ένα id μοναδικό για κάθε BGP pattern ή έξοδο join, το οποίο μας βοηθά να αναγνωρίσουμε την προέλευση του κάθε ζεύγους key/value.

Διαπιστώνουμε, λοιπόν, ότι δεν μπορούμε να χρησιμοποιήσουμε τον TableRecordReader και θα πρέπει να δημιουργήσουμε τον δικό μας RecordReader. Για τη δημιουργία του βασιστήκαμε στον κώδικα του TableRecordReader και απλά αλλάξαμε τη μορφή του ζευγαριού key/value που δημιουργείται κάθε φορά. Κρατήσαμε ίδιο τον τρόπο προσπέλασης των δεδομένων του split που χρησιμοποιείται στον TableRecordReader. Για κάθε cell, αυτή τη φορά, δημιουργούμε ένα ζεύγος key/value με κενή τιμή κλειδιού και value που ακολουθεί την παραπάνω μορφή. Τις τιμές των μεταβλητών και του jpat μπορούμε να τις βρούμε από τις επιπλέον πληροφορίες που περιέχονται στο FileTableInputSplit.

Με τον παραπάνω, απλό, RecordReader μπορούμε να δημιουργήσουμε σωστά ζευγάρια εισόδου αλλά έχουμε το πρόβλημα ότι δεν ομαδοποιούμε τα bindings. Κάθε ζευγάρι εισόδου προέρχεται από ένα cell και άρα περιέχει μόνο έναν δυνατό συνδυασμό από bindings. Με μια μικρή αλλαγή μπορούμε να παρέχουμε και ομαδοποίηση των bindings εισόδου. Κρατάμε απλά έναν μετρητή των cell που διαβάζουμε. Κάθε φορά που διαβάζουμε ένα καινούργιο cell, αυξάνουμε τον μετρητή και τοποθετούμε το binding στο value. Όταν ο μετρητής φτάσει την τιμή ομαδοποίησης που έχουμε ορίσει δημιουργείται το ζευγάρι key/value και η τιμή του value γίνεται κενή.

Με τη χρήση του μετρητή μπορούμε να έχουμε άμεσο έλεγχο στο βαθμό ομαδοποίησης των bindings εισόδου και άρα να πετύχουμε την καλύτερη δυνατή απόδοση. Η χρήση bindings μικρής ομαδοποίησης αυξάνει τα ζευγάρια εισόδου καθώς και τον συνολικό όγκο εισόδου αφού αποθηκεύουμε πολλαπλές φορές σταθερά στοιχεία όπως το jpat και τα ονόματα των μεταβλητών. Από την άλλη, η χρήση μεγάλης ομαδοποίησης οδηγεί σε τεράστια σε μέγεθος bindng, που στη συνέχεια είναι δύσκολο να επεξεργαστούν και να μεταφερθούν μέσω δικτύου στους reducers. Η επιλογή του σωστού βαθμού ομαδοποίησης είναι και αυτή μια σημαντική παράμετρος του συστήματος που χρειάζεται προσεκτική προσαρμογή.

5.3.2) Πέρασμα παραμέτρων στους mappers και reducers

Για την εκτέλεση του αλγορίθμου του PartialInputJoin οι mappers και οι reducers θα πρέπει να γνωρίζουν πολλές πληροφορίες για τις επιλογές του join. Τέτοιες πληροφορίες είναι:

- Το id του join, για την τοποθέτησή του στα δεδομένα εξόδου.

- Οι μεταβλητές σύμφωνα με τις οποίες εκτελείται το join (joining variables).
- Οι mappers πρέπει να έχουν μια αντιστοιχία μεταξύ του id της εισόδου και του joining variable. Κάθε αρχείο εισόδου γίνεται join μόνο ως προς μια μεταβλητή, άρα για κάθε id πρέπει να γνωρίζουμε το αντίστοιχο joining variable.
- Οι reducers πρέπει να γνωρίζουν τον αριθμό των query εισόδου καθώς και των αριθμό των υπόλοιπων query για τα οποία δεν έχουμε είσοδο. Οι αριθμοί, αυτοί, είναι απολύτως απαραίτητοι για την εκτέλεση του join. Μάλιστα, οι αριθμοί, αυτοί, διαφέρουν για κάθε joining variable και άρα πρέπει και εδώ να γνωρίζουμε τη σωστή αντιστοιχία.
- Οι reducers χρειάζονται τα BGP query τα οποία δεν χρησιμοποιούνται ως είσοδο. Αυτά χρειάζονται ώστε να ξέρουμε τι πρέπει να ψάξουμε κάθε φορά στα index της HBase.
- Ο reducer χρειάζεται να γνωρίζει αν το join είναι το τελικό join για το SPARQL ερώτημα. Αν το join είναι τελικό η έξοδός του πρέπει να έχει μορφή κατανοητή για τον άνθρωπο, δηλαδή να μην είναι σε μορφή κατάλληλη για πέρασμα σε επόμενο join. Στο τελικό join όλες οι τιμές των hash αντικαθιστούνται με τα URI των resources και τα binding δεν παρουσιάζονται στην ομαδοποιημένη μορφή τους.

Όλες αυτές οι πληροφορίες είναι άκρως απαραίτητες για την εκτέλεση του join και πρέπει με κάποιον τρόπο να είναι διαθέσιμες σε όλους τους κόμβους του δικτύου μας. Οι παραπάνω πληροφορίες γίνονται γνωστές κατά τη διάρκεια επιλογής της σειράς των join από τον planner. Ο planner, όμως, εκτελείται κεντρικά στον master κόμβο. Για να περάσουμε τις απαραίτητες πληροφορίες σε όλους τους κόμβους επιλέγουμε να τις αποθηκεύσουμε σε ένα αρχείο κειμένου του HDFS με όνομα Input. Το αρχείο αυτό ανανεώνεται με τις κατάλληλες πληροφορίες πριν την έναρξη της εκτέλεσης της εργασίας MapReduce. Οι mappers και οι reducers, έχουν μια συνάρτηση setup η οποία εκτελείται μια φορά για κάθε ξεχωριστό mapper και reducer. Στη συνάρτηση setup διαβάζουμε τα δεδομένα του αρχείου και τα αποθηκεύουμε στις κατάλληλες μεταβλητές, ώστε να τα έχουμε έτοιμα κατά την εκτέλεση των συναρτήσεων map και reduce.

5.3.3) Mapper

Σε αυτή την ενότητα θα αναλύσουμε την λειτουργικότητα της συνάρτησης του mapper. Αρχικά, όπως είπαμε, κατά την αρχικοποίησή του ο mapper διαβάζει, από το HDFS, το αρχείο Input και κρατάει σε μεταβλητές όλες τια απαραίτητες πληροφορίες. Ο mapper διαβάζει το String που περιέχεται στο value του ζευγαριού και βλέπει το id του. Ανάλογα με το id βρίσκει ποιά είναι η joining variable, ως προς την οποία πρέπει να κάνει join τα binding του value. Στη συνέχεια ψάχνει το value μέχρι να εντοπίσει τη joining variable. Όταν την βρει παίρνει τη λίστα με τα bindings που την ακολουθούν και για κάθε ένα από αυτά παράγει ένα ζεύγος key/value. Αυτό το ζεύγος περιέχει ως κλειδί την τιμή του joining variable και ως value τα bindings όλων των υπόλοιπων μεταβλητών που βρίσκονταν στο value εισόδου. Η αναλυτική μορφή του ζευγαριού key/value παρουσιάστηκε στο αντίστοιχο κεφάλαιο του σχεδιασμού.

5.3.4) Reducer

Εδώ θα εξετάσουμε τις λεπτομέρειες υλοποίησης της συνάρτησης του reducer που αντιστοιχεί στον αλγόριθμο του PartialInputJoin. Τα ζεύγη key/value που παράγονται από τους mappers ταξινομούνται και ομαδοποιούνται με βάση το κλειδί τους, από το πλαίσιο του MapReduce. Έτσι οι reducers λαμβάνουν ως είσοδο για κάθε κλειδί, μια λίστα με τα

values που αντιστοιχούν σε αυτό. Αρχικά, ο reducer διαβάζει την τιμή που περιέχεται στο κλειδί του ζευγαριού που επεξεργάζεται. Το κλειδί έχει την μορφή jvar$$hash και από αυτό βλέπουμε εύκολα σε ποίο joining variable αντιστοιχεί. Χρησιμοποιώντας τις πληροφορίες που περιέχονται στο αρχείο Input βρίσκουμε τον αριθμό των query εισόδου (inq), τον αριθμό των υπόλοιπων query (otherq) καθώς και την πλήρη περιγραφή των υπολοίπων, αυτών, query. Διαθέτοντας τις παραπάνω πληροφορίες είμαστε έτοιμοι να ξεκινήσουμε τη διαδικασία εκτέλεσης του join. Η διαδικασία του join γίνεται σε δυο φάσεις.

Η πρώτη φάση εκτέλεσης του join είναι οι εκτέλεση του join των query εισόδου. Σε αυτή τη φάση εξετάζουμε τη λίστα με τα values και μετράμε τα διαφορετικά id που περιέχονται σε αυτή. Αν ο αριθμός των διαφορετικών id είναι ίδιος με τον αριθμό των BGP εισόδου (inq), ο reducer προχωράει στη φάση της αναζήτησης στην HBase. Σε περίπτωση που ο αριθμός είναι μικρότερος τα συγκεκριμένα δεδομένα αγνοούνται και δεν παράγεται ζεύγος εξόδου.

Στη δεύτερη φάση πρέπει να εκτελέσουμε το join ως προς τα υπόλοιπα BGP query για τα οποία δεν είχαμε είσοδο. Για κάθε key, που πέρασε την πρώτη φάση, οι reducers γνωρίζουν ποιά είναι τα συγκεκριμένα query που πρέπει να ικανοποιεί. Ψάχνουν, λοιπόν, στο κατάλληλο index χρησιμοποιώντας και το hash που περιέχεται στο συγκεκριμένο κλειδί. Αν υπάρχει αντίστοιχα δεδομένα στο index τα προσθέτουμε στα binding του value και συνεχίζουμε με το επόμενο query. Σε περίπτωση που το index δεν διαθέτει τέτοια εγγραφή, οι reducers δεν δημιουργούν καμία έξοδο και προχωράνε στο επόμενο ζευγάρι key/value. Αν όλα τα otherq queries ικανοποιούνται τότε πρέπει να δημιουργήσουμε ένα ζευγάρι εξόδου.

Ο reducer κατά τη διαδικασία εκτέλεσης του join ανανεώνει συνεχώς το σύνολο των binding και άρα στο τέλος το διαθέτει όλα τα binding που πρέπει να παρουσιάσει στην έξοδο. Εδώ υπάρχουν δυο διαφορετικές περιπτώσεις. Η πρώτη περίπτωση είναι το join να μην είναι τελικό και τα δεδομένα εξόδου του να πρέπει να χρησιμοποιηθούν από κάποιο άλλο join. Σε αυτή την περίπτωση η έξοδός μας πρέπει να έχει τη μορφή

jpat var1$$bindings var2$$bindings .... varN$$bindings

Τα δεδομένα μας χρειάζονται πολύ μικρές αλλαγές για να εκφραστούν σε αυτή τη μορφή άρα ο reducer είναι εύκολο να δημιουργήσει την έξοδο.

Η άλλη περίπτωση εξόδου παρουσιάζεται όταν το join μας είναι τελικό, δηλαδή τα δεδομένα εξόδου του πρέπει να είναι αναγνώσιμα από τον άνθρωπο. Αρχικά, λοιπόν, παίρνουμε το σύνολο των bindings και μεταφράζουμε όλα τα hash που περιέχει, στα αντίστοιχα URI των resources. Τα bindings είναι πλέον σε μορφή κατανοητή από τον άνθρωπο, όμως αυτή δεν είναι και η συνηθισμένη μορφή απάντησης ενός SPARQL query. Πρέπει τώρα να βρούμε όλους τους διαφορετικούς συνδυασμού των bindings που περιέχονται στο σύνολό μας και για κάθε έναν να δημιουργούμε ένα ζευγάρι εξόδου. Αυτό είναι σχετικά απλό και μπορεί να υλοποιηθεί με απλούς βρόχους for. Βλέπουμε ότι η περίπτωση του τελικού join είναι σαφώς πιο απαιτητική σε χρόνο αφού έχουμε επιπλέον προσβάσεις στην HBase και προσδιορισμό όλων των δυνατών συνδυασμών. Αν ο βαθμός ομαδοποίησης που χρησιμοποιούμε είναι αρκετά μεγάλος η απόδοση του join θα είναι μειωμένη αφού έχουμε εκθετική αύξηση των διαφορετικών συνδυασμών που πρέπει να βρούμε. Βλέπουμε και πάλι ότι η σωστή επιλογή του βαθμού ομαδοποίησης είναι εξαιρετικά σημαντική για την βελτίωση της απόδοσης.

5.4) Επεξεργασία SPARQL ερωτημάτων

Για να χρησιμοποιήσουμε το σύστημα εκτέλεσης των join που δημιουργήσαμε, πρέπει να έχουμε ένα σύστημα επεξεργασίας των SPARQL ερωτημάτων και προσδιορισμού των join που πρέπει να εκτελεστούν. Η επεξεργασία ενός SPARQL ερωτήματος απαιτεί:

- τον έλεγχο συντακτικής ορθότητας του ερωτήματος
- την ανάγνωση του ερωτήματος και αποθήκευσή του σε κατάλληλες δομές
- την επεξεργασία κάθε επιμέρους query πριν τη διενέργεια των join
- τη δημιουργία ενός πλάνου εκτέλεσης των join
- την εκτέλεση των join σύμφωνα με το πλάνο

5.4.1) Jena

Το πιο ολοκληρωμένο πακέτο λογισμικού που έχει δημιουργηθεί για την εκτέλεση SPARQL ερωτημάτων είναι το πακέτο ελεύθερου κώδικα Jena [11]. Δημιουργήθηκε από την εταιρία HP και στη συνέχεια έγινε πακέτο ελεύθερου κώδικα, το οποίο είναι επεκτάσιμο και διαθέτει πολύ καλή τεκμηρίωση για την διευκόλυνση των προγραμματιστών. Η Jena παρέχει ένα προγραμματιστικό περιβάλλον το οποίο υλοποιεί τις τεχνολογίες RDF, RDFS, OWL και SPARQL. Παρέχει επίσης το πακέτο ARQ που έχει δημιουργηθεί για την αποθήκευση και επερώτηση RDF δεδομένων. Το ARQ, όμως, έχει σχεδιαστεί για δεδομένα μικρού μεγέθους καθώς διατηρεί τις απαραίτητες δομές δεδομένων στην μνήμη και όχι στο δίσκο. Αυτό σημαίνει ότι δεν μπορεί να επεξεργαστεί RDF βάσεις που είναι αρκετά μεγάλες ώστε να μη χωράνε στη μνήμη. Βασικό πλεονέκτημα της Jena είναι το γεγονός ότι μπορεί ο προγραμματιστής να δημιουργήσει το δικό του υποσύστημα (π.χ. για αποθήκευση των RDF δεδομένων) και να το συνδέσει με το σύστημα της Jena, δημιουργώντας έτσι ένα ολοκληρωμένο σύστημα που παρέχει τις νέες δυνατότητες.

Θα χρησιμοποιήσουμε, λοιπόν, το προγραμματιστικό πακέτο της jena για να υλοποιήσουμε εύκολα όλες τις λειτουργίες που απαιτούνται για την εκτέλεση ενός SPARQL ερωτήματος. Αρχικά η jena διαθέτει ένα αξιόπιστο σύστημα για τον έλεγχο της ορθότητας των SPARQL ερωτημάτων. Στη συνέχεια αποθηκεύει το ερωτήματα σε δενδρικές δομές που είναι κατάλληλες για την επεξεργασία και την απάντησή τους. Τα δυο αυτά συστήματα, της jena, θα τα χρησιμοποιήσουμε χωρίς καμία αλλαγή ώστε να ενσωματώσουμε εύκολα τις λειτουργίες τους, στο σύστημά μας.

Υπάρχουν πολλές υλοποιήσεις συστημάτων επεξεργασίας των ερωτημάτων SPARQL στο πακέτο της jena. Κάθε μια από αυτές παίρνει ως είσοδο την δενδρική δομή, στην οποία έχει αποθηκευτεί το SPARQL ερώτημα και την επεξεργάζεται κατάλληλα για τον προσδιορισμό της αντίστοιχης απάντησης. Ο βασικός τρόπος προσπέλασης και επεξεργασία της δενδρικής δομής του query είναι με χρήση ενός visitor. Η jena έχει υλοποιήσει τις βασικές λειτουργίες για προσπέλαση της δομής και προσφέρει μια διαπροσωπία, στη οποία ο προγραμματιστής μπορεί να ορίσει τι θέλει να εκτελείται σε κάθε βασική οντότητα του query. Για παράδειγμα, τα φύλλα του δέντρου, του SPARQL query, αποτελούνται από BGP ερωτήματα, ενώ οι κόμβοι του δέντρου προσδιορίζουν λειτουργίες όπως το union, join, alternative κτλ. Έτσι όταν δούμε τον κόμβο join πρέπει να εκτελέσουμε το join ανάμεσα στα υποδέντρα του.

Κάθε διαφορετική υλοποίηση της jena έχει μια κλάση που υλοποιεί την διαπροσωπία του OpVisitor και η οποία προσδιορίζει τι ακριβώς πρέπει να γίνει κατά την προσπέλαση όλων των διαφορετικών κόμβων του δέντρου. Πρέπει, λοιπόν, να φτιάξουμε για το σύστημά μας έναν τέτοιο visitor και να ορίσουμε κατάλληλα τις λειτουργίες του.

Πρέπει να τονίσουμε ότι στην jena, η επιλογή του πλάνου εκτέλεσης του ερωτήματος γίνεται με αναδιάταξη του δέντρου του query πριν την προσπέλασή του για εκτέλεση. Εμείς δεν θα χρησιμοποιήσουμε την επιλογή πλάνου της jena γιατί θα

εφαρμόσουμε τον δικό μας αλγόριθμο επιλογής. Αρχικά, λοιπόν, θα αποθηκεύουμε το query στην δενδρική δομή της jena και δεν θα εφαρμόζουμε την αναδιάταξη του δέντρου. Στη συνέχεια προσπελάζουμε το δέντρο του query με τον visitor του συστήματός μας. Ο vsitor εκτελεί ενέργειες μόνο όταν βρίσκει BGP ερωτήματα. Για κάθε ένα από αυτά προετοιμάζει τις κατάλληλες δομές που θα χρησιμοποιηθούν για την εκτέλεση του δικού μας αλγόριθμου επιλογής του πλάνου. Οι δομές αυτές περιέχουν τον προσδιορισμό των index που θα χρησιμοποιήσουμε καθώς και τα αντίστοιχα στατιστικά στοιχεία για το κάθε BGP. Αφού ετοιμάσουμε τις κατάλληλες δομές, εκτελούμε τον αλγόριθμο του JoinPlanner που αναλύθηκε στο κεφάλαιο του σχεδιασμού. Σε κάθε βήμα του άπληστου αλγορίθμου μας εκτελούμε το αντίστοιχο MapReduce join και με τα δεδομένα εξόδου του ανανεώνουμε τις δομές και τα στατιστικά στοιχεία του ερωτήματός μας.

Τα αποτελέσματα του SPARQL query βρίσκονται στο φάκελο εξόδου της τελευταίας εργασίας MapReduce που εκτελέστηκε από τον Planner. Καταφέραμε, λοιπόν, με χρήση του κώδικα της jena να προσθέσουμε στο σύστημά μας απαραίτητες λειτουργίες η βασικότερες από τις οποίες είναι:

- Ο συντακτικός έλεγχος ορθότητας των SPARQL ερωτημάτων.
- Η αποδοτική ανάγνωση ενός query string με χρήση του parser της jena.

6) Πειραματικά αποτελέσματα

Όπως έχουμε αναφέρει, το σύστημά μας έχει ως στόχο την αποδοτική αποθήκευση μεγάλων σε όγκο RDF βάσεων δεδομένων. Η κατανεμημένη αποθήκευση των δεδομένων καθώς και η κατανεμημένη επεξεργασία των ερωτημάτων, μπορεί να συμβάλλουν στη δημιουργία ενός τέτοιου συστήματος. Το σύστημά μας είναι κατάλληλα σχεδιασμένο ώστε να είναι αποδοτικό όταν τα δεδομένα επεξεργασίας είναι μεγάλα. Τα βασικά και πιο γρήγορα συστήματα επεξεργασίας SPARQL ερωτημάτων εκτελούνται σε ένα μηχάνημα και αποθηκεύουν τα δεδομένα στην κύρια μνήμη του υπολογιστή. Φυσικά, δεν μπορούμε να ανταγωνιστούμε τέτοια συστήματα όταν η βάση αποτελείται από λίγα δεδομένα εισόδου. Τα πλεονεκτήματα του συστήματός μας παρουσιάζονται όταν οι βάσεις είναι αρκετά μεγάλες και δεν μπορούν να επεξεργαστούν από τέτοια συστήματα που βασίζονται στην κύρια μνήμη.

Τα πειράματά μας, λοιπόν, θα βασιστούν σε μεγάλες σε όγκο βάσεις δεδομένων. Πρέπει να βρούμε μια RDF βάση που να είναι αρκετά μεγάλη και να έχει χρησιμοποιηθεί και σε άλλες εργασίες ώστε να έχουμε σύγκριση της απόδοσης του συστήματός μας. Υπάρχουν αρκετές RDF βάσεις που έχουν προκύψει από πραγματικά δεδομένα του Internet ή από άλλες, ήδη υπάρχουσες, βάσεις. Οι βάσεις, αυτές , όμως δεν είναι πολύ μεγάλες και δεν έχουν χρησιμοποιηθεί ευρέως σε παρόμοια πειράματα.

Τα RDF δεδομένα που θα χρησιμοποιήσουμε σαν είσοδο θα προέρχονται από το LUBM generator. Το LUBM είναι ένας κώδικας που δημιουργεί RDF datasets που αναφέρονται σε στοιχεία πανεπιστημίων. Με χρήση αυτού του generator μπορούμε να παράγουμε, όσο μεγάλα, datasets, επιθυμούμε, για να τεστάρουμε το σύστημά μας. Το LUBM είναι ευρέως διαδεδομένο dataset και έχει χρησιμοποιηθεί σε πάρα πολλές ερευνητικές εργασίες. Θα μπορέσουμε, λοιπόν, να χρησιμοποιήσουμε αποτελέσματα άλλων εργασιών, για να συγκρίνουμε το σύστημά μας με τα πιο αποδοτικά συστήματα που έχουν παρουσιαστεί.

Το LUBM δημιουργεί RDF triples που αναφέρονται σε φοιτητές, καθηγητές, ερευνητές και γενικά σε όλες τις οντότητες που σχετίζονται με πανεπιστήμια. Ο κώδικας παραγωγής του dataset παίρνει, ως παράμετρο, τον αριθμό των πανεπιστημίων που θέλουμε να περιέχονται στη βάση μας. Για κάθε πανεπιστήμιο δημιουργεί μια σειρά από αρχεία xml που περιέχουν όλες τις RDF πληροφορίες. Για να χρησιμοποιήσουμε τα δεδομένα αυτά πρέπει πρώτα να τα μετατρέψουμε στη μορφή των N-triples, δηλαδή σε κάθε γραμμή να περιέχεται ένα RDF triple. Για τη μετατροπή των αρχείων xml σε μορφή N-triples χρησιμοποιήσαμε το πακέτο της jena. Στο πακέτο της jena περιέχονται αρκετές δυνατότητες μετατροπής στους διάφορους τύπους αποθήκευσης των RDF δεδομένων.

Ένα ακόμα πλεονέκτημα της χρήσης του LUBM είναι το γεγονός ότι γνωρίζουμε ακριβώς την μορφή των dataset που δημιουργούνται. Υπάρχουν αναλυτικά στατιστικά στοιχεία για όλα τα είδη πληροφοριών που περιέχονται σε αυτό. Επιπλέον, το LUBM διαθέτει μια συλλογή από 14 test SPARQL queries, τα οποία χρησιμοποιούνται ευρέως για τον έλεγχο απόδοσης των διαφόρων συστημάτων που απαντούν σε SPARQL ερωτήματα. Το σύνολο αυτών των test queries αποτελείται από όλα τα είδη των διαφορετικών ερωτημάτων. Περιέχει ερωτήματα με μεγάλο, μεσαίο και μικρό selectivity καθώς και ερωτήματα με που απαιτούν λίγα ή περισσότερα join. Έτσι, λοιπόν, έχουμε ταυτόχρονα και ένα σύνολο ερωτήσεων που θα χρησιμοποιηθούν για τον έλεγχο της απόδοσης του συστήματός μας.

Τέλος, τα test queries του LUBM εξετάζουν πολύ το θέμα της owl γενίκευσης. Για παράδειγμα το δέκατο από τα ερωτήματα του τεστ είναι:

```
select *
where {
```

?x rdf:type ub:Student .
?x ub:takesCourse <http://www.Department0.University0.edu/Course10> .
};

Το πρώτο BGP του query10 είναι το ?x rdf:type ub:Student όμως στην βάση μας υπάρχουν μόνο triples της μορφής

<http://www.Department0.University0.edu/UndergraduateStudent4> rdf:type ub:UndergraduateStudent
<http://www.Department0.University0.edu/GraduateStudent44> rdf:type ub:GraduateStudent

Δηλαδή, οι φοιτητές ορίζονται είται ως GraduateStudent ή ως UndergraduateStudent. Ακόμα υπάρχουν τα triples στη βάση που ορίζουν ότι οι ιδιότητες GraduateStudent και UndergraduateStudent είναι υποκλάσεις της κλάσης Student. Χωρίς τη γενίκευση owl, το ερώτημα αυτό δεν θα είχε καμία απάντηση. Το σύστημά μας δεν έχει ασχοληθεί με αυτή τη λειτουργία της γενίκευσης, αλλά, έχουμε βρεί έναν απλό τρόπο ώστε να απαντάμε τέτοια ερωτήματα με σκοπό να συγκρίνουμε το σύστημά μας. Για κάθε BGP λοιπόν κοιτάμε αν έχει κάποια ιδιότητα που χρειάζεται γενίκευση. Αν υπάρχει κάποια τέτοια βρίσκουμε όλες τις subclasses της και αντικαθιστούμε το BGP με BGP που περιέχουν τα subclasses.

Θα εκτελέσουμε δυο είδη πειραμάτων. Το πρώτο είδος θα αναφέρεται στη διαδικασία του bulk import των δεδομένων, στα index της Hbase. Θα εκτελέσουμε πειράματα εισαγωγής για διαφορετικά μεγέθη από datasets για να δείξουμε την δυνατότητα κλιμάκωσης του συστήματός μας. Το δεύτερο είδος πειραμάτων αναφέρεται στην εκτέλεση SPARQL queries. Θα χρησιμοποιήσουμε τα test queries του LUBM για να μελετήσουμε την απόδοση του συστήματός μας. Η χρήση των test queries, θα μας επιτρέψει την άμεση σύγκριση με παρόμοια συστήματα που έχουν παρουσιαστεί σε ερευνητικές εργασίες.

6.1) Δημιουργία των index

Σε αυτή την ενότητα θα παρουσιάσουμε τα πειράματα, που αναφέρονται στη διαδικασία εισαγωγής των RDF δεδομένων, στα index της HBase. Όπως αναφέραμε στο κεφάλαιο της υλοποίησης, θα χρησιμοποιήσουμε τη μέθοδο MapReduce BulkImport για αυτό το σκοπό. Θα πραγματοποιήσουμε, λοιπόν, δυο σειρές πειραμάτων. Στην πρώτη σειρά θα εκτελέσουμε πειράματα εισαγωγής με μεταβλητό αριθμό κόμβων, ενώ στη δεύτερη, θα πραγματοποιήσουμε πειράματα με μεταβλητό μέγεθος δεδομένων εισόδου. Με αυτά τα δυο πειράματα θα δείξουμε τις δυνατότητες κλιμάκωσης και παραλληλοποίησης του συστήματος εισαγωγής των δεδομένων.

6.1.1) Μεταβλητός αριθμός πόρων

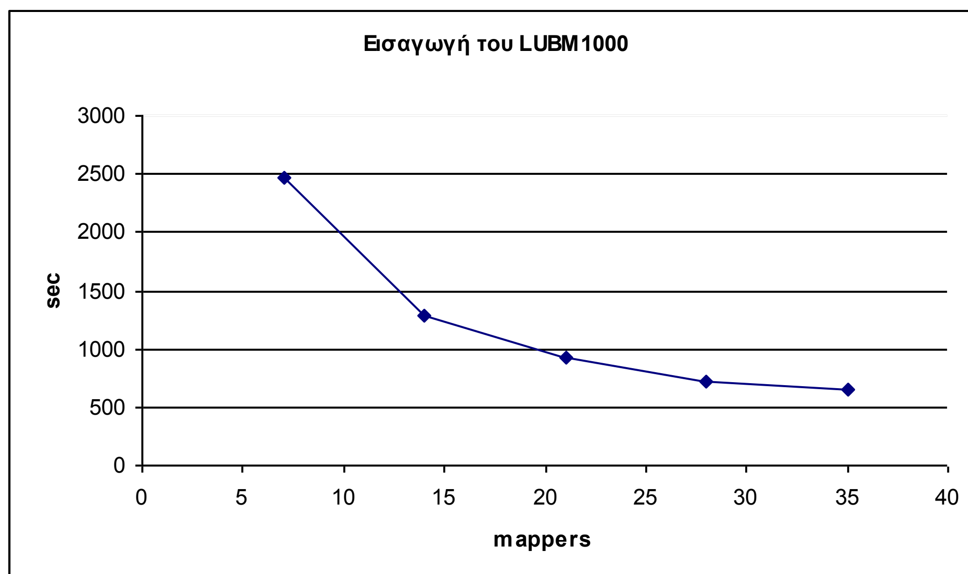
Η βασική μονάδα επεξεργασίας στο πλαίσιο του MapReduce είναι ο αριθμός των mapper και των reducer. Ένας κόμβος μπορεί να τρέχει περισσότερους από ένα mapper ή reducer, ανάλογα με την επεξεργαστική του δύναμη. Για αυτό το λόγο ο καλύτερος τρόπος προσδιορισμού των πόρων του συστήματος, είναι ο αριθμός των mappers και των reducers, που τρέχουν ταυτόχρονα. Σε αυτή την ενότητα θα χρησιμοποιήσουμε ως είσοδο το dataset LUBM1000. Το dataset, αυτό αναφέρεται σε 1000 πανεπιστήμια και διαθέτει 146 million από triples. Για την εισαγωγή του στα index της HBase θα χρησιμοποιήσουμε μεταβλητό αριθμό από πόρους. Μεταβλητή για τα πειράματά μας είναι ο αριθμός των ταυτόχρονων

mapper που τρέχουν στο cluster μας. Το cluster μας αποτελείται από 7 κόμβους. Κάθε κόμβος διαθέτει 8 virtual CPUs και 8MB μνήμης RAM. Οι κόμβοι συνδέονται μεταξύ τους με Gigabit Ethernet. Στα πειράματά μας θα έχουμε 7, 14, 28 και 35 mappers, δηλαδή 1, 2, 3, 4 και 5 mappers ανά κόμβο. Τα αποτελέσματα φαίνονται αναλυτικά στον πίνακα.

mappers	Import LUBM1000 (sec)
7	2465
14	1281
21	932
28	723
35	658

Πίνακας 34: Αποτελέσματα import για μεταβλητό μέγεθος πόρων

Τα αποτελέσματα παρουσιάζονται και μέσω της γραφικής παράστασης του σχήματος



Σχήμα 15: Μεταβλητό μέγεθος πόρων

Από τη γραφική παράσταση μπορούμε εύκολα να διαπιστώσουμε ότι η αύξηση των mappers συνεπάγεται και αύξηση της απόδοσης του συστήματός μας. Βλέπουμε ότι έχουμε καλό βαθμό παραλληλοποίησης και άρα αξιοποιούμε αποδοτικά τους πόρους τους συστήματός μας. Η αύξηση της απόδοσης δεν είναι εντελώς γραμμική, σε σχέση με τον αριθμό των mappers. Αυτό είναι λογικό αφού οι επιπλέον mappers τρέχουν στους ίδιους κόμβους και άρα επηρεάζονται. Μια ακόμα αιτία, είναι η ταυτόχρονη πρόσβαση στο δίσκο και στην κάρτα δικτύου του μηχανήματος, από τους mappers που τρέχουν σε αυτό. Αν όλοι οι mappers έτρεχαν σε διαφορετικούς κόμβους, θα είχαμε σχεδόν γραμμική αύξηση απόδοσης, λόγω του σχεδιασμού του MapReduce. Οι mappers και οι reducers δεν επικοινωνούν μεταξύ τους και άρα δε θα παρουσιαζόταν το πρόβλημα επικοινωνίας. Συμπεραίνουμε, λοιπόν, ότι το σύστημά μας είναι πολύ επεκτάσιμό και μπορούμε να έχουμε αύξηση της απόδοσης με προσθήκη νέων κόμβων.

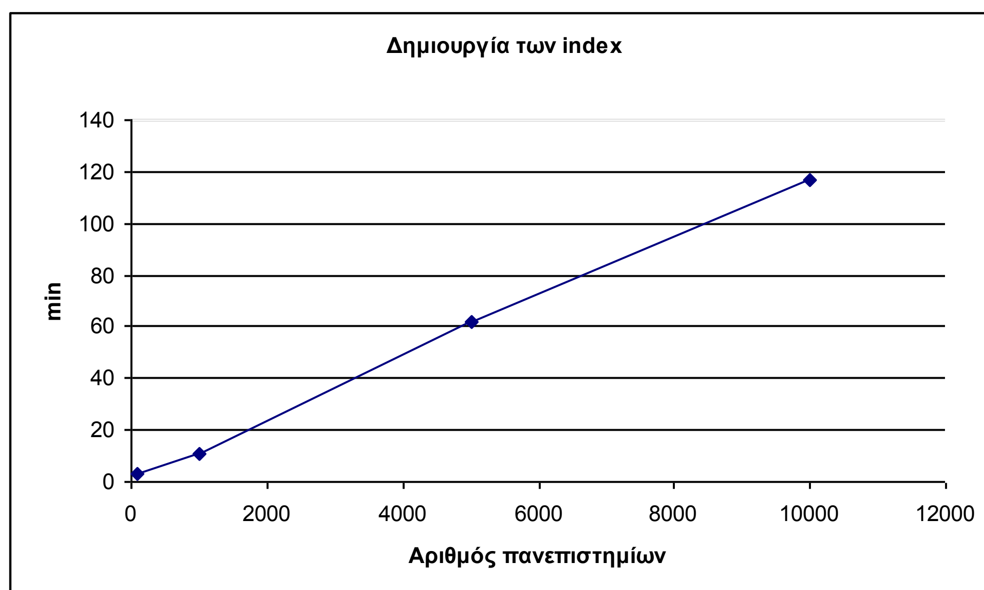
6.1.2) Μεταβλητός όγκος δεδομένων εισόδου

Σε αυτή την ενότητα θα εξετάσουμε την απόδοση του συστήματός μας, σε δεδομένα διαφορετικού όγκου. Θα χρησιμοποιήσουμε, λοιπόν, ως είσοδο, LUBM dataset με διαφορετικό αριθμό πανεπιστημίων. Το cluster μας αποτελείται από 7 κόμβους. Κάθε κόμβος διαθέτει 8 virtual CPUs και 8MB μνήμης RAM. Οι κόμβοι συνδέονται μεταξύ τους με Gigabit Ethernet. Τα test μας θα γίνουν για 100, 1000, 5000, 10000 πανεπιστήμια. Τα αποτελέσματα φαίνονται αναλυτικά στον πίνακα.

LUBM	Triples (milion)	Import (min)
100	14	3
1000	146	11
5000	632	62
10000	1125	117

Πίνακας 35: Αποτελέσματα import για μεταβλητό όγκο δεδομένων

Τα αποτελέσματα παρουσιάζονται και μέσω της γραφικής παράστασης του σχήματος



Σχήμα 16: Μεταβλητός όγκος δεδομένων

Στη γραφική παράσταση βλέπουμε ότι ο χρόνος εκτέλεσης, της εισαγωγής των RDF triples, είναι σχεδόν γραμμικός σε σχέση με το μέγεθος της εισόδου. Αυτή είναι μια πολύ σημαντική ιδιότητα, η οποία μας εξασφαλίζει την καλή απόδοση του αλγορίθμου εισαγωγής και σε πιο μεγάλα δεδομένα εισόδου. Η γραμμικότητα του αλγορίθμου σε συνδυασμό με την δυνατότητα παραλληλοποίησής του, μας δίνουν τη δυνατότητα να επεξεργαστούμε τεράστιες σε όγκο RDF βάσεις. Μπορούμε να εξασφαλίσουμε, λοιπόν, ότι με τον πολλαπλασιασμό των κόμβων του cluster, θα μπορέσουμε να διαχειριστούμε και πολλαπλάσια σε όγκο δεδομένα. Άρα, το σύστημά μας παρουσιάζει εξαιρετικές δυνατότητες επέκτασης και κλιμάκωσης.

6.2) Επεξεργασία SPARQL ερωτημάτων

Σε αυτή την ενότητα θα παρουσιάσουμε τα πειράματα, που αναφέρονται στην επεξεργασία των SPARQL ερωτημάτων. Το LUBM dataset έχει ένα σύνολο από 14 test queries που χρησιμοποιούνται για τον έλεγχο της απόδοσης των συστημάτων. Τα ερωτήματα, αυτά, έχουν χρησιμοποιηθεί ευρέως σε επιστημονικές έρευνες, για τον προσδιορισμό της απόδοσης συστημάτων επεξεργασίας SPARQL ερωτημάτων. Τα test queries, του LUBM, περιέχουν όλα τα διαφορετικά είδη SPARQL ερωτημάτων. Έχουν ερωτήματα με διαφορετικό selectivity, διαφορετικό αριθμό join και διαφορετικό αριθμό μεταβλητών. Ακόμα τα ερωτήματα, αυτά, εξετάζουν και τις ιδιότητες OWL, που παρέχει το σύστημα επεξεργασίας των queries. Για παράδειγμα εξετάζουν τις ιδιότητες της OWL γενίκευσης. Η OWL γενίκευση φαίνεται στο παρακάτω SPARQL ερώτημα.

```
select *
where {
	?x rdf:type ub:Student .
	?x ub:takesCourse <http://www.Department0.University0.edu/Course10> .
};
```

Στο ερώτημα, αυτό, βλέπουμε ότι η μεταβλητή x είναι τύπου ub:Student. Όμως στη βάση μας υπάρουν μόνο triples της μορφής ub:GraduateStudent ή ub:UndergraduateStudent. Οι ιδιότητες αυτές, ορίζονται στη βάση ως υποκλάσεις της ub:Student. Πρέπει, λοιπόν, το σύστημά μας να ανταπεξέρχεται και σε τέτοιου είδους ερωτήματα. Αυτό γίνεται εύκολα αντικαθιστώντας το ub:Student με όλες τις υποκλάσεις του.

Για τα πειράματά μας τρέξαμε μερικά από τα test ερωτήματα του LUBM. Η βάση που χρησιμοποιήσαμε για να τα τρέξουμε ήταν η LUBM 10000 που αποτελείται από 1,1 δισεκατομμύριο triples. Το dataset έχει αρχικό μέγεθος 200GB και έχει αποθηκευτεί στα index της HBase. Το cluster μας αποτελείται από 7 κόμβους. Κάθε κόμβος διαθέτει 8 virtual CPUs και 8MB μνήμης RAM. Οι κόμβοι συνδέονται μεταξύ τους με Gigabit Ethernet. Τα αποτελέσματα φαίνονται αναλυτικά στον πίνακα.

query	Χρόνος εκτέλεσης(sec)
Q1	34
Q2	1943
Q3	35
Q4	30
Q5	65
Q6	85
Q7	62
Q10	36
Q12	81
Q13	205
Q14	90

Πίνακας 36: Αποτελέσματα στα LUBMtest queries

Στη συνέχεια παρουσιάζουμε αναλυτικά τα test queries του LUBM που τρέξαμε και εξηγούμε την απόδοση του συστήματός μας σε αυτά.

Query1

(type GraduateStudent? X)
(takesCourse ?X http://www.Department0.University0.edu/GraduateCourse0)

- Το δεύτερο BGP του ερωτήματος, αυτού, είναι αρκετά επιλεκτικό. Χρησιμοποιούμε αυτό σαν είσοδο του join και στη συνέχεια, για κάθε binding του X, κοιτάζουμε στην HBase αν είναι GraduateStudent. Πετυχαίνουμε, έτσι, πολύ καλή απόδοση στην επεξεργασία του ερωτήματος.

Query2
(type GraduateStudent ?X)
(type University ?Y)
(type Department ?Z)
(memberOf ?X ?Z)
(subOrganizationOf ?Z ?Y)
(undergraduateDegreeFrom ?X ?Y)

- Αυτό το ερώτημα έχει μεγάλη πολυπλοκότητα: 3 κλάσεις και 3 ιδιότητες. Χρειάζεται 3 MapReduce join για να απαντηθεί από το σύστημά μας αφού υπάρχει τριγωνική εξάρτηση μεταξύ των μεταβλητών. Ακόμα η είσοδός του είναι τεράστια αφού εξετάζει σχεδόν όλα τα δεδομένα της βάσης. Είναι φυσικό, λοιπόν, η απόδοση του συστήματος μας να είναι μειωμένη. Στην πραγματικότητα η απόδοση είναι πολύ χειρότερη από όλα τα άλλα ερωτήματα, αλλά αυτό δικαιολογείται κυρίως από το τεράστιο μέγεθος της εισόδους και τα 3 MapReduce jobs που πρέπει να εκτελεστούν.

Query3
(type Publication ?X)
(publicationAuthor ?X http://www.Department0.University0.edu/AssistantProfessor0)

- Είναι παρόμοιο με το πρώτο query αλλά έχει διαφορετικές κλάσεις. Η απόδοσή του είναι σχεδόν ίδια με του Q1.

Query4
(type Professor ?X)
(worksFor ?X http://www.Department0.University0.edu)
(name ?X ?Y1)
(emailAddress ?X ?Y2)
(telephone ?X ?Y3)

- Το βασικό στοιχείο σε αυτό το ερώτημα είναι ότι ρωτάει για πολλαπλές ιδιότητες μιας συγκεκριμένης οντότητας. Το δεύτερο BGP του ερωτήματος είναι αρκετά επιλεκτικό. Χρησιμοποιούμε, αυτό, σαν είσοδο του join και στη συνέχεια, για κάθε binding του X, παίρνουμε από την HBase τα αντίστοιχα δεδομένα για τα υπόλοιπα BGP. Πετυχαίνουμε, έτσι, πολύ καλή απόδοση στην επεξεργασία του ερωτήματος.

Query5
(type Person ?X)
(memberOf ?X http://www.Department0.University0.edu)

- Όμοιο με το Q1, αλλά περιέχει την κλάση Person η οποία έχει πολύ μεγάλη ιεραρχία Χρειάζεται, λοιπόν, περισσότερος χρόνος για τον έλεγχο όλων των διαφορετικών subclasses του Person.

Query6
(type Student ?X)

- Επιστρέφει όλους τους graduate και undergraduate student. Έχει μεγάλη είσοδο και μικρό selectivity.

Query7
(type Student ?X)
(type Course ?Y)
(teacherOf http://www.Department0.University0.edu/AssociateProfessor0 ?Y)
(takesCourse ?X ?Y)

- Χρειάζεται δυο MapReduce join για να εκτελεστεί. Έχει μεγάλη εισοδο και καλό selectivity από το τρίτο BGP.

Query10
(type Student ?X)
(takesCourse ?X http://www.Department0.University0.edu/GraduateCourse0)

- Το δεύτερο BGP του ερωτήματος, αυτού, είναι αρκετά επιλεκτικό. Χρησιμοποιούμε αυτό σαν είσοδο του join και στη συνέχεια, για κάθε binding του X, κοιτάζουμε στην HBase αν είναι Student. Η κλάση Student έχει δυο υποκλάσεις τις GraduateStudent και UndergraduateStudent.

Query12
(type Chair ?X)
(type Department ?Y)
(worksFor ?X ?Y)
(subOrganizationOf ?Y http://www.University0.edu)

- Χρειάζεται δυο MapReduce join για να εκτελεστεί. Έχει καλό selectivity από το τρίτο BGP και μοιάζει με το Q7.

Query13
(type Person ?X)
(hasAlumnus http://www.University0.edu ?X)

-Η ιδιότητα has Alumnus είναι η αντίστροφη της degreeFrom. Για να εκτελέσουμε το query απλά αντικαταστήσαμε το δεύτερο BGP με το degreeFrom. Το δεύτερο BGP είναι επιλεκτικό και το χρησιμοποιούμε σαν είσοδο για να έχει καλή απόδοση το join.

Query14
(type UndergraduateStudent ?X)

-Από τα ποιο εύκολα ερωτήματα του test αφού ζητάει να φέρουμε όλους τους UndergraduateStudent. Το μέγεθός του βέβαι είναι πολύ μεγάλο και σε αυτό οφείλεται όλη η καθυστέρηση.

Γενικά παρατηρούμε ότι το σύστημά μας έχει πολύ καλή απόδοση σχεδόν σε όλα τα queries. Το μόνο query που καθυστερεί σημαντικά είναι το δεύτερο. Η μικρή απόδοση του

δεύτερου query βασίζεται στο μεγάλο όγκο των δεδομένων που πρέπει να επεξεργαστούμε και στην έλλειψη κάποιου BGP με μεγάλο selectivity.

6.3) Σύγκριση με άλλα συστήματα

Σε αυτή την ενότητα θα συγκρίνουμε την απόδοση του συστήματός μας με άλλα παρόμοια ερευνητικά συστήματα. Αρχικά θα το συγκρίνουμε με το HadoopRDF [16], το οποίο είναι και αυτό ένα σύστημα που εκτελεί τα SPARQL queries με MapReduce joins. Οι βασικές διαφορές του HadoopRDF και του συστήματός μας είναι:

- Αποθήκευση δεδομένων: Το σύστημά μας χρησιμοποιεί 3 HBase indexes ενώ το HadoopRDF αποθηκεύει τα δεδομένα απλά σε αρχεία του HDFS. Τα ονόματα των αρχείων χρησιμοποιούνται για να δημιουργήσουν ένα pos index.
- JoinPlanner: Εμείς χρησιμοποιούμε άπληστη συνάρτηση που βασίζεται στο συνδιασμό e-count και selectivity. Στο HadoopRDF χρησιμοποιούν μόνο το e-count. Αυτό σημαίνει ότι δημιουργούν πλάνο εκτέλεσης των join που ελαχιστοποιεί μόνο τον αριθμό των join και δεν ενδιαφέρεται για το selectivity τους.
- MapReduce join: Στο HadoopRDF χρησιμοποιούν έναν αλγόριθμο παρόμοιο με το FullInputJoin ενώ εμείς χρησιμοποιούμε τον PartialInputJoin που είναι πιο αποδοτικός.

Στο δημοσιευμένο άρθρο του HadoopRDF [16], παρουσιάζονται τα αποτελέσματά του για το LUBM dataset με τα οποία θα συγκρίνουμε τα δυο συστήματα. Για το HadoopRDF χρησιμοποιήθηκε cluster 10 κόμβων. Κάθε κόμβος έχει τα παρακάτω χαρακτηριστικά:
Pentium IV 2.80 GHz processor, 4 GB main memory, 640 GB disk space.

Το cluster μας αποτελείται από 7 κόμβους. Κάθε κόμβος διαθέτει 8 virtual CPUs και 8MB μνήμης RAM. Τα queries έτρεξαν για το LUBM10000 dataset και τα αποτελέσματα φαίνονται στον παρακάτω πίνακα.

query	HadoopRDF(sec)	3HbaseStore(sec)
Q1	248.3	**34**
Q2	**801.9**	1943
Q4	1430.2	**30**
Q12	204.4	**81**
Q13	325.4	**205**

Πίνακας 37: Σύγκριση με HadoopRDF

Βλέπουμε, λοιπόν, ότι το σύστημά μας, 3HbaseStore, παρουσιάζει καλύτερη απόδοση από το HadoopRDF σε όλα τα ερωτήματα εκτός από το Q2. Σε αυτό το ερώτημα το HadoopRDF έχει το πλεονέκτημα ότι η πρόσβαση σε απλά αρχεία είναι πολύ πιο γρήγορη από την πρόσβαση και την ανάκτηση δεδομένων από την HBase. Το ερώτημα αυτό έχει τεράστια είσοδο και η ανάκτηση αυτών των δεδομένων γίνεται πιο αποδοτικά στο HadoopRDF. Ακόμα, βλέπουμε ότι η χρήση του PartialInputJoin μας δίνει τη δυνατότητα να έχουμε καλή απόδοση σε ερωτήματα με μεγάλο selectivity. Κάτι τέτοιο δεν επιτυγχάνεται στο HadoopRDF.

Στη συνέχεια θα συγκρίνουμε το σύστημά μας με ένα από τα πιο γρήγορα συστήματα επεξεργασία RDF δεδομένων, το RDF3X. Το RDF3X τρέχει αποκλειστικά σε έναν κόμβο και χρησιμοποιεί μια threaded αρχιτεκτονική για παραλληλοποίηση των εργασιών του. Τα δεδομένα για το RDF3X προήλθαν από το άρθρο [16]. Για την εκτέλεση του RDF3X χρησιμοποιήθηκε ένας ισχυρός υπολογιστής με χαρακτηριστικά: 2.80 GHz quad core processor, 8 GB main memory, 1 TB disk space.

Το cluster που έτρεξε το σύστημά μας αποτελείται από 7 κόμβους. Κάθε κόμβος διαθέτει 8 virtual CPUs και 8MB μνήμης RAM. Τα queries έτρεξαν για το LUBM10000 dataset και τα αποτελέσματα φαίνονται στον παρακάτω πίνακα.

query	**RDF-3X(sec)**	**3HbaseStore(sec)**
Q1	**0.373**	34
Q2	**1240.21**	1943
Q4	**0.856**	30
Q12	**0.298**	81
Q13	380.731	**205**

Πίνακας 38: Σύγκριση με RDF3X

Από την παραπάνω σύγκριση βλέπουμε ότι το RDF3X είναι πιο γρήγορα στα περισσότερα ερωτήματα του Test. Αυτό βασίζεται κυρίως στο γεγονός ότι εκτελεί τα join σε έναν κόμβο. Το σύστημά μας, παρόλο που εξεργάζεται εξίσου λίγα δεδομένα, τα επεξεργάζεται με MapReduce εργασίες, η οποίες θέλουν αρκετό χρόνο για την εκκίνηση και τον τερματισμό τους. Τέλος βλέπουμε στα ερωτήματα με μεγάλη είσοδο, όπως Q2 καιQ13, η απόδοση των συστημάτων είναι σχεδόν ίδια και εξαρτάται κυρίως από την υπολογιστική δύναμη που έχουν στη διάθεσή τους. Το πλεονέκτημα του συστήματός μας έναντι του RDF3X είναι ότι είναι πολύ πιο κλιμακώσιμο. Το σύστημά μας μπορεί εύκολα να επεξεργαστεί πολύ μεγαλύτερα σε όγκο δεδομένα. Για τέτοια δεδομένα το σύστημα του RDF3X αποτυγχάνει να εκτελέσει τα queries με μεγάλο μέγεθος εισόδου.

7) Συμπεράσματα

Σε αυτή την εργασία παρουσιάσαμε ένα σύστημα, το οποίο είναι ικανό να επεξεργαστεί τεράστια σε όγκο RDF δεδομένα. Βρήκαμε αποδοτικούς τρόπους αποθήκευσης και δημιουργίας των index της HBase. Ακόμα, δημιουργήσαμε αποδοτικούς αλγορίθμους για την εκτέλεσης των SPARQL ερωτημάτων με χρήση της τεχνικής του MapReduce. Από τα πειράματα, που εκτελέσαμε, είδαμε ότι το σύστημα είναι ιδιαίτερα επεκτάσιμο και μπορεί να ανταποκριθεί σε ακόμη μεγαλύτερα dataset, απλά με την εισαγωγή νέων κόμβων. Το σύστημα, παρουσιάζει και εξαιρετική απόδοση στην απάντηση SPARQL ερωτημάτων. Είδαμε ότι είχαμε αρκετά καλύτερη απόδοση από το παρόμοιο σύστημα του HadoopRDF και ότι έχουμε πλεονεκτήματα απέναντι σε συστήματα όπως το RDF3X, που εκτελούνται σε έναν υπολογιστή. Τα πλεονεκτήματα, απέναντι σε συστήματα που τρέχουν σε ένα υπολογιστή, παρουσιάζονται όταν τα dataset, που χρησιμοποιούμε, είναι αρκετά μεγάλα και είναι δύσκολο να επεξεργαστούν τοπικά από έναν κόμβο.

Πρέπει, σε αυτό το σημείο, να πούμε ότι το σύστημα, που δημιουργήσαμε, έχει αρκετές ακόμα δυνατότητες βελτίωσης και επέκτασης. Αρχικά, μια δυνατότητα βελτίωσης είναι η απάντηση όλων των διαφορετικών SPARQL ερωτημάτων, αφού αυτή τη στιγμή απαντάει μόνο απλά select. Επιπλέον, μπορούμε να βελτιώσουμε και να ενισχύσουμε το OWL reasoning του συστήματός μας προσθέτοντας νέες δυνατότητες. Επίσης, μπορούμε να κρατάμε περισσότερα στατιστικά στοιχεία, για τα δεδομένα της βάσης μας, ώστε να έχουμε την καλύτερη δυνατή πρόβλεψη απόδοσης των join. Τέλος, μπορούμε να δημιουργήσουμε ένα μεικτό τρόπο εκτέλεσης των join, στον οποίο τα join με μικρό input θα μπορούν να γίνονται τοπικά, σε έναν κόμβο.

Αναφορές

[1] www.wikipedia.org
[2] www.w3.org
[3] Georg Lausen Christoph Pinkel Michael Schmidt, Thomas Hornung. SP2 Bench: A SPARQL performance benchmark. IEEE International Conference on Data Engineering, 2009, Beijing, China, 2009.
[4] http://labs.google.com/papers/bigtable.html
[5] http://labs.google.com/papers/mapreduce-osdi04.pdf
[6] http://developer.yahoo.com/hadoop/tutorial
[7] http://wiki.apache.org
[8] http://hadooper.blogspot.com/
[9] http://jimbojw.com/wiki/index.php
[10] http://www.w3.org/TR/rdf-sparql-query/
[11] http://jena.sourceforge.net/
[12] J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: Implementing the Semantic Web Recommendations," Proc. 13th Int'l World Wide Web Conf. Alternate Track Papers and Posters, pp. 74-83, 2004.
[13] A. Kiryakov, D. Ognyanov, and D. Manov, "OWLIM: A Pragmatic Semantic Repository for OWL," Proc. Int'l Workshop Scalable Semantic Web Knowledge Base Systems (SSWS), 2005.
[14] http://www.openlinksw.com/dataspace/dav/wiki/Main/VOSArticleLUBMBenchmar k
[15] T. Neumann and G. Weikum, "RDF-3X: A RISC-Style Engine for RDF," Proc. VLDB Endowment, vol. 1, no. 1, pp. 647-659, 2008.
[16] Mohammad Farhan Husain, James McGlothlin, Mohammad Mehedy Masud, Latifur R. Khan, and Bhavani Thuraisingham, Fellow, IEEE, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing " IEEE Transactions on Knowledge and Data Engineering, vol. 23, no. 9, September 2011
[17] D.J. Abadi, A. Marcus, S.R. Madden, and K. Hollenbach, "SW- Store: A Vertically Partitioned DBMS for Semantic Web Data Management," VLDB J., vol. 18, no. 2, pp. 385-406, Apr. 2009.
[18] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," Proc. VLDB Endowment, vol. 1, no. 1, pp. 1008-1019, 2008.
[19] M. Atre, J. Srinivasan, and J.A. Hendler, "BitMat: A Main-Memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries," Proc. Int'l Semantic Web Conf., 2008.
[20] Jianling Sun, Qiang Jin, "Scalable RDF Store Based on HBase and MapReduce" , 2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)
[21] L. Ding, T. Finin, Y. Peng, P.P. da Silva, and D.L. Mcguinness, "Tracking RDF Graph Provenance Using RDF Molecules," Proc. Fourth Int'l Semantic Web Conf., 2005.
[22] http://swat.cse.lehigh.edu/projects/lubm