Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Πληροφορικής και Τεχνολογίας Υπολογιστών

Κλιμακώσιμοι και βασισμένοι στο φόρτο εργασίας αλγόριθμοι διαχείρισης μη δομημένων δεδομένων

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

ΝΙΚΟΛΑΟΣ Π. ΠΑΠΑΗΛΙΟΥ

Αθήνα, 01/11/2016

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Πληροφορικής και Τεχνολογίας Υπολογιστών

Κλιμακώσιμοι και βασισμένοι στο φόρτο εργασίας αλγόριθμοι διαχείρισης μη δομημένων δεδομένων

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

ΝΙΚΟΛΑΟΣ Π. ΠΑΠΑΗΛΙΟΥ

Συμβουλευτική Επιτροπή: Νεκτάριος Κοζύρης
Δημήτριος Τσουμάκος
Παναγιώτης Τσανάκας

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 01/11/2016.

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής ΕΜΠ	Δημήτριος Τσουμάκος Επίκουρος Καθηγητής Ιόνιο Πανεπιστήμιο	Παναγιώτης Τσανάκας Καθηγητής ΕΜΠ
.....	.....	.....
Μανόλης Κουμπαράκης Καθηγητής ΕΚΠΑ	Ιωάννης Κωτίδης Αναπ. Καθηγητής ΟΠΑ	Νικόλαος Μαμουλής Αναπ. Καθηγητής Πανεπιστήμιο Ιωαννίνων

.....
Ανδρέας-Γεώργιος Σταφυλοπάτης
Καθηγητής ΕΜΠ

Αθήνα, 01/11/2016

. . . . . .

ΝΙΚΟΛΑΟΣ Π. ΠΑΠΑΗΛΙΟΥ
Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Η εκπόνηση της διατριβής χρηματοδοτήθηκε από το Ίδρυμα Κρατικών Υποτροφιών στο πλαίσιο των υποτροφιών αριστείας ΙΚΥ Β' Κύκλου Σπουδών (Διδακτορικό) στην Ελλάδα-Πρόγραμμα Siemens.

Ευχαριστίες

Αυτή η διατριβή είναι το αποτέλεσμα των μεταπτυχιακών σπουδών μου στο Εργαστήριο Υπολογιστικών Συστημάτων του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη των καθηγητών Δημητρίου Τσουμάκου και Νεκτάριου Κοζύρη. Θα ήθελα να επωφεληθώ αυτής της ευκαιρίας για να εκφράσω τη βαθύτατη ευγνωμοσύνη και εκτίμησή μου σε όλους όσους με καθοδήγησαν αυτά τα χρόνια.

Πρώτα απ' όλα, είμαι βαθιά ευγνώμων στον επιβλέποντα καθηγητή μου, Δημήτριο Τσουμάκο, ο οποίος ήταν ο κύριος μέντοράς μου και αποτέλεσε πηγή ενθάρρυνσης και έμπνευσης καθ' όλη τη διάρκεια των μεταπτυχιακών σπουδών μου. Στο πλευρό του έμαθα να διατηρώ ένα υψηλό ερευνητικό επίπεδο καθώς και δεξιότητες όπως η σύνταξη επιστημονικών κειμένων και η σχεδίαση υπολογιστικών συστημάτων. Αυτή η διατριβή δεν θα ήταν ίδια χωρίς τη συνεχή προσπάθεια και την υποστήριξή του.

Θα ήθελα επίσης να εκφράσω τη βαθιά ευγνωμοσύνη μου στον επιβλέποντα καθηγητή μου, Νεκτάριο Κοζύρη, που μου έδωσε την ευκαιρία να μελετήσω ένα ενδιαφέρον και σύγχρονο αντικείμενο έρευνας. Η συνεχής υποστήριξή του, τόσο πνευματική όσο και υλική, ήταν υψίστης σημασίας για τις μεταπτυχιακές σπουδές μου. Το ερευνητικό περιβάλλον που έχει δημιουργήσει ενθαρρύνει την καινοτομία και έχει τη δυνατότητα να υποστηρίξει την έρευνα παρέχοντας πρόσβαση στις πιο σύγχρονες τεχνολογίες υπολογιστικών συστημάτων.

Επιπλέον, η διατριβή αυτή δεν θα ήταν δυνατή χωρίς την οικονομική υποστήριξη του Ιδρύματος Κρατικών Υποτροφιών (ΙΚΥ) και του προγράμματος "υποτροφίες αριστείας για μεταπτυχιακές σπουδές στην Ελλάδα - Πρόγραμμα SIEMENS". Οφείλω μεγάλη ευγνωμοσύνη και

Στην οικογένειά μου

σεβασμό προς το Ίδρυμα Κρατικών Υποτροφιών, καθώς και προς την κυρία Ειρήνη Σταμογιώργου που διαχειρίζεται το πρόγραμμα υποτροφιών και αποτέλεσε το βασικό σημείο επαφής μου κατά τη διάρκεια του διδακτορικού μου.

Θα ήθελα επίσης να εκφράσω την ιδιαίτερη εκτίμηση και τις ευχαριστίες μου σε διάφορα άλλα άτομα στο Εργαστήριο Υπολογιστικών Συστημάτων που με βοήθησαν στην πορεία των σπουδών μου: Ιωάννης Κωνσταντίνου, Κατερίνα Δόκα, Ιωάννης Γιαννακόπουλος, Ιωάννης Μυτιλήνης, Δημήτρης Σαρλής και Βίκτωρας Γιαννακούρης. Σας ευχαριστώ όλους για την υπέροχη συνεργασία, τις ατελείωτες συζητήσεις και τη φιλία σας όλα αυτά τα χρόνια.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου και όλους τους φίλους μου για την ανιδιοτελή αγάπη, την υποστήριξη και τη βοήθειά τους, καθώς και για τη χαρά που φέρνουν στη ζωή μου.

Περιεχόμενα

1 Εισαγωγή **21**
1.1 Κίνητρο . . . 21
1.2 Σημασιολογικός Ιστός . . . 23
1.2.1 Πρότυπο Δεδομένων RDF . . . 25
1.2.2 Γλώσσα επερώτησης SPARQL . . . 26
1.2.3 Συνεισφορές . . . 27
1.3 Ανάλυση Δεδομένων Δικτύων . . . 30
1.3.1 Συνεισφορές . . . 31
1.4 Οργάνωση κειμένου . . . 32

2 Η κατανεμημένη βάση RDF δεδομένων H_2RDF+ **33**
2.1 Εισαγωγή . . . 33
2.2 Αρχιτεκτονική . . . 35
2.3 Σχήμα ευρετηρίασης . . . 35
2.3.1 Ευρετήρια HBase . . . 36
2.3.2 Ευρετήρια Στατιστικών . . . 37
2.4 Μαζική ευρετηρίαση RDF δεδομένων με χρήση MapReduce . . . 37
2.4.1 Πρώτη εργασία MapReduce . . . 38
2.4.2 Δεύτερη εργασία MapReduce . . . 39
2.4.3 Τρίτη εργασία MapReduce . . . 40
2.4.4 Τέταρτη εργασία MapReduce . . . 40
2.4.5 Indexing storage space . . . 41

2.5 Αλγόριθμοι Εκτέλεσης Συνενώσεων 42
2.5.1 Αλγόριθμος Merge συνένωσης με χρήση MapReduce 42
2.5.2 Αλγόριθμος Sort-Merge συνένωσης με χρήση MapReduce 43
2.5.3 Κεντρικοί Αλγόριθμοι συνενώσεων 44
2.5.4 Ομαδοποίηση Ενδιάμεσων αποτελεσμάτων 44
2.6 Εκτέλεση και Βελτιστοποίηση Ερωτημάτων 46
2.6.1 Απόδοση σάρωσης των HBase ευρετηρίων 47
2.6.2 Μοντέλο κόστους εκτέλεσης Merge συνενώσεων 48
2.6.3 Μοντέλο κόστους εκτέλεσης Sort-Merge συνενώσεων 48
2.6.4 Βελτιστοποίηση Πλάνου εκτέλεσης 49
2.6.5 Ελαστική εκτέλεση 50
2.7 Πειράματα 51
2.7.1 Συστοιχία πειραμάτων 51
2.7.2 Συγκρινόμενα συστήματα 52
2.7.3 Σύνολα δεδομένων 52
2.7.4 Συγκριση ευρετηρίων 53
2.7.5 Ευρετηρίαση δεδομένων 54
2.7.6 Σύγκριση με άλλα συστήματα 57
2.7.7 Συγκριση αλγορίθμων συνένωσης 60
2.7.8 Εκτέλεση επιλεκτικών ερωτημάτων 62
2.7.9 Κλιμακωσιμότητα ερωτημάτων 63
2.7.10 Βελτιστοποιητής πλάνου εκτέλεσης και ελαστική διάθεση πόρων 64

3 Κρυφή Μνημη SPARQL ερωτημάτων 67
3.1 Εισαγωγή 67
3.2 Απλοποίηση SPARQL ερωτημάτων 70
3.3 Κανονικοποιημένες ετικέτες για SPARQL ερωτήματα 74
3.3.1 Δημιουργία κανονικοποιημένων ετικετών για γράφους συνενώσεων SPARQL 76
3.3.2 Δημιουργία κανονικοποιημένων ετικετών για απλοποιημένους γράφους συνενώσεων SPARQL 79
3.3.3 Χρησιμοποίηση των κανονικοποιημένων ετικετών για εξέταση ισομορφισμού υπογράφων 82
3.3.4 Πολυπλοκότητα δημιουργίας κανονικοποιημένων ετικετών 84
3.4 Σχεδιασμός εκτέλεσης ερωτημάτων 85
3.4.1 Εξερεύνηση Multi-way πλάνων συνένωσης 86
3.4.2 Αλγόριθμος δυναμικού προγραμματισμού για εύρεση βέλτιστου πλάνου εκτέλεσης 88

3.5 Κρυφή μνήμη SPARQL αποτελεσμάτων 92
3.5.1 Αφαίρεση σταθερών ερωτήματος 93
3.5.2 Δέντρο Αποτελεσμάτων 95
3.6 Ελεγκτής Κρυφής Μνήμης 103
3.6.1 Εύρεση κερδοφόρων ερωτημάτων 103
3.7 Πειράματα 108
3.7.1 Δημιουργία κανονικοποιημένων ετικετών για SPARQL ερωτήματα . . . 109
3.7.2 Βελτιστοποιητής Δυναμικού Προγραμματισμού 111
3.7.3 Απόδοση κρυφής μνήμης 112

4 Ανάλυση δεδομένων κίνησης δικτύων 117
4.1 Εισαγωγή 117
4.2 Περιγραφή Συστήματος 119
4.3 Αλγόριθμοι 122
4.3.1 Map συνενώσεις ισότητας (equi-joins) 122
4.3.2 Map συνενώσεις ανισότητας (theta-join) 123
4.3.3 Map συνενώσεις ισότητας (equi-joins) με μεγάλο σύνολο μέτα-δεδομένων 123
4.4 Πειράματα 126
4.4.1 Περιγραφή του Dataset 127
4.4.2 Περιγραφή του cluster 128
4.4.3 Σύγκριση Συστημάτων 129
4.4.4 Μελέτη κλιμακωσιμότητας του συστήματος 131

5 Συσχετιζόμενες Εργασίες 135
5.1 RDF Βάσεις Δεδομένων 135
5.1.1 Κρυφές μνήμες για SPARQL ερωτήματα 138
5.2 Ανάλυση δεδομένων κίνησης δικτύων 140

6 Συμπεράσματα 143

Κατάλογος σχημάτων

1.1 Αύξηση μεγέθους των μη δομημένων δεδομένων 22
1.2 Τεχνολογίες Σημασιολογικού Ιστού 24
1.3 Σύνολα Δεδομένων Σημασιολογικού Ιστού 24
1.4 Εξέλιξη των linked data 25
1.5 RDF τριπλέτες 25
1.6 Γράφος Γνώσης 26
1.7 SPARQL ερώτημα 27
1.8 Στατιστικά στοιχεία που προέρχονται από την παρακολούθηση της κίνησης ενός IXP 31

2.1 Αρχιτεκτονική του H_2RDF+ 35
2.2 Ομαδοποιημένα ενδιάμεσα αποτελέσματα 45
2.3 Συνένωση με χρήση ομαδοποιημένων ενδιάμεσων αποτελεσμάτων 46
2.4 Χρόνος ευρετηρίασης για μεταβλητό μέγεθος δεδομένων 55
2.5 Χρόνος ευρετηρίασης για μεταβλητό αριθμό πόρων 55
2.6 Κλιμακωσιμότητα αλγορίθμων συνένωσης 61
2.7 Ρυθμός εκτέλεσης ταυτόχρονων ερωτημάτων 62
2.8 Κλιμακωσιμότητα της κατανεμημένης εκτέλεσης ερωτημάτων για διαφορετικό μέγεθος δεδομένων και αριθμό κόμβων 63
2.9 Ελαστική διάθεση πόρων 64

3.1 Γράφος του SPARQL ερωτήματος Q_e 70
3.2 Γράφος συνενώσεων του ερωτήματος Q_e 71

3.3 Απλοποιημένος γράφος ερωτήματος SPARQL για το ερώτημα Q_e 72
3.4 Απλοποιημένος γράφος συνενώσεων του ερωτήματος Q_e 73
3.5 Μετασχηματισμός του SPARQL ερωτήματος 77
3.6 Μετασχηματισμός σε κατευθυνόμενο με χρωματισμένες κορυφές γράφο 78
3.7 Ετικέτες του απλοποιημένου γράφου συνενώσεων για το ερώτημα Q_e 79
3.8 Ερώτημα που είναι ισομορφικός υπογράφος σκελετού-αστέρα του Q_e 83
3.9 Εκτέλεση του αλγορίθμου βελτιστοποίησης για το ερώτημα Q_e 92
3.10 Αρχιτεκτονική του συστήματος 93
3.11 Δέντρο προσωρινά αποθηκευμένων αποτελεσμάτων 97
3.12 Αναζήτση του δέντρου αποτελεσμάτων για το ερώτημα Q_r 97
3.13 Δέντρο Εκτίμησης Οφέλους για το αίτημα κρυφής μνήμης (Q_r) 104
3.14 Δημιουργία κανονικοποιημένων ετικετών για SPARQL ερωτήματα. Αξιολόγηση απόδοσης για διαφορετικά μοτίβα ερωτημάτων και μεγέθη ερωτήματος 109
3.15 Απόδοση βελτιστοποιητή πλάνου εκτέλεσης 112
3.16 Μέσος χρόνος απόκρισης για το W1 113
3.17 Μέσος χρόνος απόκρισης για το W2 113
3.18 Μέσος χρόνος απόκρισης για το W3 113
3.19 Μέσος χρόνος απόκρισης για το W4 113

4.1 Στατιστικά στοιχεία που προέρχονται από την παρακολούθηση της κίνησης ενός IXP 118
4.2 Αρχιτεκτονική του Datix 121
4.3 Εκτέλεση συνενώσεων με χρήση K-d Tree. 125
4.4 Σύγκριση του Hive με το Shark 131
4.5 Κλιμακωσιμότητα με βάση τον αριθμό των υπολογιστικών πόρων 132
4.6 Κλιμακωσιμότητα με βάση το μέγεθος των δεδομένων 132
4.7 Σύγκριση Text και ORC μορφής αρχείων 132

Κατάλογος πινάκων

2.1 Σχήμα μεταβλητού μήκους κωδικοποίησης 41
2.2 Σύγκριση απαιτήσεων σε χώρο αποθήκευσης 53
2.3 Σύγκριση ρυθμού ανάγνωσης και ταχύτητας αναζήτησης των ευρετηρίων . . . 54
2.4 Συγκριση απόδοση για τα H_2RDF+, H_2RDF και HadoopRDF χρησιμοποιώντας τα LUBM και Yago2 σύνολα δεδομένων 57
2.5 Σύγκριση απόδοσης των H_2RDF+ και RDF-3X 59

4.1 Χαρακτηριστικά κόμβων του cluster 128
4.2 Χρόνος εισαγωγής δεδομένων σε σχέση με το μέγεθός τους 129
4.3 Σύγκριση απλών συνενώσεων με το Datix 129

Περίληψη

Ο ρυθμός με τον οποίο τα δεδομένα περιγράφονται, ερωτώνται και ανταλλάσσονται χρησιμοποιώντας μη δομημένες αναπαραστάσεις δεδομένων συνεχώς αυξάνεται. Μια από τις κυριότερες πηγές τέτοιων δεδομένων είναι οι τεχνολογίες Σημασιολογικού Ιστού, οι οποίες χρησιμοποιούν το RDF μοντέλο για την αναπαράσταση των δεδομένων του παγκόσμιου ιστού. Η μεγάλη αύξηση των διαθέσιμων RDF δεδομένων επιβάλει την εύρεση αποδοτικών και κλιμακώσιμων λύσεων για την διαχείρισή τους. Σε αυτή την διατριβή χρησιμοποιούμε κατανεμημένες μεθόδους διαχείρισης των RDF δεδομένων, οι οποίες μπορούν να κλιμακώσουν σε απεριόριστα μεγάλο αριθμό δεδομένων. Παρουσιάζουμε το H2RDF, μια πλήρως κατανεμημένη βάση αποθήκευσης RDF δεδομένων, η οποία συνδυάζει το πλαίσιο επεξεργασίας του MapReduce με μια κατανεμημένη NoSQL βάση. Δημιουργώντας 6 διαφορετικά ευρετήρια δεδομένων με HBASE πίνακες, το H2RDF μπορεί να επεξεργαστεί σύνθετα ερωτήματα με κλιμακώσιμο τρόπο κάνοντας προσαρμοστικές αποφάσεις για την σειρά και τον τρόπο εκτέλεσης των συνενώσεων. Οι συνενώσεις εκτελούνται κατανεμημένα ή κεντρικά, σε έναν υπολογιστή, ανάλογα με το κόστος τους. Επιπλέον, παρουσιάζουμε ένα καινοτόμο σύστημα που στοχεύει στην προσαρμοστική και βασισμένη στα ερωτήματα που εκτελούνται, δεικτοδότηση RDF γράφων με τη χρήση μιας κρυφής μνήμης για αποτελέσματα SPARQL ερωτημάτων. Στην καρδιά του συστήματος βρίσκεται ένας αλγόριθμος που παράγει κανονικοποιημένες ετικέτες για SPARQL ερωτήματα και χρησιμοποιείται για την μονοσήμαντη δεικτοδότηση και αναφορά σε SPARQL υπογράφους, αντιμετωπίζοντας το πρόβλημα των ισομορφικών γράφων. Ένας αλγόριθμος δυναμικού προγραμματισμού χρησιμοποιείται για την εύρεση του βέλτιστου πλάνου εκτέλεσης των ερωτημάτων, εξετάζοντας την αξιοποίηση τόσο των βασικών RDF ευρετηρίων καθώς και

των προσωρινά αποθηκευμένων αποτελεσμάτων SPARQL ερωτημάτων. Με την παρακολούθηση των αιτημάτων στην κρυφή μνήμη, το σύστημά μας είναι σε θέση να προσδιορίσει και να τοποθετήσει στην κρυφή μνήμη ερωτήματα που, αν και δεν έχουν ζητηθεί, μπορούν να μειώσουν τους χρόνους εκτέλεσης των ερωτημάτων των χρηστών. Η προτεινόμενη κρυφή μνήμη είναι επεκτάσιμη, επιτρέποντας την ενσωμάτωσή της σε πολλαπλές RDF βάσεις δεδομένων.

Μια ακόμα πηγή συνεχώς αυξανόμενης ποσότητας δεδομένων είναι και η κίνηση δεδομένων στο Internet. Αυτό γίνεται περισσότερο εμφανές σε κόμβους ουδέτερης διασύνδεσης (IXPs) από τους οποίους πλέον διέρχονται έως και Terabytes δεδομένων ανά ώρα. Για την αποδοτική διαχείριση και επεξεργασία τέτοιων δεδομένων παρουσιάζουμε το Datix, ένα πλήρως κατανεμημένο, ανοιχτού κώδικα σύστημα ανάλυσης δεδομένων κίνησης δικτύων. Το Datix βασίζεται σε τεχνικές έξυπνης κατανομής των δεδομένων, οι οποίες μπορούν να χρησιμοποιηθούν για την υποστήριξη γρήγορων συνενώσεων και αποδοτικών λειτουργιών επιλογής δεδομένων. Σαν αποτέλεσμα, το Datix πετυχαίνει να εκτελεί σε λίγα λεπτά ερωτήματα που απαιτούσαν έως και μέρες χρησιμοποιώντας τις υπάρχουσες τεχνολογίες κεντρικής επεξεργασίας. Επίσης παρουσιάζει έως και 70% μείωση χρόνου εκτέλεσης σε σχέση με αντίστοιχες δημοφιλείς πλατφόρμες κατανεμημένης επεξεργασίας, όπως το Hive και το Shark.

Abstract

The pace at which data are described, queried and exchanged, using unstructured data representations, is constantly growing. Semantic Web technologies have emerged as one of the prevalent unstructured data sources. Utilizing the RDF description model, they attempt to encode and make openly available various World Wide Web datasets. Therefore, the constantly increasing volume of available data calls for efficient and scalable solutions for their management. In this thesis, we devise distributed algorithms and techniques for data management, which can scale and handle huge datasets. We introduce H2RDF+, a fully distributed RDF store that combines the MapReduce processing framework with a NoSQL distributed database. Creating 6 indexes over HBASE tables, H2RDF+ can process complex queries making adaptive decisions on both the join ordering and the join execution. Joins are executed using in distributed or centralized resources, depending on their cost. Furthermore, we present a novel system that addresses graph-based, workload-adaptive indexing of large RDF graphs by caching SPARQL query results. At the heart of the system lies a SPARQL query canonical labelling algorithm that is used to uniquely index and reference SPARQL query graphs as well as their isomorphic forms. We integrate our canonical labelling algorithm with a dynamic programming planner in order to generate the optimal join execution plan, examining the utilization of both primitive triple indexes and cached query results. By monitoring cache requests, our system is able to identify and cache SPARQL queries that, even if not explicitly issued, greatly reduce the average response time of a workload. The proposed cache is modular in design, allowing integration with different RDF stores.

Another ever-increasing source of unstructured data is the Internet traffic. Network datasets collected at large networks such as Internet Exchange Points (IXPs) can be in the order of Terabytes per hour. To handle analytics over such datasets, we present Datix, a fully decentralized, open-source analytics system for network traffic data that relies on smart partitioning storage schemes to support fast join algorithms and efficient execution of filtering queries. In brief, Datix manages to efficiently answer queries within minutes compared to more than 24 hours processing when executing existing Python-based code in single node setups. Datix also achieves nearly 70% speedup compared to baseline query implementations of popular big data analytics engines such as Hive and Shark.

ΚΕΦΑΛΑΙΟ 1

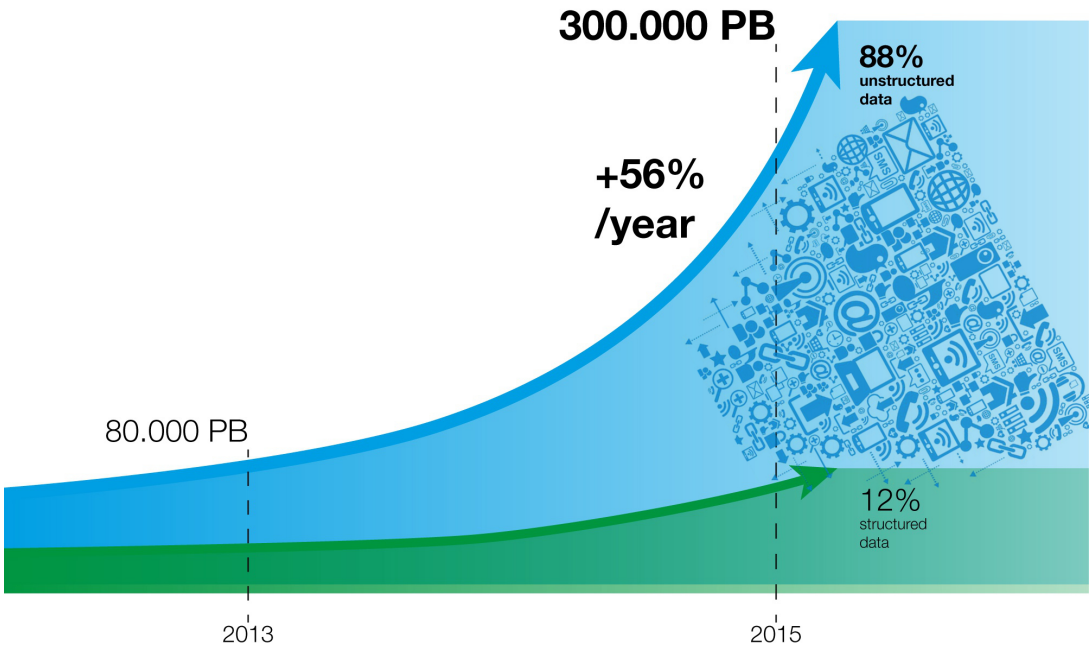
Εισαγωγή

Αυτή η διατριβή προτείνει αποδοτικούς αλγορίθμους που επιτρέπουν την κατανεμημένη ευρετηρίαση και επερώτηση μη δομημένων ή ήμι-δομημένων δεδομένων.

1.1 Κίνητρο

Το ποσό των δημοσίως και ιδιωτικώς διαθέσιμων δεδομένων έχει αυξηθεί ραγδαία τα τελευταία χρόνια [Manyika 11]. Εταιρείες και οργανισμοί καταγράφουν τρισεκατομμύρια bytes πληροφορίας σχετικά με τους πελάτες, τους προμηθευτές και τις υπηρεσίες τους. Την ίδια στιγμή, διασυνδεδεμένες συσκευές όπως έξυπνα τηλέφωνα, αισθητήρες, κ.λπ. συμβάλλουν στον "κατακλυσμό δεδομένων". Η ανάλυση τέτοιων μεγάλων σε όγκο συνόλων δεδομένων, τα ονομαζόμενα και "big data", αναδεικνύεται ως ένα βασικό στοιχείο ανταγωνισμού και υποστηρίζει νέα κύματα αύξησης παραγωγικότητας και καινοτομίας. Πράγματι, δεν αποτελεί έκπληξη ότι, οι εξατομικευμένες υπηρεσίες, η αύξηση της δημοτικότητας των πολυμέσων, των κοινωνικών δικτύων και του Διαδικτύου των πραγμάτων(Internet of Things) θα πυροδοτήσουν εκθετική αύξηση των δεδομένων στο εγγύς μέλλον [Lohr 12].

Παραδοσιακά, οι πλατφόρμες διαχείρισης δεδομένων σχεδιάζονταν με βάση δομημένα δεδομένα. Το σχεσιακό μοντέλο [Codd 70] έχει οδηγήσει την έρευνα και την καινοτομία στα συστήματα διαχείρισης δεδομένων για τουλάχιστον 4 δεκαετίες. Ωστόσο, τον τελευταίο καιρό μια σειρά επαναστατικών τεχνολογιών και εφαρμογών αμφισβητούν το status quo όσον αφορά την παραγωγή, τη διαχείριση, την ανάλυση και την αξιοποίηση των δεδομένων. Αντί να σπανίζουν,



Σχήμα 1.1: *Αύξηση μεγέθους των μη δομημένων δεδομένων*

να είναι εξειδικευμένα, ιδιωτικά και περιορισμένης πρόσβασης, τα δεδομένα πλέον παράγονται σε μεγάλες ποσότητες, έχουν μεγάλη ποικιλία, αποθηκεύονται με χαμηλό κόστος και είναι όλο και πιο ανοικτά και προσβάσιμα. Σε αυτό το εξαιρετικά πολύπλοκο και ποικιλόμορφο τοπίο δεδομένων, τα ολοένα αυξανόμενα σε μέγεθος δεδομένα παράγονται σε πολλαπλές μορφές και δεν διαθέτουν ενιαία και προκαθορισμένη δομή.

Το Σχήμα 1.1 απεικονίζει τις πρόσφατες τάσεις στην ανάπτυξη των δεδομένων. Είναι ευδιάκριτο ότι το συνολικό ποσό των μη δομημένων δεδομένων έχει ξεπεράσει το αντίστοιχο ποσό των δομημένων δεδομένων. Μόνο ένα μικρό ποσοστό της τάξεως του 12% των υφιστάμενων δεδομένων είναι δομημένο. Επιπλέον, η ποσότητα των μη δομημένων δεδομένων αυξάνεται εκθετικά παρουσιάζοντας ρυθμό αύξησης 56% ανά έτος. Τα μη δομημένα δεδομένα είναι θεμελιωδώς διαφορετικά από τα αντίστοιχα δομημένα και πρέπει να αντιμετωπιστούν κατάλληλα για χρήση σε εφαρμογές ανάλυσης δεδομένων.

Τα υπολογιστικά νέφη και η κατανεμημένη επεξεργασία αναδύονται ως οι κύριες τεχνολογίες που προσπαθούν να αντιμετωπίσουν την πρόκληση των “μεγάλων δεδομένων”. Ενώ τα κατανεμημένα συστήματα αξιοποιούν τη δύναμη των αποκεντρωμένων πόρων, τα υπολογιστικά νέφη έχουν γίνει το βασικό παράδειγμα δέσμευσης πόρων και πληρωμής ανάλογα με τις ανάγκες, μετατρέποντας το σχεδιασμό και τη φιλοσοφία του συνόλου της βιομηχανίας

πληροφορικής. Προγραμματιστές με καινοτόμες ιδέες μπορούν άμεσα να υλοποιήσουν την εφαρμογή τους χωρίς να ανησυχούν για την επένδυση μεγάλων κεφαλαίων για την αγορά και συντήρηση υπολογιστικών πόρων. Με αυτή την τεχνική μπορούν να αποφεύγονται οι υπερβολικές δαπάνες για μια υπηρεσία που δεν πιάνει τις προσδοκίες, καθώς και η έλλειψη πόρων για μια υπηρεσία που γίνεται γρήγορα δημοφιλής. Ως εκ τούτου, αυτή η ελαστικότητα των πόρων έχει αναδειχθεί ως το μεγαλύτερο πλεονέκτημα του μοντέλου των υπολογιστικών νεφών.

1.2 Σημασιολογικός Ιστός

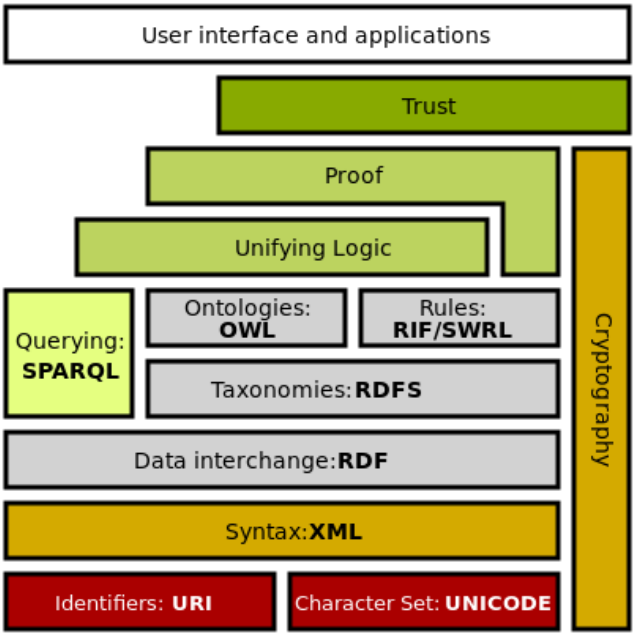
Μία από τις μεγαλύτερες πηγές δεδομένων στις μέρες μας είναι ο παγκόσμιος Ιστός (World Wide Web), ένας χώρος πληροφόρησης, όπου πληροφορίες μπορούν να μοιραστούν, να διασυνδεθούν και να προσπελαστούν μέσω του Διαδικτύου. Η παραγωγή καθώς και η κατανάλωση πληροφοριών στον παγκόσμιο Ιστό κινείται προς την ιδέα του "Σημασιολογικού Ιστού". Ο "Σημασιολογικός Ιστός" περιγράφει μια επέκταση του παγκόσμιου Ιστού, όπου οι υπολογιστές θα είναι σε θέση να καταλάβουν και να επεξεργαστούν αποτελεσματικά τα δεδομένα ακριβώς όπως και οι άνθρωποι. Το όραμα αυτό εκφράστηκε αρχικά από τον Tim Berners-Lee ο οποίος δήλωσε τα εξής:

> *"Έχω ένα όνειρο για τον παγκόσμιο Ιστό, στο οποίο οι υπολογιστές γίνονται ικανοί να αναλύουν όλα τα δεδομένα του: το περιεχόμενο, τις συνδέσεις καθώς και τις συναλλαγές μεταξύ ανθρώπων και υπολογιστών."*

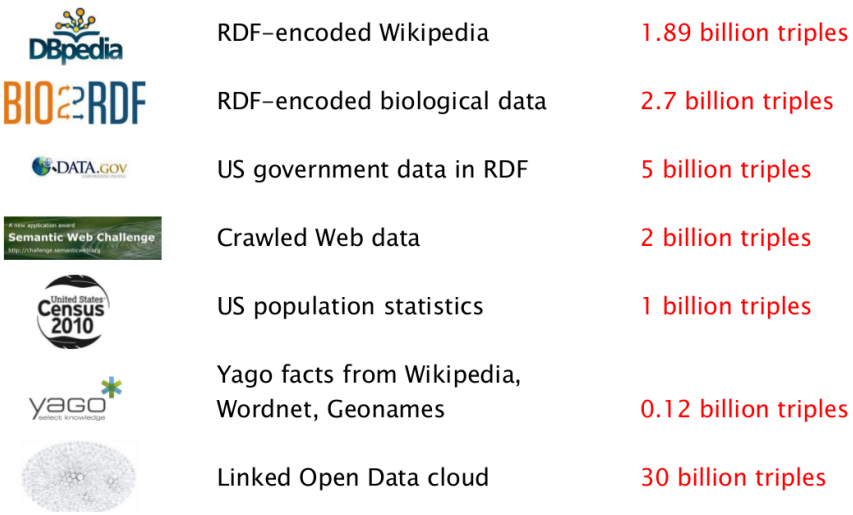
Η καινούργια αυτή εποχή έχει ως στόχο την αναπαράσταση και διασύνδεση δεδομένων που προέρχονται από διαφορετικές κοινότητες χρηστών, εφαρμογές και επιχειρήσεις. Ο όρος "Σημασιολογικός Ιστός" χρησιμοποιείται συχνά και για να αναφερθούμε στις τεχνολογίες που μπορούν να υλοποιήσουν τις παραπάνω ιδέες. Το Σχήμα 1.2 απεικονίζει τις διάφορες τεχνολογίες που έχουν παρουσιαστεί για την υλοποίηση του "Σημασιολογικού Ιστού". Αρχικά το πρότυπο RDF[1] έχει προταθεί για την αναπαράσταση των δεδομένων. Όπως αναφέρθηκε προηγουμένως, οι μορφές αναπαράστασης δεδομένων που εξαρτώνται από σχήματα, όπως η παραδοσιακή σχεσιακή αναπαράσταση, αποτυγχάνουν να προσαρμοστούν και να χειριστούν την ποικιλομορφία και συνεχώς μεταβαλλόμενη φύση των δεδομένων του Διαδικτύου. Ως αποτέλεσμα, η κοινότητα του Σημασιολογικού Ιστού έχει αναγνωρίσει το πρότυπο RDF [Manola 04] μαζί με την SPARQL γλώσσα ερωτήσεων [Prud'hommeaux 06] ως της βασικές τεχνολογίες της.

Η απομάκρυνση του σχήματος από την περιγραφή των δεδομένων RDF επιτρέπει τον σχηματισμό ενός κοινού πλαισίου που διευκολύνει την ενσωμάτωση διαφόρων πηγών δεδομένων.

[1] https://www.w3.org/RDF/



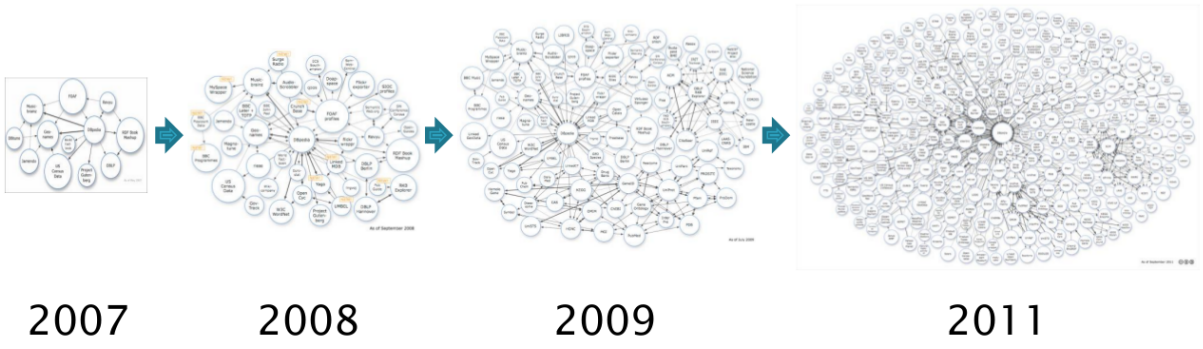
Σχήμα 1.2: *Τεχνολογίες Σημασιολογικού Ιστού*



Σχήμα 1.3: *Σύνολα Δεδομένων Σημασιολογικού Ιστού*

Αυτή η ιδιότητα έχει οδηγήσει σε μια άνευ προηγουμένου αύξηση του ρυθμού με τον οποίο παράγονται, αποθηκεύονται και ερωτώνται RDF δεδομένα, ακόμα και έξω από το καθαρά ακαδημαϊκό χώρο. Κατά συνέπεια τα τελευταία χρόνια παρακολουθούμε την ανάπτυξη πολλών μηχανών [Weiss 08, Neumann 10a, Zeng 13, Atre 08, Bonstrom 03] που στοχεύουν την αποθήκευση και ευρετηρίαση των RDF δεδομένων καθώς και την αποτελεσματική επεξεργασία

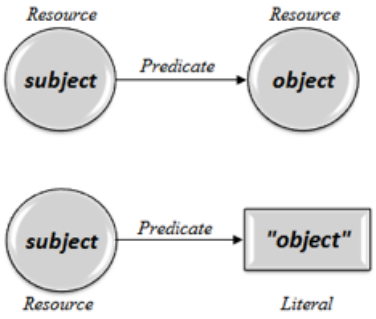
SPARQL ερωτημάτων. Το Σχήμα 1.3 περιέχει μια λίστα με τα πιο δημοφιλή δημοσίως διαθέσιμα σύνολα RDF δεδομένων. Για να παρουσιάσουμε μια πιο αναλυτική εικόνα της ανάπτυξης του "Σημασιολογικού Ιστού", το Σχήμα 1.4 δείχνει την εξέλιξη των δημοσίως διαθέσιμων RDF συνόλων δεδομένων καθώς και των διασυνδέσεών τους.



Σχήμα 1.4: *Εξέλιξη των linked data*

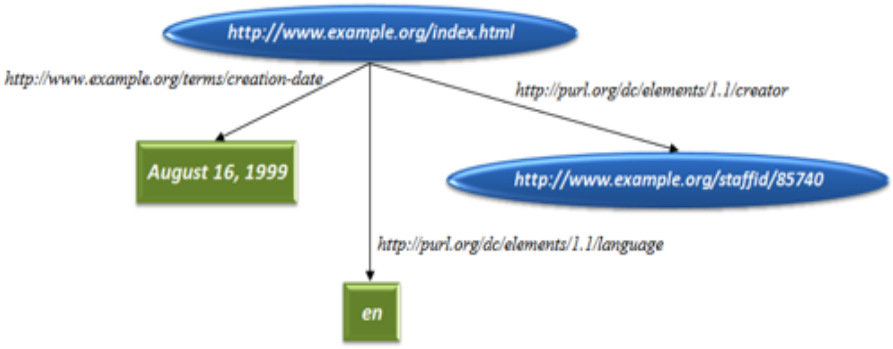
1.2.1 Πρότυπο Δεδομένων RDF

Το Resource Description Framework (RDF) είναι ένα ευέλικτο μοντέλο δεδομένων που εκφράζει μη δομημένες πληροφορίες και συσχετίσεις. Στην ουσία, το μοντέλο δεδομένων RDF αναπαρίσταται φυσικά ως ένας κατευθυνόμενος γράφος με ετικέτες, όπου οι κόμβοι αντιστοιχούν σε πόρους ή λεκτικές εκφράσεις και οι ακμές χρησιμοποιούνται για να περιγράψουν τις σχέσεις μεταξύ των πόρων. Κάθε πληροφορία RDF αναπαριστάται σαν μια τριάδα που έχει τη μορφή (subject, predicate, object), δηλαδή σαν εκφράσεις που περιέχουν υποκείμενο, ρηματική έκφραση και αντικείμενο. Βλέποντας τα RDF δεδομένα σαν γράφο, κάθε τριπλέτα αντιστοιχεί σε μια κατευθυνόμενη ακμή που περιέχει μια ετικέτα. Το Σχήμα 1.5 δείχνει μια οπτική αναπαράσταση μιας RDF τριάδας.



Σχήμα 1.5: *RDF τριπλέτες*

Οι ξεχωριστές RDF τριπλέτες συνδυάζονται για τη δημιουργία ενός γράφου γνώσης ο οποίος περιέχει όλη την πληροφορία που παρέχεται από τις επιμέρους τριπλέτες. Αυτό προϋποθέτει ότι όλοι οι πόροι που περιγράφονται έχουν ένα μοναδικό αναγνωριστικό πόρου (URI). Αυτή η μοναδική αναπαράσταση των πόρων είναι μια κεντρική ιδέα για το όραμα του "Σημασιολογικού Ιστού", καθώς επεκτείνει την μοναδική ταυτοποίηση των ιστοσελίδων, που υπάρχει στην τρέχουσα έκδοση του Web, σε κάθε πόρο του φυσικού ή ψηφιακού κόσμου.



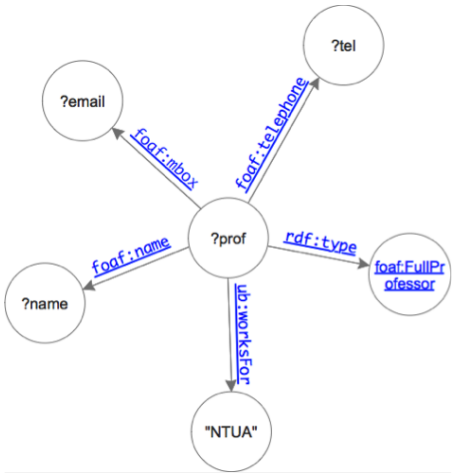
Σχήμα 1.6: *Γράφος Γνώσης*

Για παράδειγμα ο γράφος γνώσης του Σχήματος 1.6 περιέχει πληροφορίες για τον δημιουργό, τη γλώσσα και την ημερομηνία δημιουργίας της ιστοσελίδας
http://www.example.org/index.html.

1.2.2 Γλώσσα επερώτησης SPARQL

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 SPARQL βασίζεται στο ταίριασμα υπογράφων για την εκτέλεση ερωτημάτων. Ως εκ τούτου, τα ερωτήματα σε αυτό το πλαίσιο έχουν τη μορφή γράφου ο οποίος στην ουσία περιέχει δεσμευμένες ή μεταβλητές ετικέτες στους κόμβους και τις ακμές του. Για παράδειγμα, το Σχήμα 1.7 απεικονίζει ένα ερώτημα SPARQL που ανακτά όλους τους καθηγητές που εργάζονται για το ΕΜΠ μαζί με τα ονόματά τους, τα mail και τους αριθμούς τηλεφώνου τους.

Οι κόμβοι με ονόματα που ξεκινούν με "?" αποτελούν τις μεταβλητές του ερωτήματος. Μια μηχανή επερώτησης SPARQL πρέπει να βρει όλους τους υπογράφους του RDF γράφου γνώσης που ταιριάζουν με το συγκεκριμένο ερώτημα. Επίσης, η SPARQL παρέχει πιο γενικές δυνατότητες όπως είναι η χρήση προαιρετικών δομών γράφων, μονοπατιών ιδιοτήτων καθώς και ομαδοποίησης και διάταξης των αποτελεσμάτων.



Σχήμα 1.7: *SPARQL ερώτημα*

1.2.3 Συνεισφορές

Η πρώτη συμβολή αυτής της διατριβής, είναι το H_2RDF+ (Κεφάλαιο 2), μια πλήρως κατανεμημένη, βασισμένη σε τεχνικές υπολογιστικών νεφών, RDF βάση δεδομένων που συνδυάζει το πλαίσιο επεξεργασίας MapReduce [Dean 08] με τη NoSQL κατανεμημένη βάση δεδομένων HBase [Chang 08]. Το H_2RDF+ δημιουργεί πολλαπλά RDF ευρετήρια και εκτελεί κατανεμημένους αλγορίθμους Merge και Sort-Merge συνενώσεων. Βασίζεται σε τεχνικές συμπίεσης επιπέδου byte, γρήγορες σαρώσεις των δεδομένων και στην ομαδοποίηση των αποτελεσμάτων για να μπορέσει να επεξεργαστεί σύνθετα και επιλεκτικά ερωτήματα με αποτελεσματικό τρόπο. Επιπλέον, επιλέγει προσαρμοστικά το μοντέλο εκτέλεσης, κεντρικό ή κατανεμημένο, με βάση εκτιμήσεις για την πολυπλοκότητα των ερωτημάτων. Η πειραματική αξιολόγηση αποδεικνύει ότι το H_2RDF+ υπερέχει σε περίπλοκα και μη επιλεκτικά ερωτήματα, κλιμακώνει γραμμικά με τον αριθμό των διαθέσιμων πόρων, επιτυγχάνοντας παράλληλα διαδραστική, συγκρίσιμη με κεντρικές RDF μηχανές, εκτέλεση για επιλεκτικά ερωτήματα.

Ωστόσο, η μετάβαση από τις εξαρτώμενες στο σχήμα σχεσιακές βάσεις στα δεδομένα RDF που δεν περιέχουν αντίστοιχες πληροφορίες, έφερε νέες προκλήσεις για την αποδοτική ευρετηρίαση και την αναζήτηση των RDF δεδομένων. Ευρέως χρησιμοποιούμενες τεχνικές που βασίζονταν στο σχήμα των δεδομένων δεν μπορούν εύκολα να επεκταθούν στο πλαίσιο RDF. Για παράδειγμα:

- Η ομαδοποίηση των δεδομένων με χρήση πινάκων.
- Η ευρετηρίαση με βάση πεδία που χρησιμοποιούνται συχνά σε φίλτρα ή σε συνενώσεις.
- Η δημιουργία όψεων με βάση συχνά χρησιμοποιούμενα μοτίβα ερωτημάτων.

Στην πραγματικότητα, οι βάσεις δεδομένων RDF θεωρούν ότι υπάρχει περιορισμένη γνώση της δομής των δεδομένων. Αυτό συμβαίνει συνήθως μέσω RDFS τριάδων [Brickley 14]. Ωστόσο, οι RDFS πληροφορίες δεν είναι τόσο πλούσιες και υποχρεωτικές όσο το SQL σχήμα. Μπορεί να είναι ελλιπείς και αλλάζουν γρήγορα, μαζί με τα δεδομένων. Ως εκ τούτου, οι περισσότερες βάσεις δεδομένων RDF στοχεύουν την ευρετηρίαση των RDF τριάδων. Αυτό έχει ως αποτέλεσμα την χρήση πολύ μεγαλύτερου αριθμού συνενώσεων σε σχέση με τις αντίστοιχες σχεσιακές βάσεις. Για παράδειγμα, το ερώτημα 2 του TPC-H γραμμένο σε SPARQL απαιτεί 26 συνενώσεις ενώ το αντίστοιχο SQL περιέχει μόνο 5 [Gubichev 14]. Σε αντίθεση, οι RDF βάσεις δεδομένων που χρησιμοποιούν RDFS πληροφορίες για την αποθήκευση και την ομαδοποίηση των δεδομένων [Stuckenschmidt 04, Tran 10], αδυνατούν να προσαρμοστούν αποτελεσματικά στις αλλαγές σχήματος και σε δεδομένα που δεν ταιριάζουν στο σχήμα.

Σημειώνουμε ότι οι πιο εξελιγμένες βάσεις γράφων [Zhao 07, Yan 04] χρησιμοποιούν εκτενώς τεχνικές που στοχεύουν στην ευρετηρίαση συχνά εμφανιζόμενων μοτίβων γράφων. Ωστόσο, οι RDF βάσεις δεδομένων δεν έχουν λάβει ακόμη τα πλεονέκτημα αυτών των τεχνικών. Επιπροσθέτως, τα συστήματα αυτά επικεντρώνονται στην *στατική* ευρετηρίαση των σημαντικών μοτίβων γραφημάτων, δηλαδή η εύρεση τέτοιων μοτίβων βασίζεται αποκλειστικά στο υπό επεξεργασία σύνολο δεδομένων, χωρίς καμία μέριμνα για το σύνολο ερωτημάτων. Ωστόσο, η ποικιλομορφία των εφαρμοζόμενων SPARQL ερωτημάτων σε συνδυασμό με την απαίτηση για υψηλές επιδόσεις σε όλα τα διαφορετικά σύνολα ερωτημάτων, καλεί για μια δυναμική, βασισμένη στα ερωτήματα ευρετηρίαση (π.χ., [Idreos 11]).

Επιπλέον, τα ερωτήματα SPARQL τείνουν να αυξάνουν σε μέγεθος και πολυπλοκότητα παρουσιάζοντας προκλήσεις στη βελτιστοποίηση τους με χρήση αλγορίθμων δυναμικού προγραμματισμού. Για παράδειγμα, η DBpedia αναφέρει ότι τα SPARQL ερώτημα που εκτελούνται περιέχουν ερωτήσεις με έως και 10 τριπλέτες [Gallego 11]. Ερωτήματα στον τομέα της βιοϊατρικής μπορεί να περιλαμβάνουν περισσότερες από 50 τριπλέτες [Sahoo 10]. Τα πειραματικά αποτελέσματα δείχνουν ότι η εξεύρεση του βέλτιστου πλάνου εκτέλεσης χρησιμοποιώντας προσεγγίσεις δυναμικού προγραμματισμού μπορεί να παρουσιάζει απαγορευτικό χρόνο εκτέλεσης για ερωτήματα με περισσότερες από 15 τριπλέτες [Neumann 10a, Gubichev 14]. Ενώ υπάρχουν πολλοί άπληστοι και ευριστικοί αλγόριθμοι για την βελτιστοποίηση SPARQL ερωτημάτων [Tsialiamanis 12, Papailiou 13], μπορούν να επιλέξουν μη βέλτιστα πλάνα εκτέλεσης, αυξάνοντας το χρόνο εκτέλεσης του ερωτήματος [Gubichev 14] και δεν μπορούν εύκολα να ενσωματωθούν με αλγορίθμους κρυφής μνήμης που εξετάζουν *όλα* τα αποτελέσματα που μπορούν να χρησιμοποιηθούν για την εκτέλεση ενός ερωτήματος.

Ως αποτέλεσμα, η δεύτερη συνεισφορά αυτής της διατριβής (Κεφάλαιο 3) είναι η προσαρμοστική ευρετηρίαση και η δημιουργία κρυφής μνήμης SPARQL αποτελεσμάτων που μπορεί να χρησιμοποιηθεί προκειμένου να εντοπιστούν συχνά εμφανιζόμενες μορφές ερωτημάτων.

Προτείνουμε ένα πλαίσιο κρυφής μνήμης που προσφέρει τη δυνατότητα επαναχρησιμοποίησης των υπολογιζόμενων αποτελεσμάτων SPARQL. Παρακολουθώντας ενεργά το φόρτο εργασίας SPARQL, το σύστημά μας μπορεί να ανιχνεύσει συχνές μορφές ερωτημάτων και να προκαλέσει την εκτέλεση και προσωρινή αποθήκευση τους, προκειμένου να ενισχύσει την απόδοση των επακόλουθων ερωτημάτων και την προσαρμογή στο φόρτο εργασίας. Επίσης παρουσιάζουμε έναν αλγόριθμο απλοποίησης SPARQL ερωτημάτων, που χρησιμοποιείται για να χειριστεί πολύπλοκα γραφήματα. Εν συντομία, η διατριβή αυτή προτείνει:

- Έναν αλγόριθμο απλοποίησης SPARQL ερωτημάτων (Ενότητα 3.2), με βάση τεχνικές απλοποίησης αστέρων, που χωρίζει το ερώτημα σε έναν σκελετό και πολλαπλά γραφήματα αστέρων.

- Έναν αλγόριθμο δημιουργίας κανονικοποιημένων ετικετών για SPARQL ερωτήματα (κεφάλαιο 3.3) που είναι σε θέση να παράγει κανονικοποιημένων ετικέτες που είναι ίδιες για όλες τις ισομορφικές αναπαραστάσεις ενός ερωτήματος SPARQL. Αυτές οι ετικέτες χρησιμοποιούνται ως κλειδιά για την δημιουργία μιας κρυφής μνήμης SPARQL ερωτημάτων. Επιπλέον, η τεχνική απλούστευσης των ερωτημάτων καταφέρνει να μειώσει αποτελεσματικά την πολυπλοκότητα του αλγορίθμου δημιουργίας κανονικοποιημένων ετικετών για πολύπλοκες δομές ερωτημάτων.

- Επεκτείνουμε ένα *Βελτιστοποιητή Δυναμικού Προγραμματισμού* [Moerkotte 08] προκειμένου να πραγματοποιεί αιτήματα κρυφής μνήμης για όλους τους υπογράφους του ερωτήματος, χρησιμοποιώντας *κανονικοποιημένες ετικέτες* (Ενότητα 3.4). Το προκύπτων βέλτιστο πλάνο εκτέλεσης μπορεί έτσι να περιλαμβάνει, εν μέρει προσωρινά αποθηκευμένα αποτελέσματα υπογράφων.

- Ένας *Ελεγκτής Κρυφής Μνήμης* παρακολουθεί όλα τα αιτήματα κρυφής μνήμης επιτρέποντας την ανίχνευση *κερδοφόρων* μορφών ερωτημάτων. Ο ελεγκτής προκαλέσει την εκτέλεση και την προσωρινή αποθήκευση τέτοιων ερωτημάτων για να βελτιώσει τη χρησιμοποίηση της κρυφής μνήμης.

Η προτεινόμενη κρυφή μνήμη έχει αρθρωτό σχεδιασμό, επιτρέποντας την ενσωμάτωση με διάφορες RDF βάσεις δεδομένων. Ενσωματώνοντάς την στο H_2RDF+ αποδεικνύουμε ότι η προσαρμοστική δημιουργία κρυφή μνήμης μπορεί να μειώσει το μέσο χρόνο απόκριση των SPARQL ερωτημάτων έως και δύο τάξεις μεγέθους, προσφέροντας μικρούς χρόνους απόκρισης για σύνθετα σύνολα ερωτημάτων και μεγάλες συλλογές RDF δεδομένων.

1.3 Ανάλυση Δεδομένων Δικτύων

Μια ακόμα πηγή μεγάλου όγκου μη δομημένων δεδομένων είναι και το Διαδίκτυο που έχει γίνει το κυρίαρχο κανάλι για την καινοτομία, το εμπόριο, και την ψυχαγωγία. Τόσο η κίνηση όσο και η διείσδυση του Διαδικτύου αυξάνεται με ρυθμό που καθιστά δύσκολο να παρακολουθείτε η ανάπτυξη και οι τάσεις του με έναν συστηματικό και επεκτάσιμο τρόπο. Πράγματι, πρόσφατες μελέτες δείχνουν ότι η κίνηση στο Διαδίκτυο αυξάνεται με ρυθμό μεγαλύτερο από 30% σε ετήσια βάση, όπως συμβαίνει και τα τελευταία 20 χρόνια. Αυτή η ανάπτυξη αναμένεται να συνεχιστεί με τον ίδιο ρυθμό και στο μέλλον [Cisco 13]. Ωστόσο, οι φορείς εκμετάλλευσης και οι διαχειριστές του αναγκάζονται να εκτελούν μεγάλης κλίμακας αναλύσεις πάνω στα δεδομένων του για τη βελτιστοποίηση παραμέτρων που αφορούν την δρομολόγηση, διαστασιολόγηση, και ασφάλεια του δικτύου.

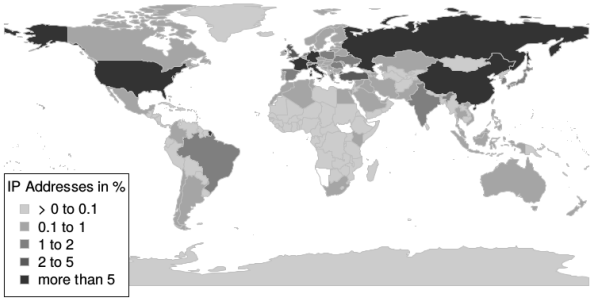
Οι κόμβοι ουδέτερης διασύνδεσης (Internet eXchange Points - IXPs) είναι φυσικός εξοπλισμός (routers, switches κ.λπ.) που επιτρέπει την άμεση διασύνδεση παρόχων υπηρεσιών Internet (Internet Service Providers - ISPs) με σκοπό την ανταλλαγή κίνησης δεδομένων Internet μεταξύ των δικτύων τους (Autonomous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

Προφανώς, ένας IXP δρομολογεί κίνηση τάξης μεγέθους μεγαλύτερη σε σχέση με ένα συγκεκριμένο ISP. Μερικοί από τους πιο επιτυχημένους IXPs, συνδέουν περισσότερα από 600 δίκτυα και μεταφέρουν δεδομένα πολλαπλών TB ανά δευτερόλεπτο. Πιο συγκεκριμένα, κατά τη διάρκεια μιας μέσης εργάσιμης ημέρας του 2013, ένας από τους μεγαλύτερους IXPs, ο AMS-IX στο Άμστερνταμ, μετέφερε περίπου 25 PB ενώ η AT&T και η Deutsche Telekom μετέφεραν 33 PB και 16 PB δεδομένων αντίστοιχα [Chatzis 13a]. Πολλές σύγχρονες μελέτες έχουν δείξει ότι η δειγματοληπτική ανάλυση της κίνησης ενός IXP σε ένα βάθος χρόνου μερικών εβδομάδων μπορεί να εξάγει ενδιαφέροντα συμπεράσματα όχι μόνο για τα συγκεκριμένα AS που διασυνδέει, αλλά και για την κατάσταση ολόκληρου του διαδικτύου. Χρησιμοποιώντας το εργαλείο sFlow[2], οι προγούμενες μελέτες συγκέντρωσαν ένα δείγμα των πακέτων που δρομολογήθηκαν. Η ανάλυση των δειγμάτων απέδειξε ότι ένα IXP έχει τέλεια "ορατότητα" του συνολικού διαδικτύου, καθώς μέσα από αυτό περνάει κίνηση προς όλα σχεδόν τα υπάρχοντα AS και για όλα τα προθέματα δημόσιων δικτύων [Chatzis 13b].

Οι προηγούμενες μελέτες [Chatzis 13b] χρησιμοποίησαν παραδοσιακές κεντρικές τεχνικές επεξεργασίας των δεδομένων που συγκεντρώνονταν δειγματοληπτικά από τους IXPs. Είναι προφανές ότι η ισχύς ενός μεμονωμένου υπολογιστή τοποθετεί ένα όριο στις δυνατότητες κλιμάκωσης της επεξεργασίας σε μεγαλύτερο αριθμό δεδομένων. Επίσης, το μέγιστο δυνατό

[2] http://www.sflow.org/

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



Σχήμα 1.8: *Στατιστικά στοιχεία που προέρχονται από την παρακολούθηση της κίνησης ενός IXP*

μέγεθος των δεδομένων προς ανάλυση περιορίζεται από τις δυνατότητες του συγκεκριμένου υπολογιστή (συνήθως επηρεάζεται από το μέγεθος της φυσικής μνήμης του μηχανήματος).

1.3.1 Συνεισφορές

Το τελευταίο κεφάλαιο αυτής της διατριβής (Κεφάλαιο 4) παρουσιάζει το Datix: μια κλιμακώσιμη πλατφόρμα επεξεργασίας και ανάλυσης δεδομένων κίνησης δικτύων. Το σύστημά μας βασίζεται σε κατανεμημένες τεχνικές αποθήκευσης και επεξεργασίας δεδομένων, όπως το MapReduce [Dean 08] και είναι σε θέση να λύσει το πιο γενικό πρόβλημα της επεξεργασίας log αρχείων όπως περιγράφεται στο [Blanas 10]. Στόχος μας είναι η αποτελεσματική εκτέλεση κατανεμημένων συνενώσεων για το συνδυασμό μια κύριας πηγής πληροφορίας (log file), που στη δική μας περίπτωση είναι τα sFlow δεδομένα που συλλέγονται σε έναν IXP, με συμπληρωματικές πληροφορίες που παρέχονται από δευτερεύοντα σύνολα δεδομένων, όπως η χαρτογράφηση των διευθύνσεων IP, η αντιστοίχηση των IP με τα AS το DNS τους κτλ. [Durumeric 13]. Οι βασικές συνεισφορές αυτής της εργασίας είναι οι εξής:

- Παρουσιάζεται ένας έξυπνος τρόπος για pre-partitioning των δεδομένων ανάλογα με τις εγγραφές που περιέχουν, για να μπορεί να γίνει πιο αποδοτικά η επεξεργασία τους κατά τη διαδικασία της συνένωσης.

- Προτείνουμε μια αποδοτική υλοποίηση για τον αλγόριθμο συνένωσης της πληροφορίας, που κάνει χρήση της τεχνικής του map join [Blanas 10] και εξειδικευμένων συναρτήσεων UDF.

- Συνδυάζοντας τα δύο παραπάνω μπορούμε να εκτελέσουμε ειδικότερα ερωτήματα που αφορούν ένα περιορισμένο τμήμα των δεδομένων σε αρκετά μικρό χρονικό διάστημα έως μερικά λεπτά, όταν τα ερωτήματα που αφορούν όλο το dataset μπορεί να απαιτούν μερικές ώρες σε κάποιες περιπτώσεις.

1.4 Οργάνωση κειμένου

Το υπόλοιπο αυτού του εγγράφου οργανώνεται ως εξής. Το Κεφάλαιο 2 περιγράφει το σχεδιασμό και την υλοποίηση του H_2RDF+. Η Ενότητα 2.1 εισάγει και παρακινεί τη δημιουργία του H_2RDF+. Η Ενότητα 2.2 περιγράφει την αρχιτεκτονική του H_2RDF+. Η Ενότητα 2.3 συζητά τις αποφάσεις ευρετηρίασης και αποθήκευσης. Η Ενότητα 2.5 αναλύει τους χρησιμοποιούμενους αλγόριθμους συνενώσεων. Η Ενότητα 2.6 παρουσιάζει την βελτιστοποίηση SPARQL ερωτημάτων. Τέλος η Ενότητα 2.7 περιέχει μια εκτενή πειραματική αξιολόγηση του H_2RDF+.

Το Κεφάλαιο 3 παρουσιάζει την υλοποίηση μρυφής μνήμης για SPARQL ερωτήματ. Η Ενότητα 3.2 περιγράφει τον αλγόριθμο απλοποίησης SPARQL ερωτημάτων. Η Ενότητα 3.3 αναλύει τον αλγόριθμο δημιουργίας κανονικοποιημένων ετικετών για SPARQL ερωτήματα. Η Ενότητα 3.4 εισάγει τον Δυναμικού Προγραμματισμού Βελτιστοποιητή ερωτημάτων. Η Ενότητα 3.6 συζητά τον Ελεγκτή Κρυφής Μνήμης. Η Ενότητα 3.7 παρουσιάζει την πειραματική αξιολόγηση του πλαισίου κρυφής μνήμης χρησιμοποιώντας το H_2RDF+.

Το Κεφάλαιο 4 παρουσιάζει το Datix με στόχο την αποδοτική επεξεργασία μεγάλου όγκου δεδομένων κίνησης δικτύων. Η Ενότητα 4.1 εισάγει το προτεινόμενο σύστημα. Στην Ενότητα 4.2 παρουσιάζονται οι κύριες αρχιτεκτονικές αποφάσεις μας. Επιπλέον, η Ενότητα 4.3 παρουσιάζει τους αλγόριθμους συνενώσεων που χρησιμοποιούνται στο Datix, ενώ το τμήμα 4.4 περιέχει μια λεπτομερή πειραματική αξιολόγηση του συστήματος.

Μια λεπτομερής έρευνα των σχετικών εργασιών τόσο για συστήματα επεξεργασίας RDF δεδομένων όσο και για την ανάλυση δεδομένων κίνησης δικτύων παρουσιάζεται στο κεφάλαιο 5. Τέλος, το κεφάλαιο 6 ολοκληρώνει τη διατριβή και παρουσιάζει τα βασικά της συμπεράσματα.

ΚΕΦΑΛΑΙΟ 2

Η κατανεμημένη βάση RDF δεδομένων H_2RDF+

2.1 Εισαγωγή

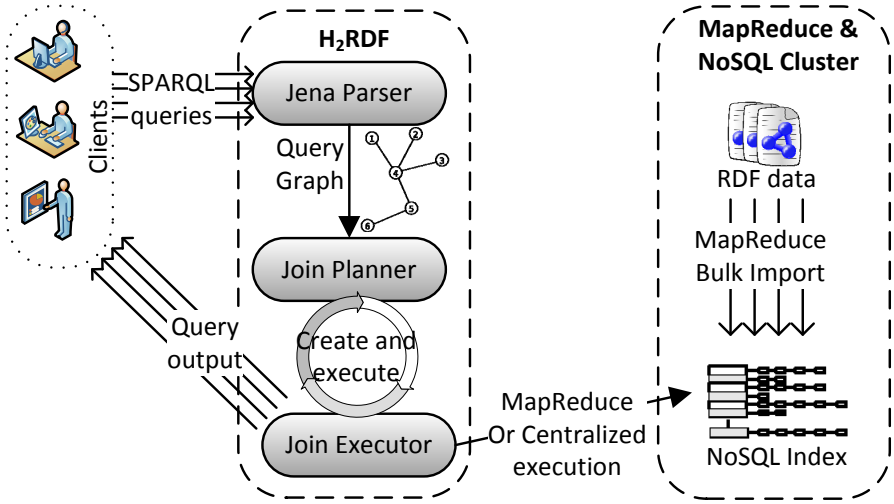
Οι κατανεμημένες RDF βάσεις δεδομένων κατανέμουν την επεξεργασία μερικών ή όλων των σταδίων της διαχείρισης των δεδομένων RDF. Ωστόσο, δεν μπορούν να προσαρμόσουν με ευέλικτο τρόπο τη συμπεριφορά τους σε σχέση με το κάθε ξεχωριστό ερώτημα. Έτσι, επικεντρώνονται σε ένα συγκεκριμένο είδος αλγορίθμων συνενώσεων ή καταλαμβάνουν όλους τους διαθέσιμους υπολογιστικούς πόρους της πλατφόρμας εκτέλεσής τους. Τα ερωτήματα SPARQL συχνά απαιτούν πολλαπλές συνενώσεις πάνω από ένα (πιθανόν μεγάλο) αριθμό από RDF τριπλέτες. Έτσι, οι RDF βάσεις δεδομένων θα πρέπει να προσαρμόζουν αποτελεσματικά την εκτέλεσή τους σε σχέση με το είδος και την πολυπλοκότητα του ερωτήματος. Η πολυπλοκότητα των συνενώσεων μεταβάλεται ανάλογα με τον αριθμό των δεδομένων και το είδος της συνένωσης. Παρ' όλα αυτά, η αύξηση του αριθμού των συνενώσεων καθώς και του μεγέθους των ενδιάμεσων αποτελεσμάτων δεν πρέπει να οδηγεί σε εκθετική αύξηση του χρόνους απόκρισης των RDF βάσεων δεδομένων.

Επιπλέον, η πλειοψηφία των κατανεμημένων προσεγγίσεων δεν έχει εκμεταλλευτεί ακόμη τα πλεονέκτημα της ευρετηρίασης όλων των διαφορετικών μεταθέσεων των στοιχείων RDF, δηλαδή *spo, pso, pos, ops, osp* και *sop* [Weiss 08]. Ένα τέτοιο σύστημα προσφέρει τα ακόλουθα πλεονεκτήματα: (1) Όλα τα μοτίβα SPARQL τριπλετών μπορούν να απαντηθούν αποτελεσματικά χρησιμοποιώντας μια σάρωση πάνω σε ένα συγκεκριμένο ευρετήριο. Για παράδειγμα,

μια τριπλέτα με δεσμευμένο subject και μεταβλητό predicate και object μπορεί να απαντηθεί χρησιμοποιώντας μια σάρωση του *spo* ή του *sop* ευρετηρίου. (2) Merge συνενώσεις που εκμεταλλεύονται τις προϋπολογισμένες ταξινομήσεις των RDF δεδομένων μπορούν να χρησιμοποιηθούν εκτενώς. Η ύπαρξη των έξι διαφορετικών ταξινομήσεων εγγυάται ότι κάθε ένωση μεταξύ μοτίβων τριπλετών μπορεί να γίνει χρησιμοποιώντας Merge συνενώσεις. Πιο ακριβοί αλγόριθμοι συνένωσης χρειάζονται μόνο όταν εξετάζουμε τη συνένωση ενδιάμεσων μη ταξινομημένων αποτελεσμάτων. Στην περίπτωση του H_2RDF+ [1], διατηρούμε τις δύο αυτές ιδιότητες, ενώ μεταβαίνουμε σε περιβάλλον κατανεμημένης και κλιμακούμενης εκτέλεσης. Συνοψίζουμε τις κύριες συνεισφορές αυτής της εργασίας ως εξής:

- Αναπτύξαμε ένα σύστημα ευρετηρίασης για την αποθήκευση των δεδομένων RDF με χρήση της HBase, που επιτρέπει την αποτελεσματική μαζική εισαγωγή και δημιουργία ευρετηρίων για μεγάλα σύνολα δεδομένων RDF. Βελτιστοποιήσαμε τις δυνατότητες ανάκτησης των κατανεμημένων ευρετηρίων μας, εφαρμόζοντας τεχνικές συμπίεσης και ελαχιστοποιώντας τις απαιτήσεις χώρου αποθήκευσης. Εφαρμόζουμε τεχνικές ομαδοποίησης που στοχεύουν στη μείωση του όγκου των ενδιάμεσων αποτελεσμάτων. Τα ευρετήρια μας αποζημιώνουν το γεγονός ότι είναι σχεδόν 10× πιο αργά από τα αντίστοιχα κεντρικά B^+δέντρα, επιτυγχάνοντας μεγάλη κλιμακωσιμότητα και δυνατότητες παράλληλης σάρωσης.
- Παρουσιάζουμε πλήρως κλιμακώσιμες, κατανεμημένες και βασισμένες στο MapReduce εκδοχές των multi-way Merge και Sort-Merge συνενώσεων.
- Σχεδιάσαμε ένα μοντέλο εκτίμησης κόστους που χρησιμοποιείται για την προσέγγιση του κόστους κάθε συνένωσης καθώς και για τη λήψη προσαρμοστικών αποφάσεων για τη σειρά και τον τρόπο εκτέλεσής τους (κεντρικό ή κατανεμημένο) .
- Εκτελέσαμε μια ενδελεχή πειραματική αξιολόγηση του συστήματος μας. Τα αποτελέσματα δείχνουν ότι το H_2RDF+ μπορεί να είναι τάξεις μεγέθους πιο γρήγορο από κεντρικές RDF βάσεις δεδομένων [Neumann 10a] για πολύπλοκα, μη-επιλεκτικά ερωτήματα, ενώ είναι μόλις δέκατα του δευτερολέπτου πιο αργό από αυτά σε επιλεκτικά ερωτήματα. Επιπλέον, αποδεικνύεται 6 έως 8 φορές πιο γρήγορο από την προηγούμενη έκδοση του [Papailiou 12] και μέχρι τάξεις μεγέθους πιο γρήγορο από μια εναλλακτικό, βασισμένο στο MapReduce σύστημα [Husain 11]. Το H_2RDF+ αποδεικνύεται ικανό να διαχειριστεί 14 δισεκατομμύρια τριπλέτες (2.5TB) χρησιμοποιώντας ένα σύμπλεγμα από 35 υπολογιστικούς κόμβους, παρέχοντας γραμμική κλιμακωσιμότητα σε σχέση με το μέγεθος των διαθέσιμων υπολογιστικών πόρων.

[1] https://github.com/npapa/h2rdf, http://h2rdf.googlecode.com



Σχήμα 2.1: *Αρχιτεκτονική του H_2RDF+*

2.2 Αρχιτεκτονική

Το Σχήμα 2.1 παρουσιάζει μια επισκόπηση της αρχιτεκτονικής του H_2RDF+ [Papailiou 13, Papailiou 14]. Το σύστημα χρησιμοποιεί την HBase [2] ως ένα κατανεμημένο υπόστρωμα ευρετηρίασης για μεγάλου όγκου RDF δεδομένα. Τα RDF δεδομένα εισάγονται σε πίνακες HBase μέσα από μια μαζική διαδικασία εισαγωγής MapReduce. Οι χρήστες έχουν τη δυνατότητα να θέσουν ερωτήματα SPARQL, η ορθότητα και η σύνταξη των οποίων αναλύεται με χρήση του Jena Parser [Carroll 04]. Ο *Join Planner* (Σχεδιαστής Πλάνου Εκτέλεσης) επιλέγει τη σειρά εκτέλεσης των συνενώσεων καθώς και για κάθε ξεχωριστή συνένωση τον αλγόριθμο που θα χρησιμοποιηθεί, τον τρόπο εκτέλεσης (κεντρικό ή κατανεμημένο) και τους απαιτούμενους πόρους. Το σύστημά μας είναι διαθέσιμο ως ένα έργο ανοικτού κώδικα[3] και προσφέρει ευρετηρίαση RDF δεδομένων και λειτουργικότητα SPARQL επερωτήσεων.

2.3 Σχήμα ευρετηρίασης

Στην ενότητα αυτή, θα εξηγήσουμε τις αποφάσεις που λάβαμε σχετικά με τον αριθμό και το είδος των ευρετηρίων που χρησιμοποιούνται στο σύστημά μας. Αν και οι περισσότερες κεντρικές RDF βάσεις δεδομένων χρησιμοποιούν συνδυασμούς Hexastore ευρετηρίων, οι κατανεμημένες προσεγγίσεις δεν έχουν λάβει ακόμη τα πλεονέκτημα αυτής της τεχνικής για να προσφέρουν αυξημένη απόδοση στα SPARQL ερωτήματα. Η διατήρηση όλων των έξι αναδιατάξεων των RDF στοιχείων, δηλαδή *spo, sop, pso, pos, ops* και *osp*, προσφέρει τα εξής πλεονεκτήματα:

[2] https://hbase.apache.org/
[3] https://github.com/npapa/h2rdf

(1) Όλα τα μοτίβα SPARQL τριπλετών μπορούν να απαντηθούν αποτελεσματικά χρησιμοποιώντας μια σάρωση πάνω σε ένα συγκεκριμένο ευρετήριο. Για παράδειγμα, μια τριπλέτα με δεσμευμένο subject και μεταβλητό predicate και object μπορεί να απαντηθεί χρησιμοποιώντας μια σάρωση του *spo* ή του *sop* ευρετηρίου. (2) Merge συνενώσεις που εκμεταλλεύονται τις προϋπολογισμένες ταξινομήσεις των RDF δεδομένων μπορούν να χρησιμοποιηθούν εκτενώς. Η ύπαρξη των έξι διαφορετικών ταξινομήσεων εγγυάται ότι κάθε ένωση μεταξύ μοτίβων τριπλετών μπορεί να γίνει χρησιμοποιώντας Merge συνενώσεις. Πιο ακριβοί αλγόριθμοι συνένωσης χρειάζονται μόνο όταν εξετάζουμε τη συνένωση ενδιάμεσων μη ταξινομημένων αποτελεσμάτων. Στην περίπτωση του H_2RDF+ , διατηρούμε τις δύο αυτές ιδιότητες, ενώ μεταβαίνουμε σε περιβάλλον κατανεμημένης και κλιμακούμενης εκτέλεσης. Συνοψίζουμε τις κύριες συνεισφορές αυτής της εργασίας ως εξής:

2.3.1 Ευρετήρια HBase

Η HBase είναι μια κατανεμημένη, NoSQL βάση δεδομένων που μπορεί να χειριστεί μεγάλες ποσότητες δεδομένων κλειδιού-τιμής. Οι πίνακες HBase είναι, στην πράξη, ταξινομημένοι χάρτες κλειδιού-τιμής που μοιράζονται στους διάφορους κόμβους βάση ενιαίων ταξινομημένων περιοχών κλειδιού. Στο σύστημά μας, χρησιμοποιούμε έναν πίνακα HBase για κάθε διαφορετικό RDF ευρετήριο. Αφού η HBase χρησιμοποιεί ένα μοντέλο κλειδιού-τιμής, τα ευρετήρια μας αποθηκεύουν τις RDF τριάδες χρησιμοποιώντας το πεδίο κλειδιού αφήνοντας τις τιμές άδειες.

Οι RDF τριάδες περιέχουν μεγάλα URI που συνήθως απαιτούν μεγάλο αποθηκευτικό χώρο, ειδικά στην περίπτωση της χρησιμοποίησης πολλαπλών ευρετηρίων. Για την επίτευξη ενός αποτελεσματικού σχήματος αποθήκευσης, χρησιμοποιούμε αναγνωριστικά (IDs) αντί για URIs, δημιουργώντας επίσης δυο ξεχωριστούς HBase πίνακες που λειτουργούν ως λεξικά για τη μετάφραση URI σε ID και το αντίστροφο. Η ανάθεση αναγνωριστικών σε URI δημιουργείται κατά την εισαγωγή δεδομένων και αναθέτει αναγνωριστικά ανάλογα με τη συχνότητα εμφάνισης των URI στο σύνολο δεδομένων. Ένα συχνά χρησιμοποιούμενο URI θα πάρει ένα ID με τιμή κοντά στο μηδέν. Για να επωφεληθούμε από τα αναγνωριστικά που σχετίζονται με τη συχνότητα εμφάνισης των URI χρησιμοποιούμε μια μεταβλητού μήκους, επιπέδου byte, κωδικοποίηση κατά την αποθήκευση ID στην HBase. Το μεταβλητό μήκος κωδικοποίησης οδηγεί σε μικρότερες παραστάσεις byte για συχνά χρησιμοποιούμενα URI και έτσι επιτυγχάνει υψηλή συμπίεση.

2.3.2 Ευρετήρια Στατιστικών

Εκτός από τους έξι δείκτες που περιγράφονται ανωτέρω, κρατάμε συγκεντρωτικά ευρετήρια στατιστικών που μπορούν να χρησιμοποιηθούν για την εκτίμηση της επιλεκτικότητας ερωτήσεων τριπλέτας, καθώς και για την εκτίμηση του μεγέθους εξόδου των συνενώσεων. Έχουμε δύο κατηγορίες συγκεντρωτικών ευρετηρίων:

1. Με δύο από τα τρία στοιχεία τριπλέτας δεσμευμένα, δηλαδή *sp_o, ps_o, po_s, op_s, os_p* και *so_p*. Για παράδειγμα, το *sp_o* ευρετήριο περιέχει ένα σύνολο (subject, predicate, count) εγγραφών, όπου το count αντιπροσωπεύει το άθροισμα των τριπλετών που περιέχουν τον αντίστοιχο συνδυασμό of subject, predicate.

2. Με ένα από τα τρία στοιχεία τριπλέτας δεσμευμένα, δηλαδή *s_po, p_so, p_os, o_ps, o_sp* και *s_op*. Για παράδειγμα, το *p_so* ευρετήριο περιέχει ένα σύνολο (predicate, count, average) εγγραφών, όπου το count αντιπροσωπεύει το άθροισμα των διαφορετικών subjects που σχετίζονται με αυτό το predicate και το average είναι η μέση τιμή objects που σχετίζονται με κάθε subject.

2.4 Μαζική ευρετηρίαση RDF δεδομένων με χρήση MapReduce

Στην ενότητα αυτή, θα περιγράψουμε διεξοδικά την προτεινόμενη διαδικασία μαζικής ευρετηρίασης MapReduce, που μπορεί να διαχειριστεί σύνολα δεδομένων RDF μεγάλου όγκου. Αποτελείται από τέσσερα εξαιρετικά κλιμακώσιμες εργασίας MapReduce που:

- Μεταφράζουν τα RDF URI σε ακέραια αναγνωριστικά (ID) λαμβάνοντας υπόψιν τη συχνότητα εμφάνισης των URI στο σύνολο των δεδομένων. Ένα συχνά χρησιμοποιούμενο URI θα μεταφραστεί σε ένα αναγνωριστικό με τιμή κοντά στο μηδέν. Τα λεξικά String-ID και ID-String αποθηκεύονται σε ξεχωριστούς πίνακες HBase. Η ανάθεση αναγνωριστικών με βάση τη συχνότητα εμφάνισης σε συνδυασμό με το σύστημα μεταβλητού μήκους κωδικοποίησης που χρησιμοποιούμε για την αποθήκευση των ID μέσα στα ευρετήρια μας επιτυγχάνει μεγάλη μείωση των απαιτήσεων σε χώρο αποθήκευσης.
- Δημιουργούν και να φορτώνουν σε πίνακες HBase τα 6 RDF ευρετήρια και τους πίνακες που κρατούν τα στατιστικά στοιχεία τους.

Για να χειριστούμε μεγάλου όγκου RDF δεδομένα πρέπει να ελαχιστοποιήσουμε τον αριθμό των απαιτούμενων λειτουργιών εισόδου/εξόδου (I/O) και δικτύου. Η μαζική διαδικασία ευρετηρίασης επιτυγχάνει την αποφυγή περιττών επαναλήψεων ανάγνωσης του συνόλου των δεδομένων RDF. Αποφεύγει επίσης την χρήση κλήσεων του HBase API για κάθε εισαγωγή

πλειάδας. Αντ 'αυτού, οι εργασίες MapReduce δημιουργούν τα αντίστοιχα HFiles (μορφή αρχείου αποθήκευσης της HBase), τα οποία στη συνέχεια φορτώνονται χωρίς παραπάνω επεξεργασία σε πίνακες HBase.

2.4.1 Πρώτη εργασία MapReduce

Στην πρώτη εργασία MapReduce κάθε mapper διαβάζει ένα μπλοκ των RDF δεδομένων και δημιουργεί ένα ταξινομημένο χάρτη που περιέχει όλα τα μοναδικά URI που βρέθηκαν στο αρχείο ακολουθούμενα από ένα μετρητή που αντιπροσωπεύει τον αριθμό των φορών που κάθε URI βρέθηκε στο αντίστοιχο μπλοκ RDF . Κατά την cleanup φάση κάθε mapper χρησιμοποιεί αυτή την πληροφορία για την παραγωγή των ακολούθων:

- Για κάθε μπλοκ RDF δημιουργείται ένα αρχείο που περιέχει όλα τα διακριτά URI του. Αυτό το αρχείο θα χρησιμοποιηθεί στα επόμενα βήματα, προκειμένου να ανακτήσουμε αποτελεσματικά τα αναγνωριστικά που απαιτούνται για να μεταφραστούν όλες τις τριάδες στο μπλοκ.
- Κάθε mapper κάνει emmit όλα τα URI-counts του προκειμένου οι reducers να παράξουν ένα συνολικό URI-count για όλο το σύνολο των δεδομένων.
- Επίσης, χρησιμοποιώντας ένα ποσοστό δειγματοληψίας, παίρνουμε ένα δείγμα των RDF τριάδων προκειμένου να δημιουργήσουμε ισορροπημένα χωρίσματα (partitions) τόσο για τον ταξινομημένο χώρο των URI όσο και για τον χώρο ευρετηρίασης (για όλες τις 6 αναδιατάξεις των RDF τριάδων). Όσον αφορά τον ταξινομημένο χώρο των URI, για κάθε δείγμα RDF τριάδας οι mappers κάνουν emmit τρία key-values ένα για κάθε μία από τις ετικέτες (s, p, o). Όλα τα δείγματα των URI key-values αποστέλλονται σε ειδικό reducer που είναι υπεύθυνος για τη δημιουργία μιας ισορροπημένης κατάτμησης του χώρου των URI. Αυτή τη στιγμή, δεν μπορούμε ακόμα να δημιουργήσουμε την κατάτμηση του χώρου ευρετηρίασης, γιατί χρειαζόμαστε τα μεταφρασμένα αναγνωριστικά που δεν έχουν ακόμη αποφασιστεί. Ως εκ τούτου, απλά δημιουργούμε για κάθε RDF μπλοκ ένα αρχείο που περιέχει όλες τις τριάδες του δείγματος.

Η πρώτη εργασία MapReduce περιλαμβάνει δύο τύπους reducer: 1) Ο πρώτος reducer χειρίζεται τα key-value του δείγματος και δημιουργεί την ισορροπημένη κατάτμηση του χώρου των URI. 2) Οι wordcount reducers συνδυάζουν όλα τα τοπικά υπολογισμένα ζεύγη URI-count. Κάθε wordcount reducer διατηρεί μια λίστα που περιέχει τα URI-count ταξινομημένα ανάλογα με το αντίστοιχο count τους. Οι reducers γράφουν τα αρχεία εξόδου τους κατά τη φάση cleanup και έτσι παράγουν URI-count μπλοκ τοπικά διατεταγμένα, σύμφωνα με τη συχνότητα εμφάνισης των URI.

2.4.2 Δεύτερη εργασία MapReduce

Η δεύτερη MapReduce εργασία είναι υπεύθυνη για την ανάθεση μοναδικών και βασισμένων στη συχνότητα εμφάνισης αναγνωριστικών για όλα τα διακριτά URI που υπάρχουν στο σύνολο δεδομένων. Οι mappers της εργασίας αυτής διαβάζουν τα τοπικά διατεταγμένα, σύμφωνα με τη συχνότητα εμφάνισης, URI-count μπλοκ που παρήγαγε το προηγούμενο βήμα. Προκειμένου να αποφευχθεί η ολική διάταξη με βάση το count όλων των ξεχωριστών URI αναθέτουμε αναγνωριστικά χρησιμοποιώντας μια χαλαρή ολική διάταξη που δεν απαιτεί περισσότερη επικοινωνία. Η πρώτη εργασία MapReduce χρησιμοποιεί ένα HashPartitioner για να μοιράσει τα URI στους reducers και ως εκ τούτου, μπορούμε να υποθέσουμε, με μεγάλη πιθανότητα, ότι όλα τα μπλοκ εξόδου θα περιέχουν URIs από όλες τις συχνές και μη συχνές κατηγορίες των ετικετών. Εκμεταλλευόμενοι αυτή την ιδιότητα αναθέτουμε αναγνωριστικά χρησιμοποιώντας τον ακόλουθο τύπο:

$$ID = \begin{cases} locID * R + redID, & \text{if } locID < min \\ offset[redID] + locID - min, & \text{if } locID >= min \end{cases} \tag{2.1}$$

όπου:
ID : είναι το ID που ανατίθεται σε ένα URI
$locID$: είναι το τοπικό ID μέσα σε κάθε μπλοκ που ανατίθεται με βάση την τοπική διάταξη των counts
min : είναι ο ελάχιστος αριθμός key-values που παράχθηκε από έναν wordcount reducer στην πρώτη εργασία MapReduce
$redID$: είναι το ID του reducer που δημιούργησε το αντίστοιχο μπλοκ εξόδου
$offset[]$: όλα τα IDs των reducer θα χρησιμοποιούνται εναλλάξ έως ότου επιτευχθεί ο ελάχιστος αριθμός των κλειδιών. Προκειμένου να αποφευχθεί η εισαγωγή τρυπών στον χώρο των ID μπορούμε να συνεχίσουμε την ανάθεση χρησιμοποιώντας συνεχόμενες περιοχές ID σε κάθε reducer. Αυτό επιτυγχάνεται με τη χρήση του $offset$ πίνακα που περιέχει το πρώτο αναγνωριστικό της αντίστοιχης συνεχόμενης περιοχής. Το $offset$ υπολογίζεται χρησιμοποιώντας τις τιμές $numKeys[redID] - min$ για κάθε reducer.

Τόσο ο ελάχιστος αριθμός key-values, min, όσο και ο $offset$ πίνακας μπορούν εύκολα να υπολογιστούν από την έξοδο της προηγούμενης εργασίας. Μπορούμε να παρατηρήσουμε ότι αυτή η φόρμουλα εναλλάσσει τα αναγνωριστικά μεταξύ των μπλοκ URI-count, δημιουργώντας μια χαλαρή διάταξη στην ανάθεση αναγνωριστικών χωρίς να εισάγει τρύπες στον χώρο των ID. Σημειώνουμε εδώ ότι με τη δημιουργία αυτής της χαλαρής διάταξης αποφεύγουμε περιττές αναγνώσεις και μεταφορές των δεδομένων.

Χρησιμοποιώντας τον ανωτέρω τύπο, οι mappers της δεύτερης εργασίας μπορούν να αναθέσουν, χωρίς επικοινωνία, αναγνωριστικά (ID) στα URI τους. Μετά την ανάθεση ID, για κάθε διαφορετική ετικέτα οι mappers κάνουν emmit δύο key-values προκειμένου να δημιουργηθεί τόσο το String-to-ID όσο και το ID-to-String λεξικό HBase. Αυτή η εργασία λαμβάνει επίσης ως είσοδο τα μπλοκ των διακριτών URI που δημιουργήθηκαν στο πρώτο βήμα. Οι mappers που διαβάζουν αυτά τα αρχεία κάνουν emmit για κάθε ξεχωριστή ετικέτα ένα key-value που περιέχει ως κλειδί το URI και ως τιμή το ID μπλοκ.

Για να φορτώσουμε τα 2 λεξικά String-ID χρησιμοποιούμε δύο ξεχωριστές συναρτήσεις κατάτμησης (partitioners). Η πρώτη συνάρτηση κατάτμησης χρησιμοποιεί την κατάτμηση των URI που υπολογίστηκε στο προηγούμενο βήμα. Η δεύτερη συνάρτηση κατάτμησης χειρίζεται το χώρο ID και απλά σπάει το χώρο σε διαδοχικές περιοχές ίσου μεγέθους. Η εξισορρόπηση φορτίου του ID χώρου καταμερισμού επιτυγχάνεται λόγω του γεγονότος ότι χρησιμοποιείται ένα συνεχόμενος χώρος αναγνωριστικών που δεν περιέχει τρύπες. Χρησιμοποιούμε δύο τύπους reducer για αυτή τη δουλειά. Οι reducers του χώρου των ID απλά δημιουργούν τα αντίστοιχα αρχεία και τα φορτώσει στους πίνακες HBase. Οι reducers του χώρου των URI εκτός από τη δημιουργία των αντίστοιχων HFiles, δημιουργούν και ένα αρχείο που περιέχει για κάθε ζεύγος URI-ID μια λίστα των μπλοκ που το συγκεκριμένο URI χρησιμοποιείται. Το δεύτερο αρχείο χρησιμοποιείται στην ακόλουθη MapReduce εργασία για να μεταφράσει τις RDF τριάδες.

2.4.3 Τρίτη εργασία MapReduce

Η τρίτη εργασία MapReduce αναλαμβάνει την μετάφραση των μπλοκ διακριτών URI. Χρησιμοποιεί τα αρχεία που δημιουργούνται κατά την προηγούμενη εργασία. Έχουμε αναθέσει έναν reducer σε κάθε μπλοκ RDF τριάδων. Η εργασία διαβάζει τα αρχεία που περιέχουν για κάθε ζεύγος URI-ID τη λίστα των μπλοκ που αυτό απαιτείται και για κάθε ένα κάνει emmit ένα key-value που έχει κλειδί το ID του μπλοκ και τιμή το ζεύγος URI-ID. Κάθε reducer παίρνει όλες τις μεταφράσεις του μπλοκ RDF δεδομένων για το οποίο είναι υπεύθυνος και απλά να τις αποθηκεύει σε ένα αρχείο HDFS.

2.4.4 Τέταρτη εργασία MapReduce

Η τελευταία εργασία MapReduce παίρνει ως είσοδο τις RDF τριάδες. Πρώτον, κάθε mapper διαβάζει το αρχείο μετάφρασης για το αντίστοιχο μπλοκ RDF και τις φορτώνει σε ένα λεξικό κεντρικής μνήμης. Στη συνέχεια αναλύει τις τριάδες RDF, μεταφράζει τα URI σε ID και εκπέμπει key-values για όλες τις 6 διαφορετικές διατάξεις. Αυτή η εργασία απαιτεί επίσης τον υπολογισμό μιας ισορροπημένης κατάτμησης για το χώρο της hexastore ευρετηρίασης. Πριν από την έναρξη της εργασίας, μεταφράζουμε τα δείγματα των τριάδων που δημιουργήθηκαν

κατά την πρώτη εργασία χρησιμοποιώντας τα λεξικά HBase και δημιουργούμε ισορροπημένες κατατμήσεις για το χώρο ευρετηρίασης. Κάθε reducer της εργασίας αυτής λαμβάνει ως είσοδο τις τριάδες που αντιστοιχούν σε μια κατάτμηση του χώρου ευρετηρίασης και ενώ δημιουργεί το αντίστοιχο HFile, υπολογίζει και δημιουργεί τα αρχεία των συγκεντρωτικών στατιστικών που περιγράφηκαν παραπάνω. Τα στατιστικά αυτά στοιχεία μπορεί εύκολα να υπολογιστούν όταν οι reducers εξετάζουν τις ταξινομημένες RDF τριάδες και έτσι δεν επιβαρύνουν με πρόσθετο κόστους δικτύου ή I/O.

2.4.5 Indexing storage space

Το H_2RDF+ χρησιμοποιεί ένα επιθετικό σύστημα συμπίεσης για την αποθήκευση των ευρετηρίων, χρησιμοποιώντας: 1) ID που βασίζονται στη συχνότητα εμφάνισης σε συνδυασμό μεταβλητού μήκους κωδικοποίηση για την εγγραφή αναγνωριστικών, 2) τον αλγόριθμο συμπίεσης Google Snappy[4], γνωστός και ως "Zippy", για να μειώσει περαιτέρω το μέγεθος των πινάκων HBase. Το σχήμα μεταβλητού μήκους κωδικοποίησης που χρησιμοποιούμε παρουσιάζεται στον Πίνακα 2.1 και υποστηρίζει IDs με μέγεθος μέχρι 8 byte.

Θετικό Πρόθεμα	**Αρνητικό Πρόθεμα**	**Συνολικά Bytes**	**ID Bits**
10******	01******	1	6+0=6
110*****	001*****	2	5+8=13
1110****	0001****	3	4+16=20
11110***	00001***	4	3+24=27
111110**	000001**	5	2+32=34
11111100	00000011	6	0+40=40
11111101	00000010	7	0+48=48
11111110	00000001	8	0+56=56
11111111	00000000	9	0+64=64

Πίνακας 2.1: *Σχήμα μεταβλητού μήκους κωδικοποίησης*

Το σχήμα κωδικοποίησης χρησιμοποιεί προθέματα μεταβλητού μήκους ώστε να κωδικοποιήσει το μήκος του κάθε αναγνωριστικού. Η συγκεκριμένη επιλογή των προθεμάτων επιτυγχάνει τους ακόλουθους στόχους:

- Διατηρεί την ιδιότητα διάταξης των κωδικοποιημένων αναγνωριστικών σε επίπεδο byte. Αυτό σημαίνει ότι μια ταξινόμηση με βάση τις τιμές byte θα δώσει το ίδιο αποτέλεσμα με μια ταξινόμηση που βασίζεται στην αριθμητική τιμή τους. Αυτή η ιδιότητα είναι πολύ σημαντική, διότι τα ευρετήρια μας εξαρτώνται σε μεγάλο βαθμό από τη byte επιπέδου ταξινόμηση.
- Τα προθέματα μεταβλητού μήκους επιτρέπουν περισσότερα αναγνωριστικά να κωδικοποιούνται με λιγότερα bytes. Ένας στατικός ορισμός προθέματος θα απαιτούσε τουλάχιστον 5 bits

[4] https://code.google.com/p/snappy

για να κωδικοποιήσει όλες τις διαφορετικές περιπτώσεις. Αυτό σημαίνει ότι μόνο 2^3=8 αναγνωριστικά θα μπορούσαν να κωδικοποιούνται χρησιμοποιώντας μόνο ένα byte. Ωστόσο, με βάση την τεχνική μας μπορούμε να κωδικοποιήσουμε 2^6=64 αναγνωριστικά χρησιμοποιώντας μόνο ένα byte. Το ίδιο ισχύει και για όλα τα αναγνωριστικά που μπορούν να κωδικοποιηθούν με λιγότερο από 5 bytes.

2.5 Αλγόριθμοι Εκτέλεσης Συνενώσεων

Η δουλειά μας κάνει μια διπλή συμβολή σε σχέση με την εκτέλεση συνενώσεων. Παρουσιάζουμε mutli-way merge και sort-merge αλγορίθμους συνενώσεων που εκτελούνται με χρήση των κατανεμημένων ευρετηρίων μας. Ο merge αλγόριθμος συνένωσης εκτελεί αποτελεσματική συνένωση πάνω από ήδη ταξινομημένα δεδομένα (δηλαδή, τους πίνακες ευρετηρίων HBase). Ο sort-merge αλγόριθμος εκτελεί συνενώσεις, όταν υπάρχουν ενδιάμεσα μη ταξινομημένα αποτελέσματα. Οι δύο αλγόριθμοι μπορούν να εκτελεστούν τόσο κατανεμημένα (με χρήση του MapReduce) όσο και κεντρικά (χρησιμοποιώντας έναν κόμβο του συμπλέγματος).

2.5.1 Αλγόριθμος Merge συνένωσης με χρήση MapReduce

Αυτός ο αλγόριθμος έχει σχεδιαστεί για τη συνένωση πολλαπλών ερωτημάτων τριάδας που περιέχουν την ίδια μεταβλητή. Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να εκτελέσουμε την ακόλουθη συνένωση ως προς τη μεταβλητή department:

```
select * where{
?person ub:memberOf ?department .
?department ub:subOrganizationOf ?university .
?department rdf:type ub:Department .
}
```

Μπορούμε να πάρουμε τις αντίστοιχες τριάδες ταξινομημένες ως προς το department, αν κάνουμε τις παρακάτω σαρώσεις ευρετηρίων: (για κάθε σάρωση αναφέρουμε το όνομα του ευρετηρίου καθώς και τις δεσμευμένες τιμές στην αντίστοιχη σειρά) {pos, ub:memberOf}, {pso, ub:subOrganizationOf}, {pos, rdf:type, ub:Department}. Για να εκτελέσουμε την κατανεμημένη συνένωση αυτών των σαρώσεων πρέπει πρώτα να βρούμε τη μεγαλύτερη σε μέγεθος δεδομένων σάρωση (δηλαδή, τη σάρωση που εκτείνεται στις περισσότερες περιοχές HBase). Υλοποιούμε τη merge συνένωση ως μια map-only εργασία που παίρνει ως αρχική είσοδο τις περιοχές HBase της μεγαλύτερης σάρωσης. Κάθε mapper επεξεργάζεται μια ταξινομημένη περιοχή της σάρωσης (region), η οποίο μεταφράζεται σε μια ταξινομημένη διαμέριση των κλειδιών της

μεταβλητής της συνένωσης. Ο κάθε mapper χρησιμοποιεί ένα τοπικό scan πάνω από τα δεδομένα της μεγάλη σάρωσης και αρχικοποιεί τα απομακρυσμένα scan των υπόλοιπων σαρώσεων τηρώντας τα όρια της διαμέρισης των κλειδιών της μεταβλητής της συνένωσης.

Για παράδειγμα, ας υποθέσουμε ότι η μεγαλύτερη σάρωση της παραπάνω συνένωσης προέρχεται από το πρώτο ερώτημα τριάδας και περιέχει δύο περιοχές με τα ακόλουθα όρια: [Dep0, Dep5) και [Dep5, Dep10). Σημειώστε ότι χρησιμοποιούμε αλφαριθμητικές τιμές εδώ για αναγνωσιμότητα, τα πραγματικά όρια είναι στο χώρο των αναγνωριστικών ID. Ο πρώτος mapper θα αρχικοποιήσει δύο απομακρυσμένα scan: {pso, ub:subOrganizationOf, [Dep0, Dep5)}, {pos, rdf:type, ub:Department, [Dep0, Dep5)} και θα τα συνενώσει με το τοπικό του scan. Ο δεύτερος mapper θα χειριστεί το εύρος [Dep5, Dep10) αντίστοιχα.

2.5.2 Αλγόριθμος Sort-Merge συνένωσης με χρήση MapReduce

Αυτός ο αλγόριθμος χρησιμοποιείται μόνο όταν συνενώνουμε ενδιάμεσα μη ταξινομημένα αποτελέσματα. Μπορεί να δεχτεί ως είσοδο ένα ή περισσότερα ενδιάμεσα αποτελέσματα και ένα ή περισσότερα ερωτήματα τριάδας. Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να εκτελέσουμε την ακόλουθη συνένωση ως προς τη μεταβλητή department:

```
select * where{
?y ?department ?w . (1)
?z ?department . (2)
?person ub:memberOf ?department . (3)
?department rdf:type ub:Department . (4)
}
```

Οι δυο πρώτες γραμμές παριστάνουν ενδιάμεσα αποτελέσματα που περιέχουν δεδομένα για όλες τις μεταβλητές που απεικονίζονται στο όνομα του προτύπου. Αυτά τα δεδομένα δεν είναι ταξινομημένα με βάση τη μεταβλητή της συνένωσης. Αρχικά, ελέγχουμε τα ερωτήματα τριάδας (πρότυπα (3) και (4)) και βρίσκουμε, με τον τρόπο που περιγράψαμε παραπάνω, τη μέγιστη σάρωση και κατ' επέκταση μια διαμέριση της μεταβλητής της συνένωσης. Η sort-merge συνένωση υλοποιείται ως εργασία MapReduce που παίρνει είσοδο μόνο τα ενδιάμεσα αποτελέσματα (πρότυπα (1) και (2)). Κάθε mapper διαβάζει τα ενδιάμεσα αποτελέσματα που του αντιστοιχούν και κάνει emmit key-values με κλειδί την τιμή της μεταβλητής συνένωσης. Η εργασία χρησιμοποιεί τη μέγιστη διαμέριση της μεταβλητής συνένωσης για να μοιράσει τα δεδομένα στους reducers. Αυτό σημαίνει ότι κάθε reducer θα λάβει ως είσοδο, με ταξινομημένη σειρά, τα κλειδιά της μεταβλητής συνένωσης που του αντιστοιχούν. Ο reducer αρχικοποιεί τις υπόλοιπες σαρώσεις ευρετηρίων για το αντίστοιχο εύρος κλειδιών και στη συνέχεια συνενώνει όλα τα ενδιάμεσα αποτελέσματα και τα αποτελέσματα των ερωτημάτων τριάδας. Σε περίπτωση που η

συνένωση έχει ως είσοδο μόνο ενδιάμεσα αποτελέσματα χρησιμοποιούμαι μια hash διαμέριση για τη συνένωση.

Για παράδειγμα, ας υποθέσουμε ότι η μεγαλύτερη σάρωση της παραπάνω συνένωσης είναι, όπως και πριν: [Dep0, Dep5) και [Dep5, Dep10). Ο πρώτος reducer θα πάρει *όλα* τα ενδιάμεσα δεδομένα που αντιστοιχούν στο πρώτο εύρος και θα αρχικοποιήσει δύο scanners: {pos, ub:memberOf, [Dep0, Dep5)} and {pos, rdf:type, ub:Department, [Dep0, Dep5)}. Ο reducer θα συνενώσει όλα τα παραπάνω για να παράξει το αποτέλεσμα. Το ίδιο θα συμβεί και με τον δεύτερο reducer πάνω από το δεύτερο εύρος τιμών.

2.5.3 Κεντρικοί Αλγόριθμοι συνενώσεων

Υλοποιούμε επίσης, τις κλασικές εκδοχές των merge και sort-merge αλγορίθμων συνένωσης σε ένα περιβάλλον κεντρικής εκτέλεσης. Η μόνη διαφορά είναι ότι χρησιμοποιούμε HBase σαρώσεις και όχι τοπικές σαρώσεις πάνω από B^+-δέντρα ή αρχεία.

2.5.4 Ομαδοποίηση Ενδιάμεσων αποτελεσμάτων

Τα ερωτήματα SPARQL περιλαμβάνουν πολλαπλές συνενώσεις και έτσι το μέγεθος των ενδιάμεσων αποτελεσμάτων μπορεί να αυξηθεί εκθετικά με την εκτέλεση κάθε επιπλέον συνένωσης. Αυτός είναι ο λόγος που χρειαζόμαστε έναν αποδοτικό τρόπο αποθήκευσης των ενδιάμεσων αποτελεσμάτων. Οι κλασικές row-oriented βάσεις δεδομένων δημιουργούν όλες τις πλειάδες των αποτελεσμάτων στο τέλος της κάθε συνένωσης. Αντίθετα, επιλέγουμε να μη δημιουργούμε όλες τις πλειάδες του αποτελέσματος αλλά να διατηρούμε την ομαδοποίηση των δεδομένων για όσο δυνατόν μεγαλύτερο διάστημα. Η lazy αυτή υλοποίηση διατηρεί ομάδες αποτελεσμάτων που περιέχουν: 1) ένα σύνολο από τα ονόματα των μεταβλητών που περιλαμβάνονται στο αποτέλεσμα, 2) για κάθε μεταβλητή, μια λίστα των τιμών της. Τα αποτελέσματα που περιέχονται μέσα σε μια ομάδα πρέπει να ικανοποιούν την ιδιότητα της όλα-με-όλα σύνδεσης, δηλαδή, οι αντίστοιχες πλειάδες μπορεί να παραχθούν από έναν nested βρόχο πάνω από όλες τις μεταβλητές. Ως παράδειγμα, ας υποθέσουμε ότι έχουμε εκτελέσει τα ακόλουθα:

```
select * where{
?department ub:subOrganizationOf ?university .
?student ub:undergraduateDegreeFrom ?university .
}
```

Τα ταξινομημένα ευρετήριά μας μπορούν να ανακτήσουν όλα τα τμήματα και τους φοιτητές ομαδοποιημένους ανά πανεπιστήμιο. Πρέπει να εκμεταλλευτούμε αυτήν την ομαδοποίηση όσο το δυνατόν περισσότερο, προκειμένου να αποφευχθεί η δημιουργία όλων των ενδιάμεσων πλειάδων του αποτελέσματος. Ας υποθέσουμε ότι η βάση δεδομένων μας περιέχει 2 πανεπιστήμια, το καθένα έχει 2 τμήματα και 3 φοιτητές. Τα row-oriented αποτελέσματα

της συνένωσης απεικονίζονται στην αριστερή πλευρά του Σχήματος 2.2. Αντί της υλοποίησης όλων αυτών των συνδυασμών αποθηκεύουμε ομαδοποιημένα αποτελέσματα, όπως απεικονίζεται στη δεξιά μεριά του Σχήματος 2.2. Σημειώστε ότι δεν υπάρχει σαφής σύνδεση μεταξύ των φοιτητών και των τμημάτων (οι φοιτητές και τα τμήματα συνδέονται μόνο με το πανεπιστήμιο και όχι μεταξύ τους), έτσι ισχύει η ιδιότητα όλα-με-όλα σύνδεσης. Επεκτείνοντας το παράδειγμά μας με μεγαλύτερους αριθμούς πλειάδων, εάν η βάση δεδομένων μας περιείχε 100 πανεπιστήμια, το καθένα με 30 τμήματα και 100 χιλιάδες φοιτητές, ένα row-oriented σύστημα θα δημιουργούσε $100 \times 30 \times 100K = 300M$ πλειάδες αναπαράγοντας πολλές φορές τα αναγνωριστικά των πανεπιστημίων και των τμημάτων. Για να αποθηκεύσουμε τα αποτελέσματα, θα χρειαζόμασταν την εγγραφή τριών αναγνωριστικών για κάθε πλειάδα (900 εκατομμύρια IDs). Στο ίδιο παράδειγμα, το σύστημά μας θα δημιουργήσει 100 ομάδες, μία για κάθε πανεπιστήμιο, κάθε ομάδα περιέχει 30 τιμές για τη μεταβλητή τμήμα και 100 χιλιάδες τιμές για τη μεταβλητή φοιτητής. Έτσι, το μέγεθος του ενδιάμεσου αποτελέσματος είναι $100 + 100 \times 30 + 100 \times 100K = 10,003,100$ IDs το οποίο είναι τάξεις μεγέθους μικρότερο από το προηγούμενο. Εφαρμόζουμε επίσης μεταβλητό μήκος κωδικοποίησης για την αποθήκευση των ID πετυχαίνοντας έτσι ένα εξαιρετικά συμπιεσμένο μέγεθος εξόδου.

?university	?department	?student
Univ0	Dep0	St1
Univ0	Dep0	St2
Univ0	Dep0	St3
Univ0	Dep1	St1
Univ0	Dep1	St2
Univ0	Dep1	St3
Univ1	Dep2	St4
Univ1	Dep2	St5
Univ1	Dep2	St6
Univ1	Dep3	St4
Univ1	Dep3	St5
Univ1	Dep3	St6

Row Oriented Results

?university	Univ0
?department	Dep0, Dep1
?student	St1, St2, St3

?university	Univ1
?department	Dep2, Dep3
?student	St4, St5, St6

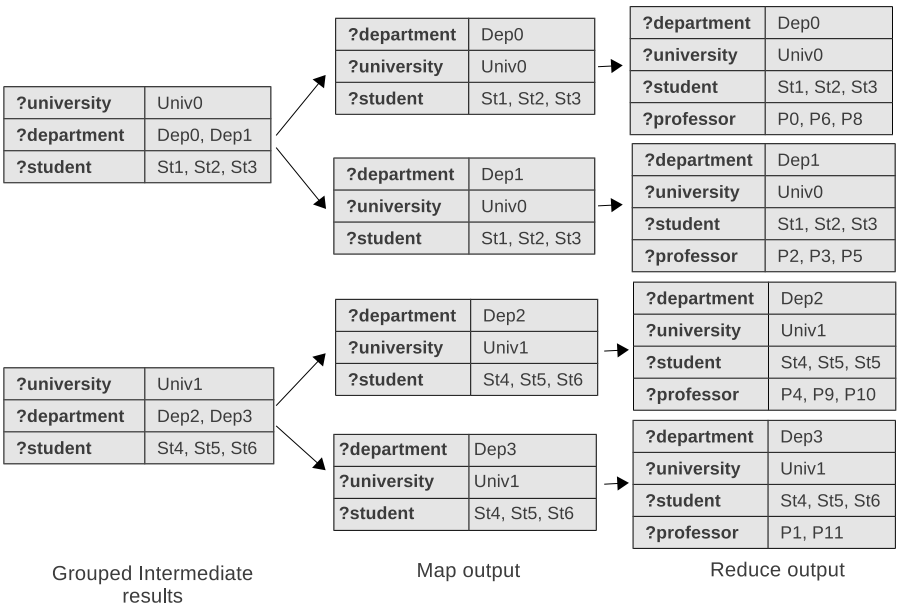
Grouped Results

Σχήμα 2.2: *Ομαδοποιημένα ενδιάμεσα αποτελέσματα*

Όπως προαναφέρθηκε, οι ομάδες χωρίζονται σύμφωνα με την αλληλουχία των συνενώσεων. Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να χρησιμοποιήσουμε τα παραπάνω αποτελέσματα στην ακόλουθη συνένωση:

```
select * where{
?department ?university ?student .
?professor ub:worksFor ?department .
}
```

Αυτή η συνένωση με βάση τη μεταβλητή department υλοποιείται με χρήση του sort-merge αλγορίθμου συνένωσης, που περιγράφηκε στην προηγούμενη ενότητα. Στο Σχήμα 2.3 μπορούμε να δούμε πώς χρησιμοποιούμε τα ομαδοποιημένα αποτελέσματα στη διαδικασία συνένωσης. Αρχικά, στη map φάση, χωρίζουμε την ομάδα σύμφωνα με τη μεταβλητή συνένωσης. Ως εκ τούτου δημιουργούμε μια ομάδα για κάθε department. Σημειώστε ότι η έξοδος του mapper δεν χωρίζεται ως προς τη μεταβλητή student, επειδή αυτές οι τιμές διατηρούν την όλα-με-όλα σχέση με τις υπόλοιπες μεταβλητές. Στη reduce φάση οι ομάδες των καθηγητών ανά τμήμα ανακτώνται από το ευρετήριο και συγχωνεύονται με τις ομάδες εισόδου για να σχηματίσουν την έξοδο.



Σχήμα 2.3: *Συνένωση με χρήση ομαδοποιημένων ενδιάμεσων αποτελεσμάτων*

2.6 Εκτέλεση και Βελτιστοποίηση Ερωτημάτων

Η απόφαση σχετικά με το πλάνο εκτέλεσης του ερωτήματος επηρεάζει σημαντικά τις επιδόσεις των RDF βάσεων δεδομένων, αφού τα ερωτήματα SPARQL συνήθως απαιτούν πολλαπλές συνενώσεις σε διαφορετικές μεταβλητές. Ο βελτιστοποιητής πλάνου εκτέλεσης (join

planner) του H_2RDF+ αποφασίζει σχετικά με τη σειρά εκτέλεσης των διαφορετικών συνενώσεων ώστε να ελαχιστοποιήσει το συνολικό χρόνο εκτέλεσης του ερωτήματος. Για να βρεθεί το βέλτιστο πλάνο πρέπει να λάβουμε υπόψη όλους τους διαφορετικούς συνδυασμούς εκτέλεσης των συνενώσεων. Προφανώς, ο αριθμός των επιλογών αυξάνεται εκθετικά με τον αριθμό των μεταβλητών συνένωσης, καθιστώντας το πρόβλημα υπολογιστικά δαπανηρό. Αντ 'αυτού, χρησιμοποιούμε έναν άπληστο αλγόριθμο, ο οποίος με βάση το κόστος των συνενώσεων αποφασίζει ποια συνένωση θα εκτελεστεί σε κάθε βήμα του ερωτήματος. Για να εκτιμηθεί το κόστος των πιθανών συνενώσεων παρουσιάζουμε ένα λεπτομερές μοντέλου κόστους που εκμεταλλεύεται τα αποθηκευμένα στατιστικά στοιχεία. Τα μοντέλα κόστους μας μπορούν να χρησιμοποιηθούν επίσης για να μπορέσει ο planner να αποφασίσει για το αν η συνένωση θα εκτελεστεί με κατανεμημένο ή κεντρικό τρόπο. Το κίνητρο πίσω από αυτή την απόφαση είναι ότι οι κατανεμημένες εργασίες MapReduce δεν μπορούν να προσφέρουν μικρούς χρόνους απόκρισης για συνενώσεις με λίγα δεδομένα και είναι ωφέλιμες μόνο σε περίπτωση μεγάλων συνενώσεων. Στην ενότητα αυτή παρουσιάζουμε το μοντέλο κόστους των συνενώσεων, καθώς και τον άπληστο αλγόριθμο εύρεσης του πλάνου εκτέλεσης.

2.6.1 Απόδοση σάρωσης των HBase ευρετηρίων

Η απόδοση των αλγορίθμων συνένωσης του H_2RDF+ εξαρτάται σε μεγάλο βαθμό από την απόδοση σάρωσης των HBase ευρετηρίων μας. Για να δημιουργήσουμε ένα μοντέλο που μπορεί να εκτιμήσει το κόστος εκτέλεσης των συνενώσεων πρέπει πρώτα να πειραματιστούμε με τις επιδόσεις των HBase ευρετηρίων μας. Οι βασικές παράμετροι μιας σάρωσης είναι ο χρόνος που απαιτεί για την εύρεση μιας συγκεκριμένης τιμής (seek latency) καθώς και ο ρυθμός ανάγνωσης εγγραφών (read throughput). Μετά από εκτενή πειράματα σαρώσεων ανακαλύψαμε ότι μια πράξη αναζήτησης διαρκεί κατά μέσο όρο 16ms και το μέσο throughput ανάγνωσης φτάνει τα 400.000 key/values(τριάδες)/δευτερόλεπτο. Μια λεπτομερής αξιολόγηση των χαρακτηριστικών αυτών μπορεί να βρεθεί στην ενότητα 2.7.4. Οι τιμές αυτές αφορούν τη συγκεκριμένη υποδομή που χρησιμοποιήσαμε στα πειράματά μας και μπορεί να αλλάζουν σε διαφορετικές συστοιχίες. Παρόλα αυτά, μπορούν να εκτιμηθούν χρησιμοποιώντας απλά πειράματα που εκτελούνται μία φορά για κάθε διαφορετική εγκατάσταση.

Ενσωματώσαμε αυτή τη γνώση στις merge συνενώσεις μας για να τις καταστήσουμε πιο αποδοτικές. Εκτός από διαδοχική σάρωση (sequential scan) των σχέσεων εισόδου της συνένωσης, ένας merge αλγόριθμος μπορεί να χρειαστεί να κάνει άλματα προς τα εμπρός σε μια σχέση, αν γνωρίζουμε ότι δεν υπάρχουν πιθανά αποτελέσματα σε κάποιο συγκεκριμένο εύρος. Σε αυτή την περίπτωση, πρέπει να λάβουμε την απόφαση για το αν θα πρέπει να εκτελέσουμε μια αναζήτηση ή να διαβάσουμε όλες τις ενδιάμεσες τιμές διαδοχικά. Από τις παραπάνω μετρήσεις μπορούμε εύκολα να παρατηρήσουμε ότι ο χρόνος που απαιτείται για τη λειτουργία

αναζήτησης είναι ίσος με το χρόνο που απαιτείται για τη διαδοχική ανάγνωση σχεδόν 6.400 εγγραφών. Έτσι, ο αλγόριθμος συνένωσης χρησιμοποιεί την λειτουργία αναζήτησης μόνο αν αυτή αναμένεται να απορρίψει περισσότερες από 6.400 εγγραφές.

2.6.2 Μοντέλο κόστους εκτέλεσης Merge συνενώσεων

Η merge συνένωση εξαρτάται από τα ερωτήματα τριάδας (Q). Το συνολικό κόστος της συνένωσης (όσον αφορά το χρόνο ολοκλήρωσης) είναι:

$$MJcost(Q) = \sum_{i \in Q} ReadKeys(Q, i)/thr \tag{2.2}$$

$$ReadKeys(Q, i) = \min\{(\min_{j \in Q} n_j) \cdot o_i \cdot SeekOverhead, n_i o_i\} \tag{2.3}$$

n_i: αριθμός τιμών για την join μεταβλητή του i ερωτήματος τριάδας.
o_i: μέσος αριθμός τιμών για τις non-joining μεταβλητές που αντιστοιχούν σε μια τιμή της join μεταβλητής. Αναφέρεται στο i ερώτημα τριάδας.
thr: scan throughput.
SeekOverhead: seek overhead (6,400 εγγραφές)
ReadKeys(Q,i): ο αριθμός των εγγραφών που θα διαβαστούν για το i ερώτημα τριάδας.

Το κόστος της συνένωσης εξαρτάται από τον αριθμό των εγγραφών που πρέπει να διαβαστούν. Για την εκτίμηση αυτού του μεγέθους βρίσκουμε πρώτα τον ελάχιστο αριθμό εγγραφών στα ερωτήματα τριάδας. Ένας merge αλγόριθμος συνένωσης θα πρέπει να διαβάσει το πολύ αυτό το ποσό κλειδιών από κάθε σχέση, χρησιμοποιώντας λειτουργίες αναζήτησης για να απορρίψει κλειδιά που δεν δημιουργούν αποτελέσματα. Όπως αναφέρθηκε πριν χρησιμοποιούμε μια ευριστική μέθοδο για να αποφασίσουμε εάν θα εκτελέσουμε μια αναζήτηση και κατά συνέπεια, στη χειρότερη περίπτωση, η συνένωση θα επιδιώξει άλματα για κάθε κλειδί πληρώνοντας κάθε φορά το *SeekOverhead*.

2.6.3 Μοντέλο κόστους εκτέλεσης Sort-Merge συνενώσεων

Σε αυτόν τον αλγόριθμο συνενώνεται ένα σύνολο ερωτημάτων τριάδας (Q) και ένα σύνολο ενδιάμεσων αποτελεσμάτων (I). Το συνολικό κόστος της συνένωσης (από άποψη χρόνου) είναι:

$$SMJcost(Q, I) = (2\sum_{i \in I} n_i o_i + \sum_{i \in Q} ReadKeys(Q \cup I, i))/thr \tag{2.4}$$

Το κόστος της sort-merge συνένωσης χωρίζεται σε δύο κύρια μέρη. Το πρώτο μέρος είναι το κόστος της συνένωσης των ενδιάμεσων αποτελεσμάτων. Τα ενδιάμεσα αποτελέσματα

διαβάζονται δύο φορές, μία φορά στη map και μία φορά στη reduce φάση. Για τα ερωτήματα τριάδας χρησιμοποιούμε την εκτίμηση που περιγράφηκε στην παραπάνω ενότητα.

2.6.4 Βελτιστοποίηση Πλάνου εκτέλεσης

Το μοντέλο του κόστους που περιγράφηκε στην προηγούμενη ενότητα είναι ένα βήμα προς την εξεύρεση του βέλτιστου πλάνου εκτέλεσης, δηλαδή της σειράς εκτέλεσης με το ελάχιστο συνολικό κόστος. Ο βελτιστοποιητής μας χρησιμοποιεί έναν άπληστο αλγόριθμο που σε κάθε βήμα της εκτέλεσης επιλέγει τη συνένωση με το μικρότερο κόστος εκτέλεσης.

Algorithm 1: H_2RDF+ PLANNER

1: $V \leftarrow \{v_1, v_1, \ldots, v_n\}$ //join μεταβλητές
2: $TQ \leftarrow \{tq_1, tq_1, \ldots, tq_m\}$ //ερωτήματα τριάδας
3: //$TQ(v)$ ερωτήματα τριάδας που περιέχουν την μεταβλητή v
4: **while** $V \neq empty$ **do**
5: *Jstruct* $\leftarrow$ $empty$ //πληροφορίες για τη συνένωση
6: $v_{join} \leftarrow min_{v_i \in V}\{Greedy(v_i, TQ(v_i))\}$
7: *Jstruct*.$addJoin(v_{join}, TQ(v_{join}))$
8: $V.remove(v_{join})$
9: $TQ.remove(TQ(v_{join}))$
10: **if** *Jstruct*.$executionType() =$ *MR* **then**
11: $executeMapReduce$(*Jstruct*)
12: **else if** $Jstruct.executionType() =$ *Cent* **then**
13: $executeCentralized$(*Jstruct*)
14: **end if**
15: **end while**

Ο άπληστος αλγόριθμός μας παρουσιάζεται στον Αλγόριθμο 1. Το σύνολο V περιέχει όλες τις μεταβλητές που θα πρέπει να συνενωθούν, προκειμένου να δοθεί απάντηση στο ερώτημα. Το σύνολο TQ περιέχει όλα τα ερωτήματα τριάδας. Όσο το V περιέχει περισσότερες μεταβλητές πρέπει να εκτελέσουμε επιπλέον συνενώσεις. Χρησιμοποιώντας την άπληστη τεχνική μας επιλέγουμε τη μεταβλητή με το μικρότερο κόστος συνένωσης. Η επιλεγμένη μεταβλητή συνενώνετε πλήρως στην τρέχουσα εργασία (multi-way συνένωση), πράγμα που σημαίνει ότι όλα τα ερωτήματα της αφαιρούνται από TQ.

Η άπληστη τεχνική μας παρουσιάζεται στον Αλγόριθμο 2. Αυτή η συνάρτηση ελέγχει αν η συνένωση απαιτεί μια merge ή sort-merge συνένωση και στη συνέχεια υπολογίζει το κόστος της εκτέλεσης με κεντρικό ή κατανεμημένο τρόπο. Το κεντρικό κόστος είναι το κόστος που περιγράφηκε στην προηγούμενη ενότητα. Το κατανεμημένο κόστος MapReduce υπολογίζεται διαιρώντας το κεντρικό κόστος με το ελάχιστο μεταξύ του αριθμού των κατατμήσεων και του αριθμό των mappers της συστοιχίας κόμβων. Ο αριθμός αυτός είναι το ποσό παραλληλισμού που θα χρησιμοποιηθεί κατά την κατανεμημένη εκτέλεση της συνένωσης. Προσθέτουμε επίσης το *MRoverhead* το οποίο αντιπροσωπεύει το χρόνο αρχικοποίησης μιας εργασίας MapReduce.

Μια εργασία MapReduce χωρίς δεδομένα εισόδου χρειάζεται τουλάχιστον 30 δευτερόλεπτα για να ολοκληρωθεί. Έτσι κίνητρο μας είναι να χρησιμοποιήσουμε κεντρική επεξεργασία, όταν μπορεί να επιτευχθεί γρήγορους χρόνους απόκρισης καθώς και να αξιοποιήσουμε τον παραλληλισμό της κατανεμημένης εκτέλεσης μόνο όταν αντιμετωπίζουμε μεγάλες συνενώσεις.

Algorithm 2: $Greedy(v, TQ)$

```
1: //TQ περιέχει τα ερωτήματα τριάδας που θα συνενωθούν
2: //Χωρίζουμε το TQ σε ερωτήματα τριάδας και ενδιάμεσα αποτελέσματα
3: (Q, I) ← splitPatterns(TQ)
4: if I ≠ empty then
5:    //Sort-merge join
6:    cost ← SMJcost(Q, I)
7: else
8:    //Merge join
9:    cost ← MJcost(Q)
10: end if
11: //Υπολογισμός MapReduce Κόσοτυς
12: MRcost ← cost/ min(patitions, mappers) + MRoverhead
13: if cost < MRcost then
14:    Jstruct.addExecutionType(Cent)
15:    return cost
16: else
17:    Jstruct.addExecutionType(MR)
18:    return MRcost
19: end if
```

2.6.5 Ελαστική εκτέλεση

Στην ενότητα αυτή, θα επικεντρωθούμε στις ιδιότητες προσαρμοστικότητας των πόρων που προσφέρονται από το σύστημά μας. Το H_2RDF+ είναι σε θέση να αποφασίσει σχετικά με τον αριθμό των πόρων που απαιτούνται για την επεξεργασία συνενώσεων. Στην πραγματικότητα, είναι σε θέση να εκτιμήσει αυτόματα το ποσό των απαιτούμενων πόρων, threads στην κεντρική περίπτωση ή mappers/reducers στην κατανεμημένη περίπτωση, ανά συνένωση. Οι προσαρμοστικές αποφάσεις γίνονται κατά τη διάρκεια της εκτέλεσης και κλιμακώνουν σύμφωνα με την εκτίμηση του κόστους της συνένωσης.

Για μικρές συνενώσεις χρησιμοποιούμε κεντρική εκτέλεση ελέγχοντας τον αριθμό των νημάτων που χρησιμοποιούνται. Τόσο η merge όσο και η sort-merge συνένωση μπορούν να εκτελεστούν με παράλληλο τρόπο κατακερματίζοντας το χώρο κλειδιών της μεταβλητής συνένωσης. Αξιοποιώντας τα στατιστικά στοιχεία που βρίσκονται στα συγκεντρωτικά ευρετήρια μας μπορούμε να εκτιμήσουμε τον αριθμό των κλειδιών της join μεταβλητής που περιέχονται σε καθεμία από τις σχέσης της συνένωσης. Χρησιμοποιούμε την εκτίμηση του μέγιστου αριθμού των κλειδιών της join μεταβλητής για να αποφασίσουμε, κατά το χρόνο εκτέλεσης, τον αριθμό

των νημάτων που θα χρησιμοποιηθούν. Στη συνέχεια διαμερίζουμε ανάλογα το χώρο των κλειδιών και αναθέτουμε τις εργασίες στα διαφορετικά νήματα εκτέλεσης. Ξεκινάμε ένα νήμα μόνο αν εκτιμάται ότι θα επεξεργαστεί περισσότερο από ένα ελάχιστο ποσό κλειδιών. Μπορούμε επίσης να θέσουμε ένα όριο στον αριθμό των ταυτόχρονων νημάτων, ώστε να αποφευχθούν καταστάσεις υπερφόρτωσης.

Μεγαλύτερες συνενώσεις εκτελούνται με χρήση MapReduce εργασιών. Οι πόροι που χρησιμοποιούνται για την εκτέλεση MapReduce επιλέγονται, επίσης, με βάση το κόστος της συνένωσης. Οι διαθέσιμοι πόροι μιας συστοιχίας MapReduce είναι ο αριθμός των ταυτόχρονων mapper και reducer. Υποθέτοντας ότι αυτοί έχουν οριστεί για μια συγκεκριμένη συστοιχία, θέλουμε να καταλαμβάνουμε μόνο τον αριθμό των mapper και reducer που απαιτούνται για την εκτέλεση της κάθε συνένωσης. Το κόστος μιας MapReduce συνένωσης είναι ανάλογο με το μέγεθος εισόδου του. Όπως συζητήθηκε, τα δεδομένα εισόδου χωρίζονται σε περιοχές HBase, κάθε περιοχή έχει ένα συγκεκριμένο μέγεθος. Στην υλοποίησή μας, κάθε εργασία mapper χειρίζεται μία περιοχή HBase και έτσι το μέγεθος της περιοχής ρυθμίζεται ώστε να περιέχει την ποσότητα των δεδομένων που απαιτείται για να αποσβεστεί το κόστος δημιουργίας του map task. Αν η περιοχή είχε λιγότερα δεδομένα, η αρχικοποίηση του mapper θα είχε μεγαλύτερη διάρκεια από τον πραγματικό χρόνο επεξεργασίας των δεδομένων. Ως εκ τούτου, ο αριθμός των πόρων που καταλαμβάνεται είναι ανάλογος με το μέγεθος της εισόδου και, κατ ’επέκταση, το κόστος της συνένωσης. Εάν οι εργασίες map που χρειάζεται η συνένωση είναι λιγότερες από τους ταυτόχρονους mappers του συμπλέγματος, οι εναπομείναντες πόροι μπορούν να διατίθενται σε άλλες συνενώσεις. Σε αντίθετη περίπτωση, όλοι οι mappers του συμπλέγματος καταλαμβάνονται.

2.7 Πειράματα

Στην ενότητα αυτή παρουσιάζουμε μια λεπτομερή αξιολόγηση των επιδόσεων του H_2RDF+.

2.7.1 Συστοιχία πειραμάτων

Η πειραματική μας διάταξη αποτελείται από μια ιδιωτική συστοιχία OpenStack η οποία διαθέτει 6 VM containers. Κάθε container διαθέτει έναν 2×6-core Intel Xeon® επεξεργαστή στα 2.67GHz, με 48 GB μνήμης RAM και δύο δίσκους των 2TB ρυθμισμένους σε RAID 0. Οι εικονικές μηχανές (VMs) που χρησιμοποιούνται διαθέτουν 2 εικονικούς πυρήνες, 4GB RAM και 300GB δίσκου, επιτρέποντας στο σύμπλεγμα να υποστηρίξει συνολικά 36 VMs. Οι συστοιχίες που χρησιμοποιούμε για την αξιολόγησή μας αποτελούνται από μεταβλητό αριθμό VMs (10-35). Ακόμα χρησιμοποιούμε ένα κεντρικό VM στο ρόλο του master των HDFS, MapReduce

και HBase. Κάθε VM της συστοιχίας τρέχει 2 mappers και 2 reducers, ο καθένας από τους οποίους καταλαμβάνει 512MB μνήμης RAM. Χρησιμοποιήσαμε την v1.1.2 έκδοση του Hadoop και την v0.94.5 της HBase αντίστοιχα.

2.7.2 Συγκρινόμενα συστήματα

Συγκρίνουμε την απόδοση του H_2RDF+ με τρεις εναλλακτικές βάσεις δεδομένων RDF: το RDF-3X [Neumann 10a], το HadoopRDF [Husain 11], καθώς και την πρώτη έκδοση του κατανεμημένου συστήματος μας H_2RDF [Papailiou 12]. Αξιολογούμε την έκδοση (v0.3.7) του RDF-3X [Neumann 10a,Neumann 10b]. Για το HadoopRDF χρησιμοποιούμε την τελευταία του έκδοση 158 από το SVN repository του.

Όλα τα παραπάνω συστήματα χρησιμοποιούν αναγνωριστικά (ID) και όχι URIs για την αποθήκευση των δεδομένων. Έχουμε παρατηρήσει ότι το στάδιο της μετάφρασης των αναγνωριστικών του αποτελέσματος σε αλφαριθμητικά είναι αρκετά χρονοβόρο για όλες τις συγκρινόμενες βάσεις δεδομένων. Σε ορισμένες περιπτώσεις, το στάδιο αυτό απαιτεί χρόνο συγκρίσιμο ή ακόμα και μεγαλύτερο από τον πραγματικό χρόνο εκτέλεσης του ερωτήματος. Σε αυτή την πειραματική σύγκριση, θα επικεντρωθούμε στον χρόνο εκτέλεσης των συνενώσεων των ερωτημάτων. Έτσι, προκειμένου να παρουσιάσουμε μια δίκαιη σύγκριση, έχουμε αφαιρέσει το στάδιο της μετάφρασης από όλα τα συγκριτικά συστήματα.

2.7.3 Σύνολα δεδομένων

Για να συγκρίνουμε τα συστήματα με μεγάλης κλίμακας, ρεαλιστικά δεδομένα, χρησιμοποιούμε δύο σύνολα RDF δεδομένων. Το σύνολο δεδομένων Yago2 [Hoffart 11] περιέχει πραγματικά δεδομένα που συγκεντρώθηκαν από διάφορες πηγές, όπως η Wikipedia, το WordNet, το GeoNames, κ.α., και περιέχει περισσότερα από 120 εκατομμύρια RDF τριάδες. Αυτό το σύνολο δεδομένων είναι σχετικά μικρό και το χρησιμοποιούμε για να δείξουμε ότι η κατανεμημένη εκτέλεση του ερωτήματος μπορεί να παρουσιάσει καλύτερες επιδόσεις ακόμα και για μικρά σύνολα δεδομένων όταν απαιτούνται μεγάλα μη-επιλεκτικά ερωτήματα. Η LUBM γεννήτρια συνόλων RDF δεδομένων [Guo 05] δημιουργεί σύνολα δεδομένων με πληροφορίες ακαδημαϊκού τομέα, επιτρέποντας ένα μεταβλητό αριθμό RDF τριάδων ελέγχοντας τον αριθμό των *πανεπιστημίων* που θα περιέχονται στο σύνολο. Μεταβάλλοντας την παράμετρο αυτή μεταξύ χιλίων και 100 χιλιάδων πανεπιστημίων, δημιουργούμε σύνολα δεδομένων που κυμαίνονται από 1,4 εκατομμύρια (25GB) μέχρι 13,8 δισεκατομμύρια τριπλέτες (2.5TB). Τα LUBM σύνολα δεδομένων χρησιμοποιούνται ευρέως για να συγκρίνουν την απόδοση των RDF βάσεων δεδομένων, ιδίως όταν απαιτούνται αυθαίρετα μεγάλα σύνολα δεδομένων. Το Lehigh

πανεπιστήμιο έχει επίσης δημοσιεύσει μια σειρά από τεστ ερωτήματα[5] που προσφέρουν ένα καλό μείγμα διαφορετικών ειδών ερωτημάτων SPARQL.

2.7.4 Συγκριση ευρετηρίων

Στην ενότητα αυτή θα αξιολογηθεί η απόδοση του συστήματος ευρετηρίασης μας. Αρχικά, εξετάζουμε τις απαιτήσεις σε χώρο αποθήκευσης. Όπως αναφέρθηκε στο κεφάλαιο 2.3.1, το H_2RDF+ χρησιμοποιεί ένα σύστημα επιθετικής συμπίεσης χρησιμοποιώντας μεταβλητό μήκος κωδικοποίησης και μικρότερα αναγνωριστικά για συχνά εμφανιζόμενες τιμές συμβολοσειράς. Συμπιέζει επίσης, τους HBase πίνακες των ευρετηρίων χρησιμοποιώντας τη Google Snappy συμπίεση[6], γνωστή και ως "Zippy". Επιλέγουμε τη βιβλιοθήκη Snappy γιατί προσφέρει πολύ υψηλή ταχύτητα αποσυμπίεσης και αρκετά καλά ποσοστά συμπίεσης. Η αποδοτική από πλευράς χρησιμοποίησης της CPU αποσυμπίεση του Snappy, τον καθιστά μια καλή επιλογή για τις NoSQL βάσεις δεδομένων αξιοποιώντας το trade-off μεταξύ χώρου αποθήκευσης, I/O και CPU ενεργειών.

Σύνολο Δεδομένων	**Αρχικό Μέγεθος**	**RDF-3X**	**H_2RDF**	**H_2RDF+ (χωρίς Snappy)**	**H_2RDF+**
LUBM1k	28 GB	9 GB	25 GB	27 GB	7 GB
LUBM10k	276 GB	77 GB	214 GB	241 GB	62 GB
LUBM20k	549 GB	156 GB	529 GB	545 GB	121 GB
Yago2	26 GB	12 GB	33 GB	35 GB	10 GB

Πίνακας 2.2: *Σύγκριση απαιτήσεων σε χώρο αποθήκευσης*

Στον πίνακα 2.2 καταχωρούμε τις απαιτήσεις σε χώρο αποθήκευσης των συγκρινόμενων συστημάτων για τα σύνολα δεδομένων LUBM και Yago. Η στήλη "Αρχικό Μέγεθος" περιέχει το μέγεθος του συνόλου δεδομένων σειριοποιημένο χρησιμοποιώντας το *N-Triples* πρότυπο. Αν και αποθηκεύουμε 6 αντί για 3 ευρετήρια και πιο λεπτομερή στατιστικά στοιχεία, το H_2RDF+ καταφέρνει να έχει μικρότερες απαιτήσεις χώρου από την προηγούμενη έκδοση του λόγω: 1) του μικρότερου μεγέθους αναγνωριστικών, το H_2RDF χρησιμοποιεί ως ID το 8-byte MD5-hash των τιμών συμβολοσειράς, 2) της byte-επιπέδου κωδικοποίησης μεταβλητού μήκους σε συνδυασμό με τη βασισμένη στη συχνότητα εμφάνισης ανάθεση των ID, 3) της Snappy συμπίεσης των μπλοκ των ευρετηρίων. Το RDF-3X προσφέρει επίσης ένα εξαιρετικά συμπιεσμένο σύστημα αποθήκευσης λόγω της gap τεχνικής συμπίεσής του [Neumann 10a] (αποθηκεύει μόνο τη διαφορά μεταξύ διαδοχικών τριάδων που περιλαμβάνονται στα ευρετήρια). Η διαφορά μεταξύ των απαιτήσεων αποθήκευσης του H_2RDF+ και του RDF-3X προέρχεται κυρίως από τη

[5] http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt
[6] https://code.google.com/p/snappy

βασισμένη στη συχνότητα εμφάνισης ανάθεση των ID και την μπλοκ-επίπεδου Snappy συμπίεση που χρησιμοποιούνται στο H_2RDF+ (επιτυγχάνει ∼70% μείωση στον απαιτούμενο χώρο αποθήκευσης).

	RDF-3X	**H_2RDF**	**H_2RDF+**
Τοπικός Ρυθμός ανάγνωσης (million triples/sec)	17	0.73	1.13
Απομακρυσμένος Ρυθμός Ανάγνωσης (million triples/sec)	-	0.2	0.4
Ταχύτητα αναζήτησης, cold cache (ms)	1	86	16
Ταχύτητα αναζήτησης, warm cache (ms)	0.2	17	7

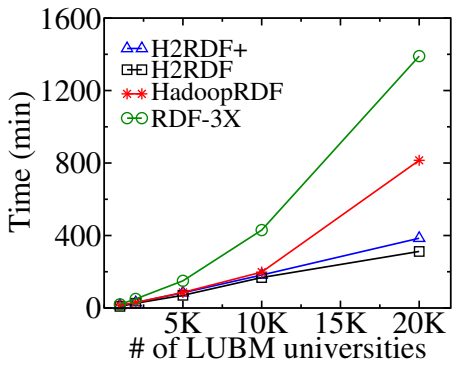
Πίνακας 2.3: *Σύγκριση ρυθμού ανάγνωσης και ταχύτητας αναζήτησης των ευρετηρίων*

Συγκρίνουμε επίσης την απόδοση ανάγνωσης και αναζήτησης των ευρετηρίων των αντίστοιχων τεχνολογιών. Όπως αναφέρθηκε στο κεφάλαιο 2.6.1, ο ρυθμού ανάγνωσης και η ταχύτητα αναζήτησης είναι πολύ σημαντικές μετρήσεις που πρέπει να βελτιστοποιηθούν και να αξιολογούνται. Από τον πίνακα 2.3, συμπεραίνουμε ότι το νέο σύστημα ευρετηρίασης μας επιτυγχάνει σημαντικές βελτιώσεις σε όλες τις κατηγορίες σε σύγκριση με την προηγούμενη εκδοσή του. Παρατηρούμε μια βελτίωση 54% στον τοπικό ρυθμό ανάγνωσης (η ανάγνωση γίνεται από το ίδιο μηχάνημα που διαθέτει τα HBase δεδομένα) και 100% βελτίωση στον απομακρυσμένο ρυθμό ανάγνωσης (η ανάγνωση γίνεται από διαφορετικό μηχάνημα από αυτό που διαθέτει τα HBase δεδομένα). Αυξάνουμε επίσης σημαντικά την ταχύτητα αναζήτησης γεγονός που οφείλεται στην πιο συμπαγή αναπαράσταση των εγγραφών της HBase.

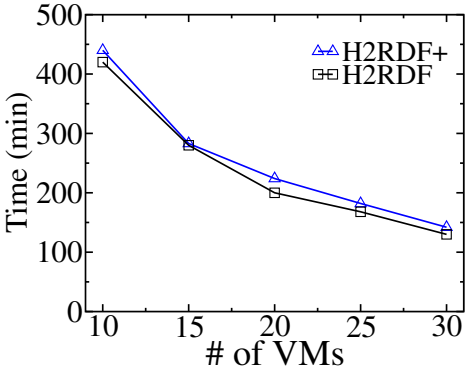
Σε σύγκριση με το RDF-3X, οι χρόνοι ανάγνωσης και αναζήτησης είναι σχεδόν μια τάξη μεγέθους μεγαλύτεροι. Το RDF-3X χρησιμοποιεί εξαιρετικά αποδοτικά B^+δέντρα που τοποθετούνται στον τοπικό σκληρό δίσκο του μηχανήματός. Αντίθετα, τα ευρετήριά μας παρουσιάζουν χαμηλότερη απόδοση λόγω των κατανεμημένων αρχιτεκτονικών της HBase και του HDFS. Αυτή η διαφορά απόδοσης όμως ανακουφίζεται από την ικανότητα των κατανεμημένων, ταυτόχρονων σαρώσεων μέσα από τις MapReduce εργασίες. Οι επιπτώσεις της χρήσης κατανεμημένων ευρετηρίων γίνεται ορατή και στην επόμενη ενότητα, στην περίπτωση των μικρών, επιλεκτικών ερωτημάτων. Η επίδραση αυτή εξαφανίζεται όταν επεξεργαζόμαστε κατανεμημένες, μη-επιλεκτικές συνενώσεις.

2.7.5 Ευρετηρίαση δεδομένων

Στην ενότητα αυτή θα αξιολογηθεί η απόδοση της μαζικής διαδικασίας ευρετηρίασης του H_2RDF+. Η διαδικασία δημιουργίας ευρετηρίων αποτελείται από 4 εργασίες MapReduce που



Σχήμα 2.4: *Χρόνος ευρετηρίασης για μεταβλητό μέγεθος δεδομένων*



Σχήμα 2.5: *Χρόνος ευρετηρίασης για μεταβλητό αριθμό πόρων*

δημιουργούν όλους τους 6 συνδυασμούς RDF ευρετηρίων. Για να δοκιμαστεί η αποτελεσματικότητα της μεθόδου ευρετηρίασής μας το συγκρίνουμε με τα RDF-3X, HadoopRDF καθώς και την πρώτη έκδοση του συστήματος μας, H_2RDF.

Αρχικά συγκρίνουμε τις ιδιότητες κλιμάκωσης,όσον αφορά το μέγεθος του συνόλου δεδομένων RDF, όλων των συστημάτων. Για να γίνει αυτό χρησιμοποιούμε τη γεννήτρια σύνολο δεδομένων LUBM που μπορεί να δημιουργήσει RDF σύνολα δεδομένων με μεταβλητό μέγεθος. Χρησιμοποιούμε σύνολα δεδομένων LUBM που περιέχουν χίλια έως 20 χιλιάδες πανεπιστήμια, δηλαδή, 0,14 - 2,7 δισεκατομμύρια τριάδες (28 έως 549 GB δεδομένων αντίστοιχα). Τα H_2RDF+, H_2RDF και HadoopRDF εκτελέστηκαν χρησιμοποιώντας ένα σύμπλεγμα από 25 VMs, ενώ το RDF-3X χρησιμοποιεί έναν κόμβο που διαθέτει έναν 4×16-core επεξεργαστή, με 128 GB μνήμης RAM και 1TB σκληρό δίσκο. Οι συνολικοί χρόνοι φόρτωσης των δεδομένων παρουσιάζονται στο Σχήμα 2.4. Αυτός είναι ο χρόνος που απαιτείται από όλα τα συστήματα ώστε να φορτώσουν το πλήρες σύνολο δεδομένων σύμφωνα με το σύστημα ευρετηρίασης τους.

Το RDF-3X, είναι ένα κεντρικό σύστημα, αναλύει όλες τις τριάδες διαδοχικά, προκειμένου να δημιουργήσει τα ευρετήρια του. Δεν εκμεταλλεύεται τις δυνατότητες παραλληλισμού που προσφέρει η σύγχρονη αρχιτεκτονική πολλαπλών πυρήνων των επεξεργαστών. Επίσης διαβάζει τα δεδομένα εισόδου αρκετές φορές, προκειμένου να δημιουργήσει τις διάφορες αναδιατάξεις των τριάδων. Αυτή η επαναληπτική σάρωση των δεδομένων εισόδου οδηγεί σε μια εκθετικά αυξανόμενη πολυπλοκότητα ευρετηρίασης. Όπως μπορούμε να δούμε στο Σχήμα 2.4, το RDF-3X παρουσιάζει τους πιο αργούς, μεταξύ των συγκρινόμενων συστημάτων, χρόνους ευρετηρίασης για τα σύνολα RDF δεδομένων.

Για την ευρετηρίαση των δεδομένων, το HadoopRDF χρειάζεται να εκτελέσει 4 εργασίες MapReduce που λαμβάνουν ως είσοδο ολόκληρο το σύνολο RDF δεδομένων. Αυτό σημαίνει ότι πρέπει να σαρώσει τα δεδομένα τέσσερις φορές με αποτέλεσμα χαμηλή απόδοση ευρετηρίασης. Επιπλέον, ορισμένες από αυτές τις εργασίες δεν κατανέμουν εξίσου τα δεδομένα στους

reducers και έτσι υπερφορτώνουν κάποιους reducers αφήνοντας τους άλλους σε αδράνεια. Η εξισορρόπηση του φορτίου μεταξύ των διαθέσιμων υπολογιστικών πόρων είναι μία από τις πιο σημαντικές προκλήσεις προκειμένου τα κατανεμημένα συστήματα να προσφέρουν καλές ιδιότητες κλιμάκωσης. Μπορούμε να παρατηρήσουμε ότι το HadoopRDF, ενώ δημιουργεί μόνο μία αναδιάταξη των RDF τριάδων χρησιμοποιώντας αρχεία HDFS, απαιτεί 2 φορές περισσότερο χρόνο από ό,τι το H_2RDF+ για τη φόρτωση του συνόλου δεδομένων LUBM20k.

Συγκρίνουμε επίσης το H_2RDF+ με την προηγούμενη έκδοσή του, το H_2RDF. Το H_2RDF χρησιμοποιεί 2 εργασίες MapReduce για να δημιουργήσει 3 από τα 6 RDF ευρετήρια. Η πρώτη είναι μια εργασία δειγματοληψίας που δημιουργεί ισορροπημένες διαμερίσεις του χώρου ευρετηρίασης, ενώ η δεύτερη εργασίας δημιουργεί και φορτώνει τα 3 ευρετήρια στην HBase. Στο Σχήμα 2.4 μπορούμε να παρατηρήσουμε ότι το H_2RDF+ καταφέρνει να δημιουργήσει 3 επιπλέον ευρετήρια και να κρατήσει πιο λεπτομερή στατιστικά στοιχεία από το H_2RDF απαιτώντας 10-20% λιγότερο χρόνο ευρετηρίασης. Αυτό οφείλεται κυρίως:

- Στη βελτιστοποιημένη διαδικασία ευρετηρίασης μας, που ελαχιστοποιεί τις φορές ανάγνωσης των αρχικών RDF δεδομένων. Ενώ απαιτεί 4 MapReduce εργασίες για την ευρετηρίαση του συνόλου δεδομένων, μόνο 2 από αυτές διαβάζουν τα αρχικά δεδομένα, ενώ τα αρχεία εξόδου των υπόλοιπων εργασιών είναι αρκετά μικρότερα από το αρχικό σύνολο δεδομένων. Αυτό μειώνει σημαντικά το χρόνο ανάγνωσης δεδομένων, I/O, που απαιτείται για την εκτέλεση της διαδικασίας ευρετηρίασης.
- Στην επιθετική συμπίεση που χρησιμοποιείται σε όλα τα στάδια ευρετηρίασης. Χρησιμοποιούμε μεταβλητού μήκους κωδικοποίηση για να γράψουμε όλα τα ενδιάμεσα και τα τελικά αποτελέσματα της διαδικασίας ευρετηρίασης και έτσι εξοικονομούμε χώρο αποθήκευσης και I/O λειτουργίες.
- Η εκτεταμένη χρήση δειγματοληψίας μας επιτρέπει να δημιουργήσουμε εξισορροπημένες κατατμήσεις για όλα τα βήματα MapReduce υπολογισμών.

Ένα άλλο σημαντικό σημείο είναι η κλιμακωσιμότητα της ευρετηρίασης σε σχέση με τον αριθμό των διαθέσιμων υπολογιστικών πόρων. Δημιουργούμε ευρετήρια για το σύνολο δεδομένων LUBM10k (1,3 δισεκατομμύρια τριάδες) με τη χρήση συστοιχιών με διαφορετικό αριθμό κόμβων. Χρησιμοποιούμε συστοιχίες με 10, 15, 20, 25 και 30 κόμβους. Τα αντίστοιχα αποτελέσματα παρουσιάζονται στο Σχήμα 2.5. Μπορούμε να παρατηρήσουμε ότι το H_2RDF+ καταφέρνει να διατηρεί τις ιδιότητες κλιμάκωσης του H_2RDF ενώ εισάγει μια πιο σύνθετη διαδικασία RDF ευρετηρίασης. Χρησιμοποιώντας το πλαίσιο MapReduce μπορούμε να κερδίσουμε σχεδόν γραμμική επιτάχυνση όταν αυξάνουμε τον αριθμό των κόμβων της συστοιχίας. Χρησιμοποιώντας 10 κόμβους μπορούμε να επιτύχουμε μια ταχύτητα εισαγωγής 49 Ktriples/sec, ενώ χρησιμοποιώντας 30 κόμβους έχουμε σχεδόν τριπλάσια ταχύτητα εισαγωγής 142Ktriples/sec.

Παρατηρούμε επίσης ότι το H_2RDF+ εισάγει μόνο μια μικρή χρονική επιβάρυνση (10-20%) σε σύγκριση με το H_2RDF.

2.7.6 Σύγκριση με άλλα συστήματα

	Yago2		
	H_2RDF+	H_2RDF	HadoopRDF
Εισαγωγή(min)	31	26	72
YQ1(sec)	0.9	0.9	52
YQ2(sec)	1.5	1.7	68
YQ3(sec)	154	952	1832
YQ4(sec)	87	728	1495
	LUBM10k		
	H_2RDF+	H_2RDF	HadoopRDF
Εισαγωγή(min)	182	168	198
LQ1(sec)	0.6	0.6	152
LQ3(sec)	0.8	0.8	231
LQ4(sec)	2.1	2.4	1289
LQ2(sec)	95	635	915
LQ9(sec)	151	787	1488
	LUBM20k		
	H_2RDF+	H_2RDF	HadoopRDF
Εισαγωγή(min)	385	312	815
LQ1(sec)	0.8	0.8	378
LQ3(sec)	0.9	1	449
LQ4(sec)	2.3	2.4	2650
LQ2(sec	131	880	1367
LQ9(sec)	292	1034	2933
	LUBM100k		
	H_2RDF+	H_2RDF	
Εισαγωγή(min)	1154	985	
LQ1(sec)	0.8	0.9	
LQ3(sec)	1.1	1.1	
LQ4(sec)	2.4	2.5	
LQ2(sec)	412	1853	
LQ9(sec)	890	2761	

Πίνακας 2.4: *Συγκριση απόδοση για τα H_2RDF+, H_2RDF και HadoopRDF χρησιμοποιώντας τα LUBM και Yago2 σύνολα δεδομένων*

Προκειμένου να πραγματοποιήσουμε μια δίκαιη σύγκριση μεταξύ των διαφόρων συστημάτων, δοκιμάζουμε πρώτα την απόδοση του H_2RDF+ έναντι των άλλων κατανεμημένων συστημάτων. Χρησιμοποιούμε τέσσερα σύνολα δεδομένων, δηλαδή τα LUBM με 10, 20 και 100 χιλιάδες πανεπιστήμια και το Yago2, που αποτελούνται από 1.3, 2.7, 13.8 και 0.12 δισεκατομμύρια RDF τριάδες αντίστοιχα. Τα H_2RDF+, H_2RDF και HadoopRDF εκτελέστηκαν χρησιμοποιώντας ένα υπολογιστικό σύμπλεγμα 25 κόμβων. Στον πίνακα 2.4 καταχωρούμε τους χρόνους εισαγωγής και ευρετηρίασης δεδομένων καθώς και τους χρόνους απόκρισης για επιλεγμένα ερωτήματα. Για μια δίκαιη σύγκριση με την κεντρική βάση δεδομένων RDF-3X, πραγματοποιούμε δύο πειράματα χρησιμοποιώντας το ίδιο *συνολικό* ποσό υπολογιστικών πόρων. Για το RDF-3X, χρησιμοποιούμε δύο setup πειραμάτων: 1) έναν server που διαθέτει έναν 2×Quad-Core επεξεργαστή, 8 GB RAM, 8 GB swap και δίσκο 1TB και 2) έναν server που διαθέτει έναν 4×16-Core επεξεργαστή, 128 GB μνήμης RAM και 1TB δίσκο. Για την διεξαγωγή των αντίστοιχων πειραμάτων του H_2RDF+ χρησιμοποιούμε αριθμό από VMs που αντιστοιχούν στις συνολικές υπολογιστικές δυνατότητες του αντίστοιχου κεντρικού server. Αυτά τα αποτελέσματα αναφέρονται στον Πίνακα 2.5.

Απόδοσης εκτέλεσης ερωτημάτων: Το LUBM παρέχει ένα σύνολο SPARQL ερωτημάτων που μπορούν να χρησιμοποιηθούν για την αξιολόγηση των συστημάτων. Όλα τα υπό σύγκριση συστήματα δεν υποστηρίζουν OWL λειτουργίες γι 'αυτό σε αυτή την πειραματική αξιολόγηση χρησιμοποιούμε μόνο τα ερωτήματα που δεν απαιτούν συλλογιστική ή, σε ορισμένες περιπτώσεις (π.χ., LQ9 ερώτημα) αφαιρούμε την ιεραρχία του *rdf:type* predicate από την αναζήτηση. Παρουσιάζουμε τα αποτελέσματα για πέντε τέτοια ερωτήματα (που αναφέρονται ως LQ) και παρέχουν ένα καλό μείγμα τόσο απλών όσο και σύνθετων ερωτημάτων. Το επιλεγμένο σύνολο καλύπτει όλες τις παραλλαγές των ερωτημάτων LUBM. Επιπλέον, είναι σε θέση να αναδείξει τις διαφορετικές αποφάσεις και τα χαρακτηριστικά του H_2RDF+ καθώς και των άλλων υπό σύγκριση συστημάτων. Το Yago2 δεν παρέχει ερωτήματα αναφοράς για την αξιολόγηση RDF συστημάτων. Ανάλογα με το LUBM, έχουμε δημιουργήσει μια σειρά από αντιπροσωπευτικά SPARQL ερωτήματα δοκιμής. Πιο αναλυτικά, διακρίνουμε δύο κύριες κατηγορίες των ερωτημάτων SPARQL: εκείνα που περιέχουν κάποιο επιλεκτικό ερώτημα τριάδας και μικρό αριθμό αποτελεσμάτων (LQ1, LQ3, LQ4, YQ1, YQ2) και αυτά που δεν περιέχουν επιλεκτικά ερωτήματα και περιέχουν πολύπλοκες δομές συνένωσης (LQ2, LQ9, YQ3, YQ4).

Το H_2RDF+ αποδίδει σημαντικά καλύτερα σε ερωτήματα με μεγάλη είσοδο. Εκμεταλλευόμαστε τις διαφορετικές ταξινομήσεις που παρέχονται από τα ευρετήριά μας μέσω των Merge και Sort-Merge συνενώσεων και έτσι επιτυγχάνουμε σχεδόν 7× κέρδος απόδοσης σε σύγκριση με το H_2RDF και 10× σε σύγκριση με το HadoopRDF. Επίσης το H_2RDF+ παρουσιάζει καλύτερη απόδοση από το RDF-3X στα περισσότερα από τα σύνθετα ερωτήματα, τόσο κατά τα πειράματα με λίγους όσο και σε αυτά με τους περισσότερους πόρους. Για παράδειγμα, για το LQ2, το RDF-3X απαιτεί σχεδόν 12GB μνήμης για να εκτελέσει το ερώτημα στο LUBM10k και

	Yago2			
Πόροι	8CPU/8GB RAM		64CPU/128GB RAM	
	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X
Εισαγωγή(min)	164	157	26	149
YQ1(sec)	0.9	0.7	0.9	0.7
YQ2(sec)	1.5	1	1.6	0.9
YQ3(sec)	241	3037	138	1929
YQ4(sec)	123	2973	79	2068
	LUBM10k			
Πόροι	8CPU/8GB RAM		64CPU/128GB RAM	
	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X
Εισαγωγή(min)	912	605	162	576
LQ1(sec)	0.6	0.4	0.6	0.3
LQ4(sec)	2.1	0.8	2.1	0.7
LQ2(sec)	373	2297	89	1277
LQ9(sec)	411	68	141	51
	LUBM20k			
Πόροι	8CPU/8GB RAM		64CPU/128GB RAM	
	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X
Εισαγωγή(min)	2075	1526	349	1398
LQ1(sec)	0.8	0.4	0.9	0.4
LQ4(sec)	2.3	0.8	2.2	0.8
LQ2(sec)	706	Failed	119	2065
LQ9(sec)	753	Failed	264	289

Πίνακας 2.5: *Σύγκριση απόδοσης των H_2RDF+ και RDF-3X*

αποδεικνύεται 6 φορές πιο αργό από το H_2RDF+ όταν χρησιμοποιείται ο μικρός server. Στο LUBM20k (και χρησιμοποιώντας τον μεγαλύτερο server), παρουσιάζεται μια 14× επιβράδυνση σε σύγκριση με το H_2RDF+. Το σύστημά μας επιτυγχάνει 3 έως 6 φορές μικρότερους χρόνους απόκρισης κατά τη μετακίνηση σε ένα μεγαλύτερο σύμπλεγμα, ενώ η επιτάχυνση του RDF-3X οφείλεται κυρίως στη μεγαλύτερη ποσότητα της μνήμης. Για το LQ9, το RDF-3X καταφέρνει να έχει καλύτερες επιδόσεις, όταν καταφέρνει να φορτώσει τα δεδομένα του ερωτήματος στη μνήμη. Ωστόσο, η προσέγγιση αυτή δεν κλιμακώνει όταν δεν υπάρχει αρκετή μνήμη, όπως συμβαίνει στην περίπτωση του LUBM20k στο μικρό διακομιστή.

Για τα μικρά, επιλεκτικά ερωτήματα το H_2RDF+ χρησιμοποιεί κεντρική εκτέλεση και καταφέρνει να παρουσιάσει απόδοση συγκρίσιμη με αυτή του RDF-3X. Η διαφορά στην απόδοση οφείλεται κυρίως στο χαμηλότερο ρυθμό ανάγνωσης και στην χαμηλότερη ταχύτητα αναζήτησης των κατανεμημένων HBase ευρετηρίων μας. Σημειώνουμε επίσης ότι υπάρχει μια μικρή βελτίωση σε σχέση με το H_2RDF λόγω της βελτιστοποιημένης ευρετηρίασης που παρουσιάστηκε σε προηγούμενες ενότητες. Σημειώνουμε επίσης ότι το HadoopRDF έχει πραγματικά

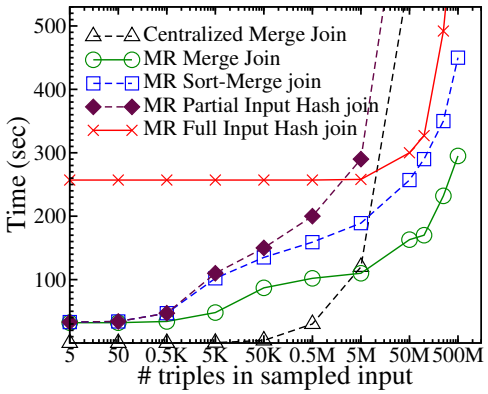
κακή απόδοση για όλα τα επιλεκτικά ερωτήματα, η οποία οφείλεται στο γεγονός ότι εκτελεί μόνο MapReduce συνενώσεις καθώς και στο γεγονός ότι δεν μπορεί να επωφεληθεί πλήρως από την επιλεκτικότητα των ερωτημάτων.

Από τα αποτελέσματα αυτά, συμπεραίνουμε ότι το H_2RDF+ επεξεργάζεται όλα τα είδη ερωτημάτων, σύμφωνα με τους στόχους που τέθηκαν στο σχεδιασμό του. Καταφέρνει να προσδιορίσει σωστά τις επιλεκτικές έναντι των μη-επιλεκτικών συνενώσεων, και να τις εκτελέσει με κεντρικό ή κατανεμημένο τρόπο. Σε ερωτήματα υψηλής επιλεκτικότητας, είναι σχεδόν τόσο αποτελεσματικό όσο το RDF-3X, με μια μικρή διαφορά (μερικά δέκατα του δευτερολέπτου), λόγω του γεγονότος ότι τα ευρετήριά μας είναι κατανεμημένα σε πολλούς κόμβους του συμπλέγματος. Αυτή η μικρή διαφορά απόδοσης μετριάζεται από την ικανότητά του συστήματος για εξυπηρέτηση πολλαπλών ταυτόχρονων ερωτημάτων (βλέπε ενότητα 2.7.8). Για τα περισσότερο απαιτητικά ερωτήματα, αποδεικνύεται καλύτερο τόσο από τα κεντρικά όσο και από τα κατανεμημένα ανταγωνιστικά συστήματα.

LUBM αξιολόγηση πλήρους κλίμακας: Ο Πίνακας 2.4 περιέχει επίσης τους χρόνους ευρετηρίασης και εκτέλεσης ερωτημάτων για το σύνολο δεδομένων LUBM100k, για το H_2RDF+ και για το H_2RDF. Το σύνολο αυτό δεδομένων αποτελείται από 14 δισεκατομμύρια τριάδες (2.5 TB) και χρησιμοποιούμε ένα σύμπλεγμα από 35 κόμβους. Το σύστημά μας επιτυγχάνει μια ταχύτητα εισαγωγής των 202 Ktriples/sec, η οποία είναι μια πολύ καλή επίδοση σύμφωνα με το [W3C 15]. Οι χρόνοι απόκρισης των ερωτημάτων ακολουθούν την τάση που περιγράφηκε στα προηγούμενα πειράματα. Για επιλεκτικά ερωτήματα, χρησιμοποιείται κεντρική εκτέλεση, οδηγώντας σε χρόνους εκτέλεσης που κυμαίνονται μεταξύ 0,8 και 2,4 sec. Για τα μη-επιλεκτικά ερωτήματα μεγάλης εισόδου, όπως τα LQ2 και LQ9, επιτυγχάνει 3–4 φορές μικρότερους χρόνους απόκρισης από το H_2RDF.

2.7.7 Συγκριση αλγορίθμων συνένωσης

Στην ενότητα αυτή, συγκρίνουμε την απόδοση των αλγορίθμων συνένωσης χρησιμοποιώντας διαφορετικά μεγέθη εισόδου. Για να δοκιμαστεί η κλιμακωσιμότητα των αλγορίθμων μας δημιουργούμε το παρακάτω πείραμα. Χρησιμοποιούμε ένα σύμπλεγμα από 25 VMs και το predicate *ud: takesCourse* από το σύνολο δεδομένων LUBM20k. Το predicate αυτό περιέχει 515 εκατομμύρια τριάδες που περιγράφουν τις συνδέσεις μεταξύ των φοιτητών και των μαθημάτων. Χρησιμοποιώντας τυχαία δειγματοληψία, δειγματοληπτούμε τα αντίστοιχα δεδομένα και αποθηκεύουμε τις τριάδες σε ξεχωριστά HBase ευρετήρια. Το Σχήμα 2.6 δείχνει τους χρόνους εκτέλεσης που απαιτούνται για τη συνένωση όλων των δεδομένων του predicate *ud: takesCourse* με τα αντίστοιχα δείγματα δεδομένων χρησιμοποιώντας διάφορους αλγορίθμους συνένωσης. Τα δείγματα τριάδων περιέχουν 5-500 εκατομμύρια τριάδες.

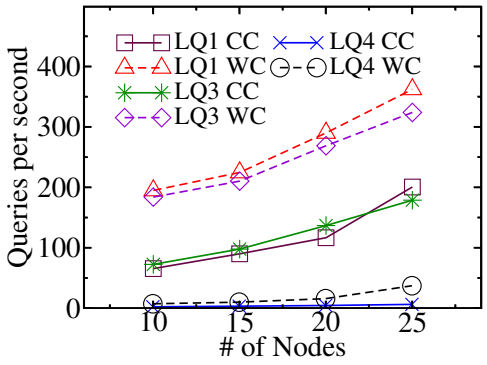


Σχήμα 2.6: *Κλιμακωσιμότητα αλγορίθμων συνένωσης*

Παρατηρούμε ότι για συνενώσεις που περιέχουν ένα επιλεκτικό πρότυπο εισόδου, η πιο αποτελεσματική συνένωση είναι ο κεντρικός Merge αλγόριθμος συνένωσης. Αυτό συμβαίνει επειδή οι MapReduce συνενώσεις επιβαρύνονται πάντα με ένα χρόνο αρχικοποίησης που είναι περίπου 30 δευτερόλεπτα. Η απόδοση των κεντρικών συνενώσεων επιδεινώνεται όσο αυξάνεται το μέγεθος των δεδομένων. Αυτό οφείλεται στο γεγονός ότι οι αλγόριθμοι δεν εκμεταλλεύονται τις δυνατότητες παράλληλης σάρωσης των ευρετηρίων μας.

Σχετικά με τους αλγορίθμους MapReduce συνενώσεων, εξετάζουμε την απόδοση των Merge, Sort-Merge, Partial Input Hash και Full Input Hash [Papailiou 12] αλγορίθμων συνένωσης. Μπορούμε να παρατηρήσουμε ότι ο Merge αλγόριθμος συνένωσης έχει την καλύτερη απόδοση κλιμάκωσης. Αυτό οφείλεται στο γεγονός ότι εκτελεί τη συνένωση με χρήση της ταξινόμησης των δυο σχέσεων και έτσι ελαχιστοποιεί την επιβάρυνση της μετακίνησης των δεδομένων. Όμως αυτός ο αλγόριθμος δεν μπορεί να εκτελεστεί πάνω σε ενδιάμεσα μη-ταξινομημένα δεδομένα. Σε αυτή την περίπτωση, μπορούμε να δούμε ότι ο Sort-Merge αλγόριθμος συνένωσης αποδεικνύεται ως η πιο επεκτάσιμη λύση. Η διαφορά μεταξύ των Sort-Merge και Partial Input Hash αλγορίθμων συνένωσης είναι ο τρόπος κατανομής των δεδομένων στο πλαίσιο του MapReduce. Ο Sort-Merge αλγόριθμος χωρίζει τα δεδομένα χρησιμοποιώντας μια διαμέριση που προέρχεται από τα ταξινομημένα ευρετήριά μας, ενώ ο Partial Input Hash αλγόριθμος χρησιμοποιεί hash τεχνική διαμέρισης. Αυτό έχει αντίκτυπο στη reduce φάση της συνένωσης. Η Sort-Merge συνένωση εκτελεί αποδοτικές σειριακές αναγνώσεις των ευρετηρίων στην reduce φάση. Αντίθετα ο Partial Input Hash αλγόριθμος εκτελεί τυχαίες HBase αναζητήσεις για κάθε κλειδί της μεταβλητής συνένωσης. Η δεύτερη προσέγγιση παρουσιάζει χαμηλή κλιμάκωση, όταν η μικρή είσοδος αυξάνει σε μέγεθος. Τέλος, στην περίπτωση που δεν έχουμε ταξινομημένο ευρετήριο σε καμία είσοδο της συνένωσης, ο Full Input Hash αλγόριθμος συνένωσης παρουσιάζει την καλύτερη απόδοση.

2.7.8 Εκτέλεση επιλεκτικών ερωτημάτων

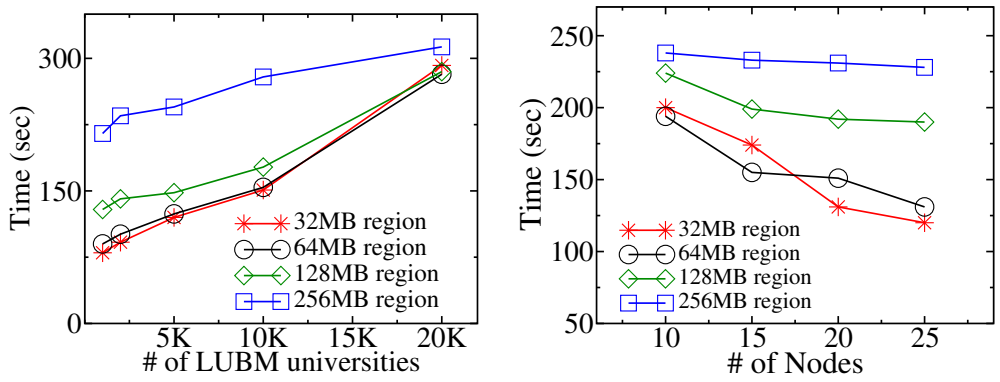


Σχήμα 2.7: *Ρυθμός εκτέλεσης ταυτόχρονων ερωτημάτων*

Στην περίπτωση κεντρικών συνενώσεων, δείχνουμε ότι η ταυτόχρονη εκτέλεση μπορεί να οδηγήσει σε πολύ μεγάλο ρυθμό εκτέλεσης ερωτημάτων. Για να επιτευχθεί αυτό, το H_2RDF+ χρησιμοποιεί ένα zookeeper quorum που είναι υπεύθυνο για τη διανομή των κεντρικών συνενώσεων στους κόμβους του συμπλέγματος. Κάθε κόμβος έχει μια μέγιστη χωρητικότητα συνενώσεων που μπορεί να εκτελέσει ταυτόχρονα, 4 στα πειράματά μας. Όλες οι δοκιμές εκτελούνται χρησιμοποιώντας το σύνολο δεδομένων LUBM5k. Χρησιμοποιούμε 10, 15, 20 και 25 κόμβους για να δείξουμε την επίδραση της αύξησης του μεγέθους του συμπλέγματος στον ρυθμό εκτέλεσης ερωτημάτων. Τα αποτελέσματα για τα ερωτήματα LQ1, LQ3 και LQ4 παρουσιάζονται στο Σχήμα 2.7. Καταγράφουμε το μέσο ρυθμό εκτέλεσης ερωτημάτων ανά δευτερόλεπτο. Τρέχουμε την ίδια δοκιμή δύο φορές για να μετρήσουμε το ρυθμό εκτέλεσης ερωτημάτων με χρήση cold cache(CC) και warm cache (WC). Δεν χρησιμοποιούμε επιπλέον σχήμα προσωρινής αποθήκευσης αποτελεσμάτων, αλλά βασίζονται στο σχήμα κρυφής μνήμης που παρέχεται από την HBase. Παρατηρούμε ότι τα αποτελέσματα ζεστής cache παρουσιάζουν 2 έως 3 φορές υψηλότερη απόδοση σε σύγκριση με την εκτέλεση χωρίς cache. Αυτό δείχνει ότι το σύστημά μας μπορεί να επωφεληθεί από την προσωρινή αποθήκευση. Παρατηρούμε επίσης μια σχεδόν γραμμική αύξηση απόδοσης σε σχέση με τον αριθμό των κόμβων της συστοιχίας. Για παράδειγμα το LQ1 έχει μια απόδοση 65 ερωτήματα/sec (15,4 ms ανά ερώτημα) σε ένα σύμπλεγμα 10 κόμβων (40 ταυτόχρονες συνενώσεις). Αυτή είναι μια επιτάχυνση της τάξεως των 40 φορών, καθώς η ατομική εκτέλεση του LQ1 διαρκεί 0,6 sec. Τα LQ1 και LQ3 έχουν σχεδόν την ίδια απόδοση, γεγονός που οφείλεται στο παρόμοιο κόστος εκτέλεσής τους. Το H_2RDF+ χρειάζεται 0,6 και 0,7 δευτερόλεπτα για να απαντήσει LQ1 και LQ2 αντίστοιχα. Όσο για τη μικρότερη απόδοση για το LQ4, αυτό οφείλεται στην αύξηση του κόστους εκτέλεσής του, καθώς απαιτούνται περίπου 2 δευτερόλεπτα για να απαντηθεί. Το LQ4 παρουσιάζει την ίδια κλιμακωσιμότητα και κέρδος κρυφής μνήμης όπως τα προηγούμενα ερωτήματα.

2.7.9 Κλιμακωσιμότητα ερωτημάτων

Στην ενότητα αυτή αξιολογούμε τις ιδιότητες κλιμάκωσης της κατανεμημένης επεξεργασίας των ερωτημάτων. Χρησιμοποιούμε το LQ9, διότι είναι ένα από τα πιο σύνθετα ερωτήματα που εξετάστηκαν, απαιτώντας τρεις κατανεμημένες συνενώσεις. Δοκιμάζουμε την κλιμάκωση του ερωτήματος χρησιμοποιώντας διαφορετικά μεγέθη σύνολων δεδομένων και αριθμού κόμβων (VMs). Τα αποτελέσματα κλιμακωσιμότητας για το LQ9 παρουσιάζονται στο Σχήμα 2.8. Καταγράφουμε την απόδοση των MapReduce συνενώσεων χρησιμοποιώντας διαφορετικά μεγέθη συνόλων δεδομένων χρησιμοποιώντας ένα σύμπλεγμα 25 κόμβων. Το μέγεθος της εισόδου καθώς και του αποτελέσματος του LQ9 (που επηρεάζουν άμεσα το χρόνο εκτέλεσής του) εξαρτώνται από το μέγεθος του συνόλου δεδομένων. Μια άλλη παράμετρος που δοκιμάζεται εδώ είναι η επίδραση του μεγέθους των HBase regions στην εκτέλεση MapReduce συνενώσεων. Μεγάλα regions εμφανίζουν χαμηλότερες επιδόσεις για μικρά σύνολα δεδομένων, επειδή ο αριθμός των εργασιών που δημιουργείται αδυνατεί να αξιοποιήσει πλήρως τους πόρους συμπλέγματος. Για μεγαλύτερα σύνολα δεδομένων, όλα τα μεγέθη region επιτυγχάνουν καλή απόδοση, γιατί υπάρχουν αρκετές περιοχές ώστε να αξιοποιηθεί πλήρως το σύμπλεγμα. Για μικρότερα μεγέθη περιοχής (64MB ή 32MB) η πολυπλοκότητα είναι σχεδόν γραμμική σε σχέση το μέγεθος των δεδομένων εισόδου.



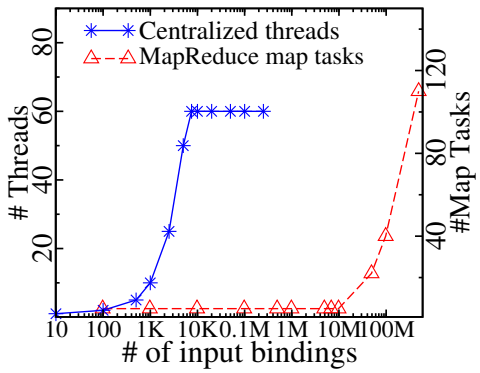
Σχήμα 2.8: *Κλιμακωσιμότητα της κατανεμημένης εκτέλεσης ερωτημάτων για διαφορετικό μέγεθος δεδομένων και αριθμό κόμβων*

Το Σχήμα 2.8 εμφανίζει επίσης το χρόνο εκτέλεσης του ερωτήματος LQ9 καθώς ο αριθμός των κόμβων αυξάνεται. Όλες οι δοκιμές εκτελούνται χρησιμοποιώντας το σύνολο δεδομένων LUBM5k. Μεταβάλουμε το μέγεθος του συμπλέγματος από 10 έως 25 κόμβους καθώς και το μέγεθος περιοχής (region) από 32MB εως 256MB. Στην περίπτωση των 32MB, η εκτέλεση των συνενώσεων είναι εξαιρετικά κλιμακώσιμη, κερδίζοντας μεγάλη επιτάχυνση με την προσθήκη περισσότερων κόμβων. Οι αποκλίσεις από τη γραμμική επιτάχυνση προκαλούνται κυρίως από

το γεγονός ότι ο αριθμός των εργασιών map μπορεί να ταιριάζει καλά ή όχι με τον αριθμό κόμβων. Καθώς το μέγεθος περιοχής μεγαλώνει, οφείλουμε να παρατηρήσουμε ότι η προσθήκη περισσότερων κόμβων δεν επηρεάζει σημαντικά την επιτάχυνση. Αυτό οφείλεται στο γεγονός ότι τα μεγαλύτερα μεγέθη περιοχής συνεπάγονται λιγότερες εργασίες που δεν μπορούν να αξιοποιήσουν πλήρως τους πόρους συμπλέγματος.

2.7.10 Βελτιστοποιητής πλάνου εκτέλεσης και ελαστική διάθεση πόρων

Στην ενότητα αυτή συγκρίνουμε τις επιδόσεις των αλγορίθμων συνένωσης ανάλογα με τους πόρους εκτέλεσης που τους ανατίθενται. Επιπλέον, παρουσιάζουμε τις προσαρμοστικές ιδιότητες εκτέλεσης του συστήματός μας. Για να δοκιμαστεί η κλιμακωσιμότητα των αλγορίθμων μας επαναλαμβάνουμε το πείραμα μεταβλητού μεγέθους συνένωσης που παρουσιάστηκε στην ενότητα 2.7.7. Το Σχήμα 2.9 δείχνει τον αριθμό των υπολογιστικών πόρων που καταλαμβάνεται από τους αλγορίθμους μας για τη συνένωση των τριάδων του predicate *ud: takesCourse* από το συνόλου δεδομένων LUBM1k (25 εκατομμύρια τριάδες) με εκείνες των διαφόρων δειγμάτων.



Σχήμα 2.9: *Ελαστική διάθεση πόρων*

Οι αλγόριθμοι συνένωσής μας μπορεί να εκτελεστούν με κεντρικό ή κατανεμημένο τρόπο. Το Σχήμα 2.9 απεικονίζει τον αριθμό των πόρων που αφιερώνονται στην εκτέλεση κάθε συνένωσης. Μπορούμε να δούμε ότι το ύψος των πόρων που δεσμεύεται είναι ανάλογο του κόστους της συνένωσης. Στην περίπτωση των κεντρικών συνενώσεων, ο βελτιστοποιητής μας ρυθμίζει τον αριθμό των ταυτόχρονων νημάτων, ενώ στην περίπτωση της χρήσης MapReduce επιλέγει τον αριθμό των map tasks. Στο συγκεκριμένο πείραμα έχουμε ρυθμίσει τον βελτιστοποιητή να ξεκινά ένα επιπλέον νήμα εκτέλεσης μόνο όταν υπάρχει ένα ελάχιστο ποσό 100 εγγραφών. Θέτουμε επίσης το μέγιστο αριθμό των ταυτόχρονων νημάτων στα 60.

Όσον αφορά τις κατανεμημένες συνενώσεις ο βελτιστοποιητής μας μπορεί να αποφασίσει τον αριθμό των map εργασιών σύμφωνα με το κόστος της συνένωσης. Αυτό γίνεται με

την εύρεση της μέγιστης σάρωσης εισόδου, δηλαδή της σάρωσης που καλύπτει τις περισσότερες περιοχές HBase. Κατά την εύρεση της μέγιστης σάρωσης, ένα map task εκχωρείται σε κάθε μια από τις HBase περιοχές της συγκεκριμένης σάρωσης. Καθώς το μέγεθος των τριάδων στο δείγμα κυμαίνεται από 10 έως 500 εκατομμύρια τριάδες το μέγεθος της μεγαλύτερης σάρωσης μεταβάλλεται. Όταν οι τριάδες του δείγματος είναι λιγότερες από 25 εκατομμύρια, το *ud: takesCourse* είναι η μέγιστη σάρωση και εκτείνεται σε 4 περιοχές. Όταν οι τριάδες του δείγματος μεγαλώνουν γίνονται αυτές η μεγαλύτερη σάρωση και έτσι τα map task αυξάνονται ανάλογα με το μέγεθός τους.

ΚΕΦΑΛΑΙΟ 3

Κρυφή Μνημη SPARQL ερωτημάτων

3.1 Εισαγωγή

Η απομάκρυνση του σχήματος από την περιγραφή των δεδομένων RDF επιτρέπει τον σχηματισμό ενός κοινού πλαισίου που διευκολύνει την ενσωμάτωση διαφόρων πηγών δεδομένων. Αυτή η ιδιότητα έχει οδηγήσει σε μια άνευ προηγουμένου αύξηση του ρυθμού με τον οποίο παράγονται, αποθηκεύονται και ερωτώνται RDF δεδομένα, ακόμα και έξω από το καθαρά ακαδημαϊκό χώρο[1] [2] . Κατά συνέπεια τα τελευταία χρόνια παρακολουθούμε την ανάπτυξη πολλών μηχανών [Weiss 08, Neumann 10a, Zeng 13, Atre 08, Papailiou 13, Bonstrom 03] που στοχεύουν την αποθήκευση και ευρετηρίαση των RDF δεδομένων καθώς και την αποτελεσματική επεξεργασία SPARQL ερωτημάτων.

Ωστόσο, η μετάβαση από τις εξαρτώμενες στο σχήμα σχεσιακές βάσεις στα δεδομένα RDF που δεν περιέχουν αντίστοιχες πληροφορίες, έφερε νέες προκλήσεις για την αποδοτική ευρετηρίαση και την αναζήτηση των RDF δεδομένων. Ευρέως χρησιμοποιούμενες τεχνικές που βασίζονταν στο σχήμα των δεδομένων δεν μπορούν εύκολα να επεκταθούν στο πλαίσιο RDF. Για παράδειγμα:

- Η ομαδοποίηση των δεδομένων με χρήση πινάκων.
- Η ευρετηρίαση με βάση πεδία που χρησιμοποιούνται συχνά σε φίλτρα ή σε συνενώσεις.

[1] http://www.bbc.co.uk/blogs/bbcinternet/2012/04/ sports_dynamic_semantic.html
[2] http://data.gov.uk/

- Η δημιουργία όψεων με βάση συχνά χρησιμοποιούμενα μοτίβα ερωτημάτων.

Στην πραγματικότητα, οι βάσεις δεδομένων RDF θεωρούν ότι υπάρχει περιορισμένη γνώση της δομής των δεδομένων. Αυτό συμβαίνει συνήθως μέσω RDFS τριάδων [Brickley 14]. Ωστόσο, οι RDFS πληροφορίες δεν είναι τόσο πλούσιες και υποχρεωτικές όσο το SQL σχήμα. Μπορεί να είναι ελλιπείς και αλλάζουν γρήγορα, μαζί με τα δεδομένων. Ως εκ τούτου, οι περισσότερες βάσεις δεδομένων RDF στοχεύουν την ευρετηρίαση των RDF τριάδων. Αυτό έχει ως αποτέλεσμα την χρήση πολύ μεγαλύτερου αριθμού συνενώσεων σε σχέση με τις αντίστοιχες σχεσιακές βάσεις. Για παράδειγμα, το ερώτημα 2 του TPC-H γραμμένο σε SPARQL απαιτεί 26 συνενώσεις ενώ το αντίστοιχο SQL περιέχει μόνο 5 [Gubichev 14]. Σε αντίθεση, οι RDF βάσεις δεδομένων που χρησιμοποιούν RDFS πληροφορίες για την αποθήκευση και την ομαδοποίηση των δεδομένων [Stuckenschmidt 04, Tran 10], αδυνατούν να προσαρμοστούν αποτελεσματικά στις αλλαγές σχήματος και σε δεδομένα που δεν ταιριάζουν στο σχήμα.

Σημειώνουμε ότι οι πιο εξελιγμένες βάσεις γράφων [Zhao 07, Yan 04] χρησιμοποιούν εκτενώς τεχνικές που στοχεύουν στην ευρετηρίαση συχνά εμφανιζόμενων μοτίβων γράφων. Ωστόσο, οι RDF βάσεις δεδομένων δεν έχουν λάβει ακόμη τα πλεονέκτημα τέτοιων των τεχνικών. Επιπροσθέτως, τα συστήματα αυτά επικεντρώνονται στην *στατική* ευρετηρίαση των σημαντικών μοτίβων γραφημάτων, δηλαδή η εύρεση τέτοιων μοτίβων βασίζεται αποκλειστικά στο υπό επεξεργασία σύνολο δεδομένων, χωρίς καμία μέριμνα για το σύνολο ερωτημάτων. Ωστόσο, η ποικιλομορφία των εφαρμοζόμενων SPARQL ερωτημάτων σε συνδυασμό με την απαίτηση για υψηλές επιδόσεις σε όλα τα διαφορετικά σύνολα ερωτημάτων, καλεί για μια δυναμική, βασισμένη στα ερωτήματα ευρετηρίαση (π.χ., [Idreos 11]).

Επιπλέον, τα ερωτήματα SPARQL τείνουν να αυξάνουν σε μέγεθος και πολυπλοκότητα παρουσιάζοντας προκλήσεις στη βελτιστοποίηση τους με χρήση αλγορίθμων δυναμικού προγραμματισμού. Για παράδειγμα, η DBpedia αναφέρει ότι τα SPARQL ερώτημα που εκτελούνται περιέχουν ερωτήσεις με έως και 10 τριπλέτες [Gallego 11]. Ερωτήματα στον τομέα της βιοϊατρικής μπορεί να περιλαμβάνουν περισσότερες από 50 τριπλέτες [Sahoo 10]. Τα πειραματικά αποτελέσματα δείχνουν ότι η εξεύρεση του βέλτιστου πλάνου εκτέλεσης χρησιμοποιώντας προσεγγίσεις δυναμικού προγραμματισμού μπορεί να παρουσιάζει απαγορευτικό χρόνο εκτέλεσης για ερωτήματα με περισσότερες από 15 τριπλέτες [Neumann 10a, Gubichev 14]. Ενώ υπάρχουν πολλοί άπληστοι και ευριστικοί αλγόριθμοι για την βελτιστοποίηση SPARQL ερωτημάτων [Tsialiamanis 12, Papailiou 13], μπορούν να επιλέξουν μη βέλτιστα πλάνα εκτέλεσης, αυξάνοντας το χρόνο εκτέλεσης του ερωτήματος [Gubichev 14] και δεν μπορούν εύκολα να ενσωματωθούν με αλγορίθμους κρυφής μνήμης που εξετάζουν *όλα* τα αποτελέσματα που μπορούν να χρησιμοποιηθούν για την εκτέλεση ενός ερωτήματος.

Ως αποτέλεσμα, η δεύτερη συνεισφορά αυτής της διατριβής (Κεφάλαιο 3) είναι η προσαρμοστική ευρετηρίαση και η δημιουργία κρυφής μνήμης SPARQL αποτελεσμάτων που μπορεί

να χρησιμοποιηθεί προκειμένου να εντοπιστούν συχνά εμφανιζόμενες μορφές ερωτημάτων. Προτείνουμε ένα πλαίσιο κρυφής μνήμης που προσφέρει τη δυνατότητα επαναχρησιμοποίησης των υπολογιζόμενων αποτελεσμάτων SPARQL. Παρακολουθώντας ενεργά το φόρτο εργασίας SPARQL, το σύστημά μας μπορεί να ανιχνεύσει συχνές μορφές ερωτημάτων και να προκαλέσει την εκτέλεση και προσωρινή αποθήκευση τους, προκειμένου να ενισχύσει την απόδοση των επακόλουθων ερωτημάτων και την προσαρμογή στο φόρτο εργασίας. Επίσης παρουσιάζουμε έναν αλγόριθμο απλοποίησης SPARQL ερωτημάτων, που χρησιμοποιείται για να χειριστεί πολύπλοκα γραφήματα. Εν συντομία, η διατριβή αυτή προτείνει:

- Έναν αλγόριθμο απλοποίησης SPARQL ερωτημάτων (Ενότητα 3.2), με βάση τεχνικές απλοποίησης αστέρων, που χωρίζει το ερώτημα σε έναν σκελετό και πολλαπλά γραφήματα αστέρων.

- Έναν αλγόριθμο δημιουργίας κανονικοποιημένων ετικετών για SPARQL ερωτήματα (κεφάλαιο 3.3) που είναι σε θέση να παράγει κανονικοποιημένων ετικέτες που είναι ίδιες για όλες τις ισομορφικές αναπαραστάσεις ενός ερωτήματος SPARQL. Αυτές οι ετικέτες χρησιμοποιούνται ως κλειδιά για την δημιουργία μιας κρυφής μνήμης SPARQL ερωτημάτων. Επιπλέον, η τεχνική απλούστευσης των ερωτημάτων καταφέρνει να μειώσει αποτελεσματικά την πολυπλοκότητα του αλγορίθμου δημιουργίας κανονικοποιημένων ετικετών για πολύπλοκες δομές ερωτημάτων.

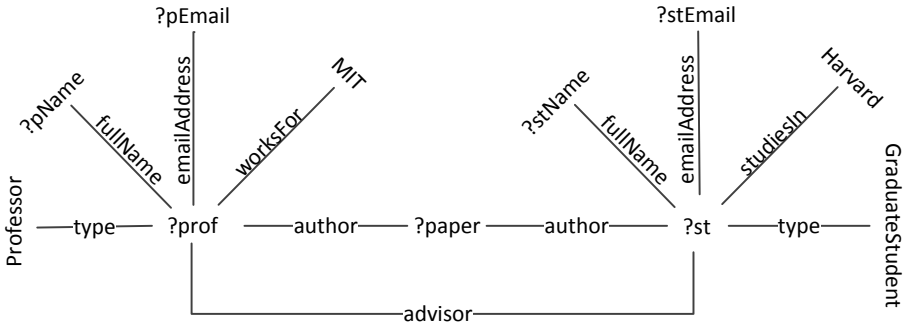
- Επεκτείνουμε ένα *Βελτιστοποιητή Δυναμικού Προγραμματισμού* [Moerkotte 08] προκειμένου να πραγματοποιεί αιτήματα κρυφής μνήμης για όλους τους υπογράφους του ερωτήματος, χρησιμοποιώντας *κανονικοποιημένες ετικέτες* (Ενότητα 3.4). Το προκύπτων βέλτιστο πλάνο εκτέλεσης μπορεί έτσι να περιλαμβάνει, εν μέρει προσωρινά αποθηκευμένα αποτελέσματα υπογράφων.

- Ένας *Ελεγκτής Κρυφής Μνήμης* παρακολουθεί όλα τα αιτήματα κρυφής μνήμης επιτρέποντας την ανίχνευση *κερδοφόρων* μορφών ερωτημάτων. Ο ελεγκτής προκαλέσει την εκτέλεση και την προσωρινή αποθήκευση τέτοιων ερωτημάτων για να βελτιώσει τη χρησιμοποίηση της κρυφής μνήμης.

Η προτεινόμενη κρυφή μνήμη έχει αρθρωτό σχεδιασμό, επιτρέποντας την ενσωμάτωση με διάφορες RDF βάσεις δεδομένων. Ενσωματώνοντάς την στο H_2RDF+ αποδεικνύουμε ότι η προσαρμοστική δημιουργία κρυφή μνήμης μπορεί να μειώσει το μέσο χρόνο απόκριση των SPARQL ερωτημάτων έως και δύο τάξεις μεγέθους, προσφέροντας μικρούς χρόνους απόκρισης για σύνθετα σύνολα ερωτημάτων και μεγάλες συλλογές RDF δεδομένων.

3.2 Απλοποίηση SPARQL ερωτημάτων

Υπάρχουν δύο βασικοί τρόποι αναπαράστασης ενός SPARQL ερωτήματος με χρήση γράφων. Η πρώτη αναπαράσταση είναι ο γράφος του SPARQL ερωτήματος και χρησιμοποιεί κόμβους που αντιστοιχούν στις μεταβλητές του ερωτήματος και ακμές που αναπαριστούν τα ερωτήματα τριάδας που συνδέουν τις διάφορες μεταβλητές. Αυτή η μορφή παρουσιάζει τη δομή του ερωτήματος αν θεωρήσουμε ότι οι RDF τριάδες είναι ακμές ενός γράφου δεδομένων. Για παράδειγμα, το Σχήμα 3.1 απεικονίζει το γράφο του παρακάτω SPARQL ερωτήματος:

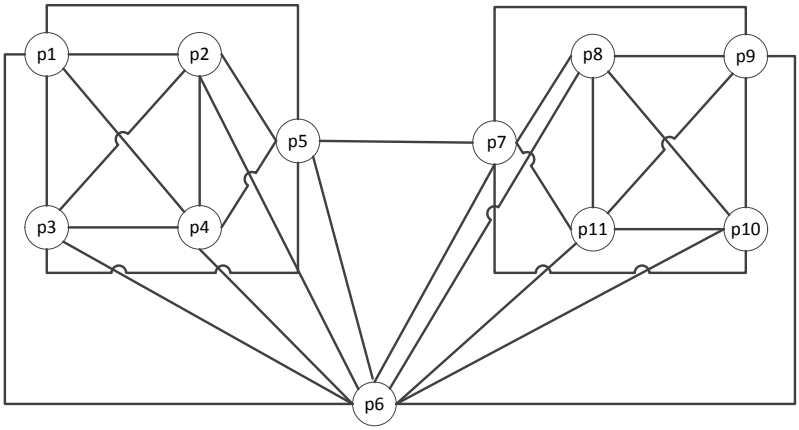


Σχήμα 3.1: *Γράφος του SPARQL ερωτήματος Q_e*

Example query Q_e:
*select * where {*
?prof type Professor . (p1)
?prof fullName ?pName. (p2)
?prof emailAddress ?pEmail . (p3)
?prof worksFor "MIT" . (p4)
?prof author ?paper . (p5)
?prof advisor ?st . (p6)
?st author ?paper . (p7)
?st fullName ?stName . (p8)
?st emailAddress ?stEmail . (p9)
?st studiesIn "Harvard" . (p10)
?st type GraduateStudent . (p11)}

Ένας άλλος τρόπος αναπαράστασης ενός SPARQL ερωτήματος είναι ο *γράφος συνενώσεων*. Σε αυτή την αναπαράσταση, κάθε ερώτημα τριάδας έχει μετατραπεί σε έναν κόμβο και δύο κόμβοι συνδέονται, αν μοιράζονται μια κοινή μεταβλητή. Ως εκ τούτου, οι κόμβοι αντιπροσωπεύουν σαρώσεις των RDF δεδομένων που ταιριάζουν με το κάθε επιμέρους ερώτημα

τριάδας. Οι ακμές του γραφήματος αντιστοιχούν στις συνενώσεις που πρέπει να εκτελεστούν προκειμένου να απαντηθεί το ερώτημα. Αυτό το γράφημα μοιάζει με τους γράφους συνενώσεων που χρησιμοποιούνται για τη βελτιστοποίηση ερωτημάτων στις παραδοσιακές σχεσιακές βάσεις δεδομένων. Στο παράδειγμά μας ο γράφος συνενώσεων για το ερώτημα Q_e παρουσιάζεται στο Σχήμα 3.2.



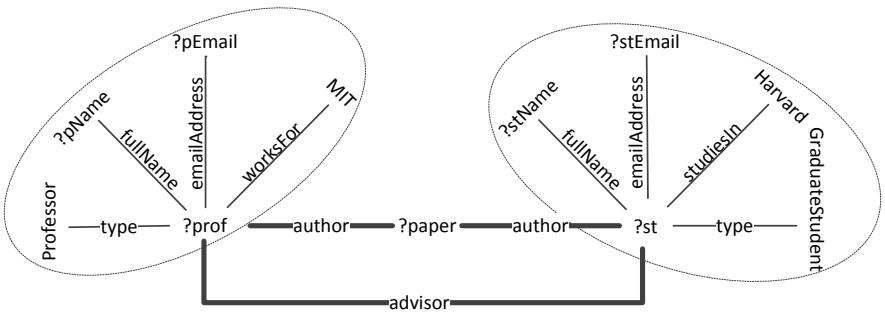
Σχήμα 3.2: *Γράφος συνενώσεων του ερωτήματος* Q_e

Η εύρεση του βέλτιστου πλάνου εκτέλεσης ενός SPARQL ερωτήματος είναι μια απαιτητική εργασία, λόγω της πολυπλοκότητας των γράφων συνενώσεων. Όπως αναφέρθηκε προηγουμένως, τα ερωτήματα SPARQL είναι μεγάλα σε μέγεθος και μπορεί σε κάποιες περιπτώσεις να περιέχουν μέχρι και 50 ερωτήματα τριάδας. Η χρήση αλγορίθμων δυναμικού προγραμματισμού για τον έλεγχο όλων των πιθανών πλάνων εκτέλεσης του ερωτήματος γίνεται απαγορευτική όταν τα ερωτήματα έχουν περισσότερα από 10 ερωτήματα τριάδας [Gubichev 14]. Επιπλέον, τα ερωτήματα SPARQL περιγράφονται σε μορφή RDF γράφων χωρίς καμία πληροφορία για το σχήμα των δεδομένων. Η έλλειψη γνώσης για το σχήμα των δεδομένων καθώς και για τις δυνατότητες ομαδοποίησής τους οδηγεί στην επανεκτέλεση συχνά εμφανιζόμενων προτύπων ερωτημάτων με χρήση συνενώσεων. Για παράδειγμα, ένα συχνό πρόβλημα είναι η ανάκτηση επιπρόσθετων στοιχείων για μια συγκεκριμένη οντότητα (π.χ. όνομα, e-mail και αριθμό τηλεφώνου για έναν καθηγητή). Ενώ αυτό το ερώτημα θα απαιτούσε μία λειτουργία αναζήτησης και ανάγνωσης από έναν πίνακα σε μια σχεσιακή βάση δεδομένων, με χρήση SPARQL περιγράφεται ως ένα γράφος με μορφή αστέρα που για την απάντησή του απαιτούνται συνενώσεις των διαφόρων σαρώσεων των RDF δεδομένων.

Τα μοτίβα αστέρων χρησιμοποιούνται ευρέως στα SPARQL ερωτήματα [Gubichev 14] όμως μετατρέπονται σε κλίκες στον γράφο συνενώσεων του ερωτήματος. Αυτό οφείλεται στο γεγονός ότι *όλα* τα ερωτήματα τριάδας ενός μοτίβου αστέρα πρέπει να συνενωθούν με βάση την κοινή τους μεταβλητή. Οι αλγόριθμοι βελτιστοποίησης πλάνων εκτέλεσης χρησιμοποιούν τον

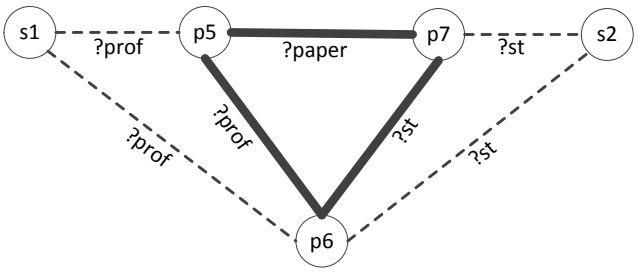
γράφο συνενώσεων για την παραγωγή του βέλτιστου πλάνου εκτέλεσης του ερωτήματος και συνεπώς, η παρουσία μεγάλων κλικών στο γράφο συνενώσεων οδηγεί σε εκθετική πολυπλοκότητα [Moerkotte 08]. Η δημιουργία κανονικοποιημένων ετικετών για SPARQL ερωτήματα βασίζεται επίσης στον γράφο συνενώσεων. Όπως παρουσιάσαμε στο [Papailiou 15], τόσο ή δημιουργία ετικετών όσο και η χρήση τεχνικών δυναμικού προγραμματισμού για τον έλεγχο χρησιμοποίησης των προσωρινά αποθηκευμένων αποτελεσμάτων αντιμετωπίζουν εκθετική πολυπλοκότητα για τα κοινώς χρησιμοποιούμενα μοτίβα αστέρων στα SPARQL ερωτήματα.

Ωστόσο, η βελτιστοποίηση των ερωτημάτων με μορφή αστέρα έχει επιτυχώς αντιμετωπιστεί με ευριστικές τεχνικές. Οι περισσότερες βάσεις RDF δεδομένων χρησιμοποιούν πολλαπλά ή όλα τα δυνατά RDF ευρετήρια [Weiss 08, Neumann 10a, Papailiou 13]. Αφού τα δεδομένα μια συνένωσης μπορούν να ανακτηθούν με διάφορες ταξινομήσεις, το πιο αποδοτικό πλάνο εκτέλεσης ενός ερωτήματος αστέρα είναι μια σειρά από 2-way merge συνενώσεις [Neumann 10a, Gubichev 14] ή μια multi-way merge συνένωση [Papailiou 13]. Στην περίπτωση των multi-way συνενώσεων [Papailiou 13] στην πραγματικότητα δεν υπάρχει εναλλακτικό πλάνο εκτέλεσης για την απάντηση του ερωτήματος αστέρα. Επιπλέον, όταν χρησιμοποιείτε μια ακολουθία 2-way συνενώσεων [Neumann 10a], ευριστικοί αλγόριθμοι όπως τα characteristic sets [Neumann 11] μπορούν να χρησιμοποιηθούν για την εκτίμηση του μεγέθους των ενδιάμεσων αποτελεσμάτων. Για την εύρεση της σειράς εκτέλεσης των συνενώσεων μπορεί να χρησιμοποιηθεί η τεχνική του hierarchical characterisation [Gubichev 14] που μπορεί να παρέχει σχεδόν βέλτιστα πλάνα εκτέλεσης για ερωτήματα αστέρος σε γραμμικό χρόνο, χωρίς να εξετάσει όλες τις πιθανές αναδιατάξεις των συνενώσεων. Ως εκ τούτου, ενώ τα μοτίβα αστέρα περιγράφονται με περίπλοκα γραφήματα κλίκας στον γράφο των συνενώσεων, μπορούμε να βρούμε καλά πλάνα εκτέλεσης με χρήση ευριστικών τεχνικών που αποφεύγουν την πολυπλοκότητα της απαρίθμησης όλων των αναδιατάξεων των συνενώσεών τους.



Σχήμα 3.3: *Απλοποιημένος γράφος ερωτήματος SPARQL για το ερώτημα* Q_e

Σε αυτήν την εργασία, προτείνουμε μια μέθοδο απλοποίησης SPARQL ερωτημάτων που μειώνει την πολυπλοκότητα του γράφου συνενώσεων του ερωτήματος, ενώ μπορεί να χρησιμοποιηθεί τόσο για την παροχή σχεδόν βέλτιστων πλάνων εκτέλεσης όσο και για τον εντοπισμό όλων των προσωρινά αποθηκευμένων αποτελεσμάτων που μπορούν να χρησιμοποιηθούν για την απάντηση του ερωτήματος. Η προσέγγιση απλούστευσής μας χωρίζει τον αρχικό γράφο του ερωτήματος σε ένα γράφο σκελετό και σε πολλαπλούς γράφους αστέρα. Οι υπογράφοι αστέρα αφαιρούνται από τον αρχικό γράφο του ερωτήματος και αντικαθίστανται από γενικευμένους κόμβους αστέρων. Για το παράδειγμά μας, το Σχήμα 3.3 απεικονίζει το γράφο σκελετού του ερωτήματος Q_e χρησιμοποιώντας έντονες γραμμές στις ακμές του. Οι κόμβοι αστέρων απεικονίζονται με διακεκομμένες ελλείψεις. Η απλοποιημένη αυτή μορφή του γράφου του ερωτήματος οδηγεί σε μια επίσης απλοποιημένη αναπαράσταση του γράφου συνενώσεων η οποία απεικονίζεται στο Σχήμα 3.4. Σημειώνουμε ότι σε σύγκριση με τον αρχικό γράφο συνενώσεων, που απεικονίζεται στο Σχήμα 3.2, ο απλοποιημένος γράφος συνενώσεων αποτελείται από πολύ λιγότερους κόμβους και είναι πιο χαλαρά συνδεδεμένος, γεγονός που οφείλεται στην αντικατάσταση ολόκληρων υπογράφων μορφής κλίκας με απλούς κόμβους.



Σχήμα 3.4: *Απλοποιημένος γράφος συνενώσεων του ερωτήματος Q_e*

Η αλγοριθμική περιγραφή της διαδικασίας απλοποίησης των SPARQL ερωτημάτων μπορεί να βρεθεί στον Αλγόριθμο 3. Ο αλγόριθμος παίρνει ως είσοδο το γράφο του SPARQL ερωτήματος, καθώς και τον αντίστοιχο γράφο συνενώσεων και παράγει τον απλοποιημένο γράφο συνενώσεων, ένα σύνολο που περιέχει τα ερωτήματα σκελετού και ένα λεξικό που περιέχει τα ερωτήματα αστέρα ομαδοποιημένα με βάση το starID τους (ένα μοναδικό αναγνωριστικό που αποδίδεται στο κάθε μοτίβο αστέρα). Για να βρούμε τα ερωτήματα σκελετού, επεξεργαζόμαστε όλα τα ερωτήματα τριάδας του αρχικού γράφου. Κάθε ερώτημα τριάδας ανήκει στο *σκελετό* ή σε ένα μοτίβο *αστέρα*. Αν το ερώτημα τριάδας περιλαμβάνει περισσότερες από μία μεταβλητές που χρειάζονται συνένωση χαρακτηρίζεται ως ερώτημα σκελετού. Οι μεταβλητές που απαιτούν συνένωση είναι αυτές που ανήκουν σε περισσότερα από 2 ερωτήματα τριάδας. Ένα ερώτημα τριάδας που περιέχει 2 ή περισσότερες μεταβλητές που απαιτούν συνένωση είναι μέρος μιας διαδρομής στο γράφημα του SPARQL ερωτήματος και έτσι ανήκει στο σκελετό του ερωτήματος. Αντίθετα, αν ένα ερώτημα τριάδας περιέχει μόνο μία μεταβλητή συνένωσης, τότε

ανήκει στο μοτίβο αστέρα που σχηματίζεται γύρω από αυτή τη μεταβλητή και προστίθεται στο λεξικό αστέρων χρησιμοποιώντας ως κλειδί το starID που έχει εκχωρηθεί στη συγκεκριμένη μεταβλητή. Η διάκριση αυτή μπορεί να δημιουργήσει υπογράφους αστέρα που περιέχουν ένα ή περισσότερα ερωτήματα τριάδας.

Algorithm 3: Απλοποίηση ερωτήματος SPARQL

Input: Γράφος ερωτήματος SPARQL $G_q = (V_q, E_q)$, Γράφος συνενώσεων $G_j = (V_j, E_j)$
Output: Υπογράφοι αστέρα $St = \langle s, L(s) \rangle$¨: λεξικό από (starID, λίστα αστέρων), σύνολο ερωτημάτων σκελετού Sk, απλοποιημένος γράφος συνενώσεων $G_{sj} = (V_{sj}, E_{sj})$

$Sk \leftarrow \emptyset$; $St \leftarrow \emptyset$;
for $tp \in E_q$ **do**
  if $tp.getJoinVariables() > 1$ **then**
    $Sk \leftarrow Sk \cup tp$;
  else
    $St \leftarrow St.add(tp.getJoinVariableStarId(), tp)$;
  end
end
$V_{sj} \leftarrow Sk \cup St.keySet()$;
$E_{sj} \leftarrow \emptyset$;
for $(p_s, p_d) \in E_j$ **do**
  if $p_s \in St \vee p_d \in St$ **then**
    $E_{sj} \leftarrow E_{sj} \cup replaceTpIdwithStarId(p_s, p_d)$;
  else
    $E_{sj} \leftarrow E_{sj} \cup (p_s, p_d)$;
  end
end

Αφού εντοπίσουμε τα μοτίβα σκελετού και αστέρα, δημιουργούμε τον απλοποιημένο γράφο συνενώσεων χρησιμοποιώντας ως κόμβους τα μοτίβα σκελετού και έναν κόμβο ανά μοτίβο αστέρα. Οι ακμές του αρχικού γράφου συνενώσεων μετατρέπονται στον απλοποιημένο γράφο μόνο εάν συνδέουν πάνω από έναν κόμβο σκελετού. Αν η προέλευση ή ο προορισμός τους, είναι ένα ερώτημα τριάδας αστέρα τότε οι ακμές μετασχηματίζονται ώστε να συνδέουν τον αντίστοιχο κόμβο αστέρα στο απλοποιημένο γράφημα. Η ακραία περίπτωση των μοτίβων αστέρα που περιέχουν μόνο ένα ερώτημα τριάδας δεν επηρεάζει τον γράφο των συνενώσεων. Η πολυπλοκότητα του αλγορίθμου απλοποίησης ερωτημάτων είναι γραμμική σε σχέση με το μέγεθος του γράφου του ερωτήματος, αφού εξετάζει ακριβώς μία φορά κάθε ερώτημα τριάδας για να αποφασίσει αν ανήκει στο σκελετό του ερωτήματος ή σε ένα υπογράφημα αστέρα.

3.3 Κανονικοποιημένες ετικέτες για SPARQL ερωτήματα

Σε αυτή την ενότητα προτείνουμε ένα αλγόριθμο δημιουργίας κανονικοποιημένων ετικετών για SPARQL γράφους. Επίσης επεκτείνουμε τον βασικό αλγόριθμο ώστε να μπορεί να

υπολογίζει κανονικοποιημένες ετικέτες με βάση τον απλοποιημένο γράφο συνενώσεων. Αποφεύγουμε έτσι την εκθετική πολυπλοκότητα που εισάγουν οι υπογράφοι μορφής κλίκας του γράφου συνενώσεων.

Η ευρετηρίαση μοτίβων γράφων είναι ένα δύσκολο έργο, διότι απαιτεί την αντιμετώπιση του προβλήματος του ισομορφισμού [Köbler 94], το οποίο αποτελεί ένα θεμελιώδες πρόβλημα στη θεωρία γραφημάτων. Αυτό το πρόβλημα προκύπτει όταν το ίδιο μοτίβο ερωτήματος εμφανίζεται με μικρές αποκλίσεις, όπως η αναδιάταξη των ερωτημάτων τριάδας, η μετονομασία των μεταβλητών κλπ. Για παράδειγμα, τα ακόλουθα ερωτήματα SPARQL είναι ισομορφικά.

?prof worksFor "MIT" .	*?v1 studiesIn "Harvard" .*
?prof author ?paper .	*?v2 author ?v3 .*
?st author ?paper .	*?v2 worksFor "MIT" .*
?st studiesIn "Harvard"	*?v1 author ?v3*

Όλοι οι ισομορφισμοί ενός ερωτήματος SPARQL πρέπει να εντοπίζονται και να συνδέονται με την ίδια εγγραφή προκειμένου η υλοποίηση της κρυφής μνήμης να είναι αποδοτική. Για την αντιμετώπιση αυτού του προβλήματος επεκτείνουμε ένα αλγόριθμο κανονικοποιημένων ετικετών γράφων και εισάγουμε την έννοια των κανονικοποιημένων ετικετών για SPARQL ερωτήματα.

Definition Ένας αλγόριθμος δημιουργίας ετικετών C παίρνει ως είσοδο ένα γράφο G και παράγει μια μοναδική ετικέτα L=C(G). Ο C είναι ένας κανονικοποιημένος αλγόριθμος ετικετών γράφων εάν και μόνο εάν για κάθε γράφημα H ισομορφικό του G έχουμε C(G)=C(H). Ονομάζουμε L την κανονικοποιημένη ετικέτα του G. Επιπλέον, η L εισάγει μια κανονικοποιημένη διάταξη των κόμβων του G.

Το πρόβλημα της δημιουργίας κανονικοποιημένων ετικετών μοιράζεται την ίδια υπολογιστική πολυπλοκότητα με το πρόβλημα του ισομορφισμού γράφων και ανήκει στην κατηγορία πολυπλοκότητας GI. Η GI είναι μια από τις λίγες ανοιχτές κλάσεις πολυπλοκότητας αφού δεν είναι γνωστό να είναι πολυωνυμικού ή εκθετικού χρόνου [Köbler 94, Hartke 09]. Υπάρχουν όμως πολλές ενδείξεις ότι η GI δεν είναι NP-complete και επίσης υπάρχουν πολλοί αποτελεσματικοί αλγόριθμοι ανοικτού κώδικα που λύνουν το πρόβλημα του ισομορφισμού και της δημιουργίας κανονικοποιημένων ετικετών για γράφους [McKay 14, Junttila 07, Darga 08] και είναι σε θέση να χειριστούν μεγάλα μεγέθη γράφων. Ένας από τα πρώτους και πιο ισχυρούς αλγορίθμους κανονικοποιημένων ετικετών είναι ο *nauty* του McKay [McKay 81] που εισήγαγε μια καινοτόμο χρήση των αυτομορφισμών για την μείωση του χώρου αναζήτησης των ισομορφισμών ενός γράφου. Ο *Bliss* [Junttila 07] επεκτείνει τον *nauty* εισάγοντας κάποιες επιπλέον

ευριστικές που ενισχύουν την απόδοσή του σε δύσκολα γραφήματα. Τέτοιοι αλγόριθμοι μπορούν να υπολογίζουν κανονικοποιημένες ετικέτες για γραφήματα με χιλιάδες κόβους σε χιλιοστά του δευτερολέπτου, καθιστώντας τους ιδανικούς για το χειρισμό ακόμα και των πιο περίπλοκων SPARQL ερωτημάτων. Ωστόσο, η πιο περιγραφική μορφή γράφου που υποστηρίζεται από τους παραπάνω αλγορίθμους είναι το κατευθυνόμενο με χρωματισμένες κορυφές γράφημα.

Definition Ένας κατευθυνόμενος με χρωματισμένες κορυφές γράφος είναι ένας γράφος G=(V,E,c), όπου το $V = \{1, 2, ..., n\}$ είναι ένα σύνολο κορυφών, το $E \subseteq V \times V$ είναι ένα σύνολο κατευθυνόμενων ακμών και η $c : V \rightarrow N$ είναι μια συνάρτηση χρωματισμού που αναθέτει σε κάθε κορυφή ένα θετικό ακέραιο (χρώμα).

Για να χρησιμοποιήσουμε τους υπάρχοντες αλγορίθμους κανονικοποιημένων ετικετών προτείνουμε έναν μετασχηματισμό των ερωτημάτων SPARQL σε κατευθυνόμενους με χρωματισμένες κορυφές γράφους. Τα μετασχηματισμένα γραφήματα SPARQL μπορούν στη συνέχεια να χρησιμοποιηθούν για την παραγωγή κανονικοποιημένων ετικετών με χρήση ενός από τους προαναφερθέντες αλγορίθμους. Η προτεινόμενη μετατροπή εγγυάται ότι δεν χάνεται καμία πληροφορία του SPARQL ερωτήματος. Αποφεύγεται επίσης η εισαγωγή false positives ή false negatives, δηλαδή, μη-ισόμορφικών SPARQL ερωτημάτων, που παράγουν την ίδια ετικέτα ή ισομορφικών ερωτήματα που έχουν διαφορετικές ετικέτες.

3.3.1 Δημιουργία κανονικοποιημένων ετικετών για γράφους συνενώσεων SPARQL

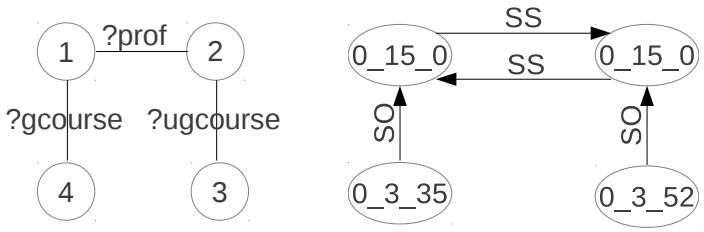
Στην ενότητα αυτή, παρουσιάζουμε τον αλγόριθμο δημιουργίας κανονικοποιημένων ετικετών για γράφους συνενώσεων SPARQL. Ως τρέχων παράδειγμα, ας υποθέσουμε ότι θέλουμε να δημιουργήσουμε μια κανονικοποιημένη ετικέτα για το ακόλουθο ερώτημα SPARQL:

?prof teacherOf ?gcourse . (1)
?prof teacherOf ?ugcourse . (2)
?ugcourse type UndergraduateCourse . (3)
?gcourse type GraduateCourse . (4)

Ο πρώτος στόχος είναι να μετατρέψουμε τα ερωτήματα SPARQL σε κατευθυνόμενους με χρωματισμένες κορυφές και ακμές γράφους.

Definition Ένας κατευθυνόμενος με χρωματισμένες κορυφές και ακμές γράφος είναι ένας γράφος G=(V,E,c_v,c_e), όπου το $V = \{1, 2, ..., n\}$ είναι ένα σύνολο κορυφών, το $E \subseteq V \times V$ είναι ένα σύνολο κατευθυνόμενων ακμών, η $c_v : V \rightarrow N$ και η $c_e : E \rightarrow N$ είναι συναρτήσεις χρωματισμού για της κορυφές και για τις ακμές αντίστοιχα.

Ο γράφος συνενώσεων του ερωτήματος φαίνεται στο αριστερό γράφημα του Σχήματος 3.5, όπου τα IDs των κορυφών αντιστοιχούν στα IDs των ερωτημάτων τριάδας που παρουσιάστηκαν παραπάνω. Αρχικά, αφαιρούμε όλες τις πληροφορίες που σχετίζονται με τα ονόματα των μεταβλητών. Για να το κάνουμε αυτό, για κάθε ερώτημα τριάδας δημιουργούμε μια ετικέτα που αποτελείται από τρία IDs, ένα για κάθε θέση της RDF τριάδας. Τα σταθερά στοιχεία μεταφράζονται σε ακέραια αναγνωριστικά IDs χρησιμοποιώντας ένα String-ID λεξικό ενώ οι μεταβλητές μεταφράζονται σε μηδέν. Στο παράδειγμά μας, ας υποθέσουμε ότι το λεξικό String-ID περιέχει: {teacherOf→15, type→3, UndergraduateCourse→52, GraduateCourse→35}. Η ετικέτα του πρώτου ερωτήματος τριάδας είναι "0_15_0". Διαχειριζόμαστε τα *OPTIONAL* ερωτήματα τριάδας χρησιμοποιώντας έναν ειδικό χαρακτήρα '|' στην αρχή της ετικέτας. Επίσης αναθέτουμε κατευθύνσεις και χρώματα στις ακμές με βάση το είδος της συνένωσης που αντιπροσωπεύουν. Όλοι οι πιθανοί τύποι συνενώσεων είναι {SS, SP, SO, PS, PP, PO, OS, OP, OO}. Για να μειώσουμε τον αριθμό των χρωμάτων που απαιτούνται εφαρμόζουμε μια διάταξη θέσεων τριάδας (S<P<O) και προσθέτουμε στον μετασχηματισμένο γράφο μόνο εκείνες τις ακμές των οποίων η αρχική θέση είναι μικρότερη ή ίση με την θέση προορισμού. Έτσι απαιτούνται μόνο οι 6 παρακάτω τύποι ακμών {SS, SP, SO, PP, PO, OO}. Ο δεύτερος γράφος του Σχήματος 3.5 απεικονίζει τις ετικέτες των κορυφών καθώς και των ακμών του γράφου του ερωτήματος.

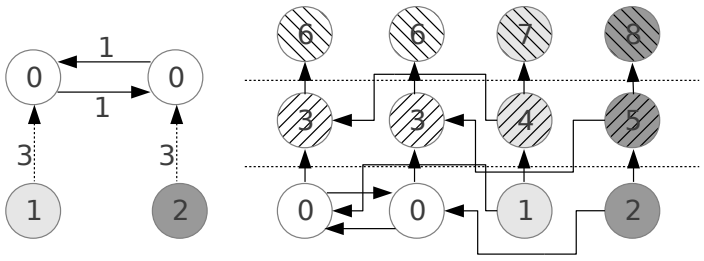


Σχήμα 3.5: *Μετασχηματισμός του SPARQL ερωτήματος*

Το τελευταίο βήμα είναι ο μετασχηματισμός των ετικετών σε θετικά ακέραια χρώματα. Για να το επιτύχουμε, ταξινομούμε τις ετικέτες των κορυφών και χρησιμοποιούμε ως αναγνωριστικό την θέση κάθε ετικέτας στο ταξινομημένο σύνολο. Όσον αφορά τις ακμές χρησιμοποιούμε την παρακάτω μετάφραση {SS→1, SP→2, SO→3, PP→4, PO→5, OO→6}. Ο τελικός γράφος είναι ένας κατευθυνόμενος με χρωματισμένες κορυφές και ακμές γράφος και απεικονίζεται στο Σχήμα 3.6. Οι πληροφορίες ταξινόμησης, ομαδοποίησης, φιλτραρίσματος και προβολής του SPARQL ερωτήματος δεν χρησιμοποιούνται για τη δημιουργία κανονικοποιημένων ετικετών. Ερωτήματα που είναι ισομορφικά αλλά διαφέρουν σε αυτά τα στοιχεία θα πάρουν την ίδια κανονικοποιημένη ετικέτα αλλά θα διαχειριστούν στις επόμενες ενότητες ως διαφορετικές εκδοχές του ίδιου γράφου, Ενότητα 3.4. Ο παραπάνω μετασχηματισμός αφαιρεί όλες τις πληροφορίες που σχετίζονται με τα όνομα των μεταβλητών του ερωτήματος ενώ

καταφέρνει να διατηρήσει όλη τη δομική πληροφορία του ερωτήματος χρησιμοποιώντας τις κατευθυνόμενες ακμές συνενώσεων [Papailiou 15].

Στο [McKay 90] παρουσιάζεται ένας πρακτικός τρόπος μετασχηματισμού ενός κατευθυνόμενου με χρωματισμένες κορυφές και ακμές γράφου σε ένα κατευθυνόμενο με χρωματισμένες κορυφές γράφο, χωρίς να αλλάζουν οι ιδιότητες ισομορφισμού του. Πιο συγκεκριμένα, αν υπάρχουν v χρώματα κορυφών και όλα τα διαθέσιμα χρώματα ακμών είναι ακέραιοι στο πεδίο $\{1, 2, \ldots, 2^d - 1\}$, κατασκευάζουμε ένα γράφο με d επίπεδα. Κάθε ένα από τα επίπεδα περιέχει n κορυφές, όπου n είναι ο αριθμός των κορυφών του αρχικού γράφου. Όπως αναφέρθηκε στην προηγούμενη παράγραφο χρειαζόμαστε μόνο 6 χρώματα ακμών, ένα για κάθε πιθανό τύπο συνένωσης, και ως εκ τούτου το d ισούται με 3 και το μετασχηματισμένο γράφημα περιέχει $3n$ κορυφές. Οι κορυφές των διαφόρων επιπέδων (κάθε μία από τις οποίες αντιστοιχεί σε μία κορυφή του αρχικού γραφήματος) είναι κάθετα συνδεδεμένες χρησιμοποιώντας μονοπάτια. Τα χρώματα των κορυφών του πρώτου επιπέδου παραμένουν ίδια όπως και στον αρχικό γράφο. Τα χρώματα μεταφέρονται και στα υψηλότερα επίπεδα προσθέτοντας v στο αντίστοιχο χρώμα του χαμηλότερου επιπέδου. Για κάθε ακμή του αρχικού γραφήματος η δυαδική αναπαράσταση του αριθμού του χρώματος ορίζει το ποια επίπεδα πρέπει να περιέχουν την οριζόντια άκρη. Για παράδειγμα, μια ακμή με χρώμα 3, του οποίου η δυαδική αναπαράσταση είναι 011, θα τοποθετηθεί τόσο στο πρώτο όσο και το δεύτερο επίπεδο του νέου γράφου. Ο μετασχηματισμός για το παράδειγμά μας απεικονίζεται στο δεύτερο γράφημα του Σχήματος 3.6, όπου τα αναγνωριστικά των κορυφών αντιπροσωπεύουν τα χρώματα που έχουν ανατεθεί.



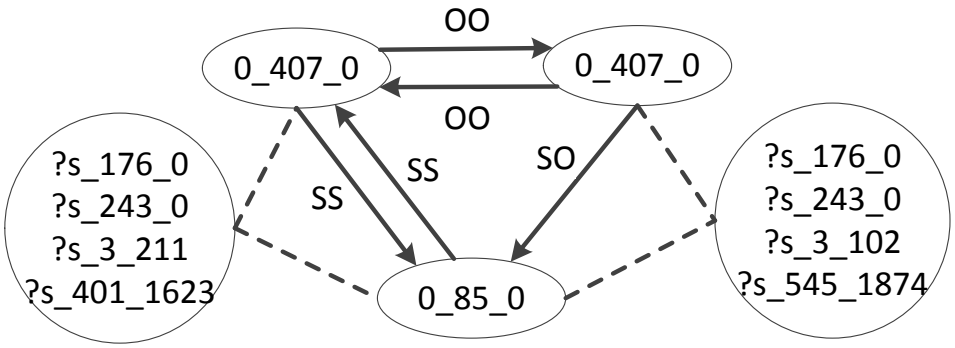
Σχήμα 3.6: *Μετασχηματισμός σε κατευθυνόμενο με χρωματισμένες κορυφές γράφο*

Σε αυτό το σημείο ο αλγόριθμος *Bliss* χρησιμοποιείται για να παραχθεί μια κανονικοποιημένη ετικέτα για το παραπάνω γράφημα. Η διαδικασία επιστρέφει μια κανονική διάταξη των κορυφών του γραφήματος. Όπως αναφέρεται στο [McKay 90], η σειρά με την οποία ο αλγόριθμος παραγωγής κανονικοποιημένων ετικετών επεξεργάζεται τις κορυφές του πρώτου επιπέδου του μετασχηματισμένου γράφου μπορεί να θεωρηθεί ως μια κανονικοποιημένη διάταξη των κορυφών του αρχικού γράφου. Έτσι, μετά την εκτέλεση του *Bliss* μπορούμε να πάρουμε μια κανονικοποιημένη διάταξη των κορυφών του αρχικού μας ερωτήματος. Στο παράδειγμά μας η κανονικοποιημένη διάταξη είναι {1,2,4,3}, όπου τα ids είναι τα αναγνωριστικά

των ερωτημάτων τριάδας του SPARQL ερωτήματος. Χρησιμοποιώντας αυτή τη διάταξη παράγουμε μια αλφαριθμητική ετικέτα για το SPARQL ερώτημα. Για να γίνει αυτό, επεξεργαζόμαστε τα ερωτήματα τριάδας με την συγκεκριμένη σειρά και για κάθε ερώτημα προσθέτουμε την ετικέτα του στο τέλος της τρέχουσας ετικέτας. Ενώ παράγουμε την κανονικοποιημένη ετικέτα παράγουμε επίσης μια κανονικοποιημένη διάταξη των μεταβλητών του ερωτήματος. Για παράδειγμα η μεταβλητή ?prof είναι η πρώτη μεταβλητή που συναντάμε και τις αναθέτουμε το κανονικοποιημένο ID 1. Στο παράδειγμά μας, τα κανονικοποιημένα ονόματα των μεταβλητών είναι {?prof→?1, ?gcourse→?2, ?ugcourse→?3} και η ετικέτα του ερωτήματος ?1_15_?2&?1_15_?3&?2_3_35&?3_3_52. Ο αλγόριθμος που παρουσιάστηκε παράγει την ίδια ακριβώς αλφαριθμητική ετικέτα για κάθε ισομορφικό SPARQL ερώτημα ως εκ τούτου είναι ένας αλγόριθμος παραγωγής κανονικοποιημένων ετικετών για τους γράφους των SPARQL ερωτημάτων. Αυτή η ετικέτα μπορεί να χρησιμοποιηθεί ως κλειδί για την υλοποίηση της κρυφής μνήμης SPARQL αποτελεσμάτων.

3.3.2 Δημιουργία κανονικοποιημένων ετικετών για απλοποιημένους γράφους συνενώσεων SPARQL

Ας υποθέσουμε ότι θέλουμε να δημιουργήσουμε μια κανονικοποιημένη ετικέτα για τον απλοποιημένο γράφο συνενώσεων που παρουσιάστηκε στο Σχήμα 3.4. Όπως πριν θεωρούμε ότι το λεξικό String-ID περιέχει: {author→407, advisor→85, type→3, fullName→176, emailAddress→243, worksFor→401, studiesIn→545, Professor→211, GraduateStudent→102, MIT →1623, Harvard →1874}. Αρχικά μετατρέπουμε τον απλοποιημένο γράφο συνενώσεων του Σχήματος 3.4 σε ένα γράφο με ετικέτες που παρουσιάζεται στο Σχήμα 3.7. Η διαδικασία παραγωγής της κανονικοποιημένης ετικέτας για το σκελετό του ερωτήματος είναι η ίδια με αυτή που παρουσιάστηκε στην προηγούμενη ενότητα. Για κάθε έναν από τους κόμβους αστέρα, δημιουργούμε μια λίστα που περιέχει όλες τις ετικέτες των ερωτημάτων τριάδας του. Η μόνη



Σχήμα 3.7: *Ετικέτες του απλοποιημένου γράφου συνενώσεων για το ερώτημα* Q_e

διαφορά στη διαδικασία παραγωγής ετικετών είναι ότι, σε αυτή την περίπτωση, δεν αντικαθιστούμε τη μεταβλητή συνένωσης του προτύπου αστέρα με '0', αλλά με "?s", προκειμένου να διατηρήσουμε την πληροφορία της θέσης της μεταβλητής αστέρα μέσα στα ερωτήματα τριάδας. Μετά τη δημιουργία της λίστας των ετικετών την ταξινομούμε λεξικογραφικά με σκοπό να δημιουργήσουμε μια μοναδική αλφαριθμητική ετικέτα για το μοτίβο αστέρα. Η δημιουργία κανονικοποιημένων ετικετών για τον απλοποιημένο γράφο συνενώσεων χωρίζεται στα ακόλουθα στάδια:

1. Δημιουργούμε μια κανονικοποιημένη ετικέτα για το γράφημα σκελετού του γράφου συνενώσεων. Αυτό γίνεται χρησιμοποιώντας τον απλό αλγόριθμο επισήμανσης που παρουσιάστηκε στην προηγούμενη ενότητα.

2. Χρησιμοποιούμε την κανονικοποιημένη διάταξη των μεταβλητών που παράγεται από την επισήμανση του σκελετού για να δημιουργήσουμε τις κανονικοποιημένες ετικέτες για τους υπογράφους αστέρων.

3. Συνενώνουμε την αλφαριθμητική ετικέτα του σκελετού με τις επιμέρους ετικέτες των μοτίβων αστέρα, προκειμένου να παραχθεί η κανονικοποιημένη ετικέτα για το σύνολο του ερωτήματος.

Στο παράδειγμά μας η κανονικοποιημένη ετικέτα είναι ?1_407_?2&?3_407_?2& ?3_85_?1 και η κανονικοποιημένη διάταξη των μεταβλητών είναι {?st→?1, ?paper→?2, ?prof→?3}. Η κανονικοποιημένη διάταξη των μεταβλητών του γραφήματος σκελετού παράγει κανονικοποιημένα IDs για όλες τις μεταβλητές συνενώσεων των μοτίβων αστέρα. Στην ακραία περίπτωση ενός ερωτήματος αστέρα, όπου το υπογράφημα σκελετός είναι άδειο η μεταβλητή συνένωσης του αστέρα παίρνει το μικρότερο ID: ?1. Έχοντας δημιουργήσει την κανονικοποιημένη διάταξη όλων των μεταβλητών αστέρα, τη χρησιμοποιούμε για να επεξεργαστούμε τα διάφορα πρότυπα αστέρα. Κάθε υπογράφος αστέρα, έχει ως ετικέτα τη λεξικογραφικά ταξινομημένη λίστα των ετικετών των ερωτημάτων τριάδας του, όπως απεικονίζεται στο Σχήμα 3.7. Στο παράδειγμά μας δημιουργούμε πρώτα την ετικέτα του μοτίβου αστέρα γύρω από τη μεταβλητή ?st→?1 του οποίου η κανονικοποιημένη ετικέτα είναι (?1,[?s_176_0,?s_243_0,?s_3_102,?s_545_1874]). Η ετικέτα του δεύτερου μοτίβου αστέρα γύρω από τη μεταβλητή ?prof→?3 είναι (?3, [?s_176_0,?s_243_0,?s_3_211,?s_401_1623]). Κατά τη δημιουργία των ετικετών αστέρα, επεκτείνουμε επίσης την κανονικοποιημένη διάταξη των μεταβλητών με τις υπόλοιπες μεταβλητές των προτύπων αστέρα {?stName→?4, ?stEmail→?5, ?pName→?6, ?pEmail→?7}. Ωστόσο, αυτές οι τιμές δεν έχουν αντικατασταθεί στην τελική κανονικοποιημένη ετικέτα. Όπως θα συζητήσουμε στην επόμενη ενότητα, αυτό παρέχει τη δυνατότητα οι ετικέτες μας να χρησιμοποιούνται, εκτός από ακριβή έλεγχο ισομορφισμού και για τον έλεγχο ισομορφισμού υπογράφων. Για να δημιουργήσουμε την κανονικοποιημένη ετικέτα του συνόλου του ερωτήματος ενώνουμε την

ετικέτα του σκελετού με τις ετικέτες των μοτίβων αστέρα χρησιμοποιώντας την κανονικοποιημένη διάταξη των μεταβλητών αστέρα. Ως εκ τούτου, η ετικέτα του συνολικού ερωτήματος είναι: ?1_407_?2&?3_407_?2&?3_85_?1&{(?1,[?s_176_0,?s_243_0,?s_3_102,?s_545_1874]), (?3, [?s_176_0,?s_243_0,?s_3_211,?s_401_1623])}.

Proof Ο αλγόριθμος που παρουσιάστηκε είναι ένας αλγόριθμος παραγωγής κανονικοποιημένων ετικετών για το γράφο του SPARQL ερωτήματος. Η διαδικασία κανονικοποίησης του υπογράφου σκελετού είναι ίδια με εκείνη που αναφέρεται στην ενότητα 3.3.1. Ως εκ τούτου, η ετικέτα που δημιουργήθηκε για το σκελετό του ερωτήματος είναι μια κανονικοποιημένη ετικέτα. Όσον αφορά τους υπογράφους αστέρα, αυτοί μπορεί να περιγραφούν από ένα σύνολο ερωτημάτων τριάδας E που συνδέονται μόνο με βάση την κοινή μεταβλητή συνένωσης του αστέρα και είναι επίσης αποσυνδεδεμένα από τον υπόλοιπο γράφο του ερωτήματος. Ο αλγόριθμός μας βασίζει την δημιουργία ετικετών μοτίβων αστέρα σε: i) την αφαίρεση της πληροφορίας των ονομάτων των μεταβλητών από τις ετικέτες του συνόλου E, ii) τη δημιουργία της κανονικοποιημένης ετικέτας του συνόλου των ερωτημάτων με βάση τη λεξικογραφική σειρά τους, iii) την ανάκτηση του κανονικοποιημένου ID για τη μεταβλητή συνένωσης του αστέρα από την ετικέτα του σκελετού του γραφήματος και την ανάθεση κανονικοποιημένων ID στις υπόλοιπες μεταβλητές με βάση αυτό.

Θα αποδείξουμε ότι η λεξικογραφική διάταξη των ετικετών του συνόλου των ερωτημάτων τριάδας E παρέχει μια κανονικοποιημένη ετικέτα για ένα γράφημα αστέρα. Αρχικά, όλα τα ισομορφικά ερωτήματα αστέρα παράγουν το ίδιο σύνολο E, ανεξάρτητα από τα ονόματα των μεταβλητών που χρησιμοποιούνται, καθώς και τη διάταξη των ερωτημάτων τριάδας. Αντίστοιχα, το σύνολο των ερωτημάτων τριάδας E μπορεί να χρησιμοποιηθεί για να δημιουργήσει ένα ισομορφικό γράφημα αστέρα. Αυτό μπορεί να γίνει δίνοντας μοναδικά αναγνωριστικά σε όλες τις μεταβλητές που παρουσιάζονται ως "0" στις ετικέτες των ερωτημάτων τριάδας. Η μεταβλητή συνένωσης του αστέρα παρουσιάζεται ως "?s" μπορεί επίσης να ανατεθεί σε ένα μοναδικό ID. Ως εκ τούτου, η δημιουργία μιας κανονικοποιημένης ετικέτας για το σύνολο E παρέχει μια κανονικοποιημένη ετικέτα για το γράφημα αστέρα. Λόγω του γεγονότος ότι δεν υπάρχουν εξαρτήσεις μεταξύ των ετικετών που περιέχονται στο E, η λεξικογραφική διάταξη μπορεί να χρησιμοποιηθεί ως μια κανονικοποιημένη διάταξη. Έτσι, η λεξικογραφική διάταξη του συνόλου E παρέχει μια κανονικοποιημένη ετικέτα για ένα γράφημα αστέρα.

Επιπλέον, συνενώνοντας την ετικέτα του σκελετού με τις ετικέτες των μοτίβων αστέρα χρησιμοποιώντας την κανονικοποιημένη διάταξη των μεταβλητών αστέρα παρέχει μια κανονικοποιημένη ετικέτα για το σύνολο του ερωτήματος. Για να αποδειχθεί αυτό, οφείλουμε να παρατηρήσουμε ότι δυο ισομορφικά ερωτήματα έχουν τον ίδιο σκελετό και έτσι παράγουν την ίδια ακριβώς κανονικοποιημένη διάταξη για τις μεταβλητές συνενώσεων των μοτίβων αστέρα. Ως εκ τούτου, για όλα τα ισομορφικά ερωτήματα, οι ετικέτες αστέρα θα συνενωθούν με την

ίδια σειρά και έτσι οι παραγόμενες συνολικές ετικέτες είναι κανονικοποιημένες ετικέτες για ολόκληρο το ερώτημα και μπορούμε να τις χρησιμοποιήσουμε για τη διενέργεια δοκιμών ισομορφισμού.

3.3.3 Χρησιμοποίηση των κανονικοποιημένων ετικετών για εξέταση ισομορφισμού υπογράφων

Χρησιμοποιώντας τον προηγούμενο αλγόριθμο για τη δημιουργία κανονικοποιημένων ετικετών μας παρέχεται επίσης η δυνατότητα να εκτελούμε εξέταση για ισομορφισμό υπογράφων σε συγκεκριμένες κατηγορίες γράφων. Πιο συγκεκριμένα, το πρόβλημα του ισομορφισμού υπογράφων ορίζεται ως:

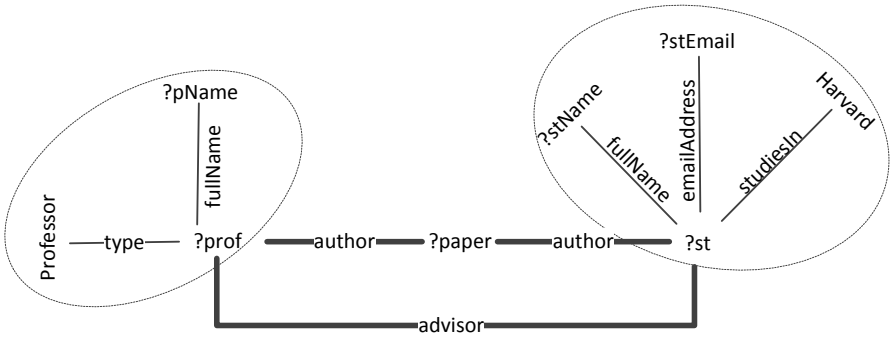
Definition Δεδομένου δυο γράφων $G(V, E)$ και $H(V', E')$ θέλουμε να ελέγξουμε αν υπάρχει κάποιος υπογράφος $G_1(V_1, E_1) : V_1 \subseteq V, E_1 \subseteq E$, του G που είναι ισομορφικός με τον H.

Παρόλο που οι κανονικοποιημένες ετικέτες έχουν σχεδιαστεί για το ισομορφισμό γράφων, οι προτεινόμενες ετικέτες σκελετού-αστέρα μπορούν να χρησιμοποιηθούν αποδοτικά για την εξέταση ισομορφισμού υπογράφων για γράφους που έχουν τον ίδιο σκελετό αλλά διαφέρουν στα μοτίβα αστέρων τους. Πιο συγκεκριμένα, ορίζουμε το πρόβλημα εξέτασης ισομορφισμού υπογράφων σκελετού-αστέρα.

Definition Δεδομένου δυο γράφων $G(V, E)$, $H(V', E')$ και των αντίστοιχων απλοποιημένων γράφων (Sk, St), (Sk', St'), όπου $Sk \subseteq (V, E)$, $Sk' \subseteq (V', E')$ και $St = \langle s, L(s) \subseteq V \rangle$, $St' = \langle s, L'(s) \subseteq V' \rangle$ είναι οι υπογράφοι σκελετού και αστέρων των G και H αντίστοιχα. Ο H είναι ισομορφικός υπογράφος σκελετού-αστέρα του G ανν υπάρχει ένας υπογράφος, $G_1(V_1, E_1) : V_1 \subseteq V, E_1 \subseteq E$, του G με $Sk_1 = Sk$, ο οποίος είναι ισομορφικός του H. Επιπλέον, αν ο H είναι ισομορφικός υπογράφος σκελετού-αστέρα του G, τότε είναι αληθές ότι οι υπογράφοι σκελετού τους είναι ισομορφικοί $Sk = Sk_1 = Sk'$. Όσον αφορά τους υπογράφους αστέρα τους, ισχύει ότι $St' \subseteq St : \{\forall (s, L'(s)) \in St' : (s, L(s)) \in St, L'(s) \subseteq L(s)\}$

Για να είναι ισομορφικός υπογράφος σκελετού-αστέρα, ένα ερώτημα πρέπει να έχει ισομορφικό γράφο σκελετού με ένα άλλο και άρα τις ίδιες κανονικοποιημένες ετικέτες σκελετού. Επιπλέον, όλα τα μοτίβα αστέρων του ενός θα πρέπει να είναι υπογράφοι των μοτίβων αστέρα του άλλου ερωτήματος. Για παράδειγμα, το ερώτημα του Σχήματος 3.8 είναι ισομορφικός υπογράφος σκελετού-αστέρα του ερωτήματος που απεικονίζεται στο Σχήμα 3.3.

Ο αλγόριθμος που χρησιμοποιεί τις κανονικοποιημένες ετικέτες για την εξέταση ισομορφισμού υπογράφου σκελετού-αστέρα παρουσιάζεται στον Αλγόριθμο 4. Ο αλγόριθμος ξεκινά με τον έλεγχο της ισότητας των κανονικοποιημένων ετικετών σκελετού. Αν οι υπογράφοι σκελετού των εξετασθέντων ερωτημάτων είναι ισομορφικοί, συνεχίζουμε τη δοκιμή υπογράφων



Σχήμα 3.8: *Ερώτημα που είναι ισομορφικός υπογράφος σκελετού-αστέρα του Q_e*

Algorithm 4: Εξέταση ισομορφισμού υπογράφου σκελετού-αστέρα

Input: Δυο γράφοι G, H και $(G_{sk}, G_{st}), (H_{sk}, H_{st})$,οι κανονικοποιημένες ετικέτες σκελετού και αστέρων τους

Output: True/false αν ο H είναι ισομορφικός υπογράφος σκελετού-αστέρα του G

if $G_{sk} \neq H_{sk}$ **then**
  return false;
else
  $sit \leftarrow H_{st}.iterator()$; $hst = sit.nextStar()$; $starMatches = false$;
  for $gst \in G_{st}$ **do**
    if $gst.getStarVar() = hst.getStarVar()$ **then**
      $pit = hst.iterator()$; $hp = pit.nextPattern()$; $patternMatches = false$;
      for $gp \in gst$ **do**
        if $gp = hp$ **then**
          $patternMatches = true$;
          if $pit.hasMore()$ **then**
            $hp = pit.nextPattern()$;
          else
            break;
          end
        end
      end
      if $pit.hasMore()||patternMatches = false$ **then**
        return false;
      end
      $starMatches = true$;
    end
  end
  if $sit.hasMore()||starMatches = false$ **then**
    return false;
  end
  return true;
end

αστέρα, αλλιώς επιστρέφουμε *false* γιατί τα δύο ερωτήματα, εξ ορισμού, δεν είναι ισομορφικά υπογραφήματα σκελετού-αστέρα. Για να ελέγξουμε τον ισομορφισμό των μοτίβων αστέρα πρέπει να διασφαλίσουμε ότι *όλα* τα ερωτήματα τριάδας των υπογράφων αστέρα του H περιέχονται στον G. Για να γίνει αυτό αποτελεσματικά, λαμβάνουμε υπόψη την κανονικοποιημένη διάταξη των μεταβλητών συνένωσης των αστέρων, καθώς και τη λεξικογραφική σειρά των ετικετών τριάδας στο εσωτερικό των ετικετών αστέρων.

Λόγω του γεγονότος ότι οι υπογράφοι σκελετού των δύο ερωτήματων είναι ισομορφικοί, ο αλγόριθμος κανονικοποιημένης επισήμανσης θα δημιουργήσει την ίδια κανονικοποιημένη διάταξη των μεταβλητών για όλες τις μεταβλητές συνένωσης αστέρα. Η μόνη διαφορά είναι ότι ο G μπορεί να έχει περισσότερα μοτίβα αστέρα από τον H. Για να βρούμε αν *ολα* τα ερωτήματα τριάδας αστέρα του H υπάρχουν στον G απλά εξετάζουμε με χρήση της κανονικοποιημένης διάταξης τους υπογράφους αστέρα του H και προσπαθούμε να τους ταιριάξουμε με τους υπογράφους αστέρα του G. Για να ελέγξουμε αν δύο υπογράφοι αστέρα ταιριάζουν, πρέπει πρώτα να ελέγξουμε αν έχουν τα ίδια ID στην μεταβλητή συνένωσής τους. Αν αυτό ισχύει, συνεχίζουμε τον έλεγχο με την εξέταση των επιμέρους λεξικογραφικά ταξινομημένων ερωτημάτων τριάδας. Και πάλι, το σύνολο των ερωτημάτων τριάδας για ένα υπογράφημα αστέρα του G επιτρέπεται να έχει περισσότερα ερωτήματα από το αντίστοιχο του H, αλλά *όλα* τα ερωτήματα του H πρέπει να είναι παρόντα στα αντίστοιχα μοτίβα αστέρα του G. Η χρήση των γενικών αναγνωριστικών "0" και "?s" στις ετικέτες αστέρων, εξασφαλίζει ότι η λεξικογραφική σειρά των ερωτημάτων τριάδας των αστέρων θα είναι η ίδια και στα δύο ερωτήματα. Ως εκ τούτου, ο προτεινόμενος αλγόριθμος μπορεί να εκτελέσει τη δοκιμή ισομορφισμού υπογράφου σκελετού-αστέρα με ένα μόνο πέρασμα πάνω από τα κανονικοποιημένες ετικέτες των δοκιμαζόμενων ερωτημάτων.

3.3.4 Πολυπλοκότητα δημιουργίας κανονικοποιημένων ετικετών

Σε αυτή την ενότητα, θα εξετάσουμε την πολυπλοκότητα των προτεινόμενων αλγορίθμων κανονικοποιημένης επισήμανσης. Ο πρώτος αλγόριθμος που εξετάσαμε είναι η δημιουργία κανονικοποιημένων ετικετών της ενότητας 3.3.1. Η πολυπλοκότητα αυτού του αλγορίθμου συνδέεται άμεσα με την πολυπλοκότητα του αλγορίθμου *Bliss* που προσπαθεί να λύσει το πρόβλημα ισομορφισμού γράφων (GI). Το πρόβλημα αυτό είναι γνωστό ότι έχει στην χειρότερη περιπτωση χρονική πολυπλοκότητα $O(2^{\sqrt{n \log n}})$ σε γράφους με n κορυφές [Arvind 00]. Ωστόσο, αυτή η πολυπλοκότητα δεν είναι αντιπροσωπευτική για το *Bliss* επειδή η πολυπλοκότητά του εξαρτάται κυρίως από την ποσότητα των αυτομορφισμών που είναι παρόντες στην δομή του γράφου και όχι τόσο από τον αριθμό των κορυφών του. Πράγματι, υπάρχουν παραδείγματα γράφων που παρουσιάζουν εκθετική πολυπλοκότητα αλλά στη γενική περίπτωση ο

Bliss παρουσιάζει πολυωνυμική πολυπλοκότητα. Ο αλγόριθμος επισήμανσης SPARQL ερωτημάτων εισάγει ένα πολυωνυμικό χρόνο, $O(n)$, μετατροπής του ερωτήματος εισόδου, ο οποίος είναι αμελητέος σε σύγκριση με τον εκθετικό χρόνο που απαιτείται στην χειρότερη περίπτωση από το *Bliss*. Η μεγαλύτερη επιβάρυνση του αλγορίθμου μας είναι το γεγονός ότι μετατρέπει τις n κορυφές του αρχικού γράφου σε $3n$ κορυφές και έτσι εισάγει μια πολυωνυμική αύξηση στο μεγέθους εισόδου.

Σχετικά με τον αλγόριθμο κανονικοποιημένης επισήμανσης των απλοποιημένων γράφων, που παρουσιάστηκε στην ενότητα 3.3.2, αυτός εξαρτάται και πάλι από την πολυπλοκότητα του *Bliss*. Η κύρια διαφορά τώρα είναι ότι ο *Bliss* τρέχει μόνο για το υπογράφημα σκελετού του ερωτήματος. Η πολυπλοκότητα της επισήμανσης του σκελετού είναι η ίδια με αυτή που συζητήθηκε παραπάνω, αλλά τώρα εξαρτάται από τον αριθμό των ερωτημάτων τριάδας του σκελετού sk, ο οποίος μπορεί να είναι πολύ μικρότερος από n. Επιπλέον, η διαδικασία απλοποίησης αφαιρεί όλους τους υπογράφους αστέρα από το ερώτημα και έτσι αφαιρεί κλίκες από το γράφημα συνενώσεων. Ως εκ τούτου, το γράφημα σκελετός περιέχει όχι μόνο λιγότερο κορυφές από το αρχικό γράφημα συνενώσεων, αλλά είναι επίσης πιο χαλαρά συνδεδεμένο. Ως αποτέλεσμα, το γράφημα σκελετός περιέχει λιγότερους αυτομορφισμούς από το αρχικό και έτσι μειώνει την πολυπλοκότητα που απαιτείται από τον *Bliss* αλγόριθμο. Η δημιουργία κανονικοποιημένων ετικετών για τα μοτίβα αστέρων έχει πολυωνυμική πολυπλοκότητα $O(st \log(st))$, με βάση τον αριθμό των ερωτημάτων τριάδας αστέρα st, επειδή απαιτεί μόνο μια λεξικογραφική ταξινόμηση των ετικετών τους.

Τέλος, αφού δημιουργούμε κανονικοποιημένες ετικέτες για γραφήματα SPARQL ερωτημάτων, μπορούμε να εκτελέσουμε την εξέταση ισομορφισμού υπογράφου σκελετού-αστέρα σε πολυωνυμικό χρόνο. Η πολυπλοκότητα του προτεινόμενου αλγορίθμου είναι γραμμική σε σχέση με τον αριθμό των ερωτημάτων τριάδας του ερωτήματος n, γιατί στη χειρότερη περίπτωση χρειάζεται να εξετάσει μία φορά την ετικέτα κάθε ερωτήματος. Η ακριβής δοκιμή ισομορφισμού έχει επίσης γραμμική πολυπλοκότητα στο μέγεθος της ετικέτας n διότι πρέπει να εκτελέσουμε μια εξέταση ισότητας μεταξύ των δύο ετικετών.

3.4 Σχεδιασμός εκτέλεσης ερωτημάτων

Η εύρεση του βέλτιστου πλάνου εκτέλεσης συνενώσεων για σύνθετα ερωτήματα ήταν πάντα μια μεγάλη ερευνητική πρόκληση για τη βελτιστοποίηση συστημάτων βάσεων δεδομένων. Επιπλέον, το σύστημά μας θα πρέπει να εξετάσει αποτελεσματικά ποια από τα διαθέσιμα προσωρινά αποθηκευμένα αποτελέσματα μπορούν να χρησιμοποιηθούν για την παροχή αποτελεσμάτων για ένα υπογράφο του ερωτήματος. Και τα δύο αυτά προβλήματα έχουν εκθετική πολυπλοκότητα σε σχέση με το μέγεθος του ερωτήματος, διότι πρέπει να ελέγξουν το σύνολο

των υπογράφων του. Ενώ υπάρχουν πολλοί άπληστοι, ευριστικοί αλγόριθμοι για το σχεδιασμό του πλάνου εκτέλεσης ενός SPARQL ερωτήματος [Tsialiamanis 12, Papailiou 13], τέτοιοι αλγόριθμοι δεν μπορούν εύκολα να συνδυαστούν με έναν αλγόριθμο που εξετάζει τη χρήση προσωρινά αποθηκευμένων αποτελεσμάτων. Σε αντίθεση, οι αλγόριθμοι δυναμικού προγραμματισμού [Moerkotte 06, Neumann 10a] εξετάζουν όλους τους υπογράφους ενός ερωτήματος και έτσι μπορούν εύκολα να τροποποιηθούν για την εύρεση τόσο του βέλτιστο πλάνου εκτέλεσης όσο και του πλάνου χρησιμοποίησης των αποθηκευμένων αποτελεσμάτων.

Ένας από τους παλαιότερους και πιο αποδοτικούς αλγορίθμους δυναμικού προγραμματισμού για τον σχεδιασμό του πλάνου εκτέλεσης ερωτημάτων είναι ο *DPsize* [Gassner 93], που χρησιμοποιείται ευρέως σε εμπορικές βάσεις δεδομένων, όπως η DB2 της IBM. Ο *DPsize* περιορίζει το χώρο αναζήτησης σε left-deep δέντρα και παράγει πλάνα εκτέλεσης σε αύξουσα σειρά μεγέθους. Μια πιο πρόσφατη προσέγγιση, ο *DPccp* [Moerkotte 06] καθώς και η παραλλαγή του ο *DPhyp* [Moerkotte 08] θεωρούνται ως οι πιο αποτελεσματικοί αλγόριθμοι δυναμικού προγραμματισμού για τη βελτιστοποίηση ερωτημάτων. Μειώνουν το χώρο αναζήτησης εξετάζοντας μόνο τους συνδεδεμένους υπογράφους του ερωτήματος ακολουθώντας μια bottom-up διάταξη. Επιπλέον, ο *DPccp* έχει χρησιμοποιηθεί επιτυχώς για τη βελτιστοποίηση SPARQL ερωτημάτων στο RDF-3X [Neumann 10a].

Επιπλέον, τα ερωτήματα SPARQL περιέχουν πολλά μοτίβα αστέρα τα οποία καθιστούν τις τεχνικές multi-way συνενώσεων ελκυστικές για τις διάφορες μηχανές εκτέλεσης SPARQL ερωτημάτων. Μια multi-way συνένωση μπορεί να επεξεργαστεί ένα ερώτημα αστέρα, με αυθαίρετο αριθμό ερωτημάτων τριάδας, σε ένα βήμα. Ειδικά στην περίπτωση της ευρετηρίασης όλων των αναδιατάξεων των RDF τριάδων, ένα ερώτημα αστέρα μπορεί να εκτελεστεί αποτελεσματικά με μια multi-way merge συνένωση αντί για μια ακολουθία 2-way merge συνενώσεων [Papailiou 13]. Ο *DPccp* διερευνά μόνο 2-way πλάνα συνενώσεων οπότε τον επεκτείνουμε προσθέτοντας υποστήριξη για εξερεύνηση mutli-way πλάνων συνενώσεων καθώς και εξέτασης της χρησιμοποίησης των προσωρινά αποθηκευμένων αποτελεσμάτων της κρυφής μνήμης. Ωστόσο, το πλαίσιο κρυφής μνήμης που παρουσιάζουμε μπορεί επίσης να ενσωματωθεί σε μηχανές εκτέλεσης που δεν υποστηρίζει multi-way συνενώσει χρησιμοποιώντας τον αρχικό *DPccp* αλγόριθμο.

3.4.1 Εξερεύνηση Multi-way πλάνων συνένωσης

Στην ενότητα αυτή, περιγράφουμε τις αλλαγές που απαιτούνται ώστε ο *DPccp* να εξερευνήσει αποτελεσματικά multi-way πλάνα συνενώσεων. Η αναπαράσταση των ερωτημάτων εισόδου $G = (V, E, c)$ που χρησιμοποιούμε για τον αλγόριθμο δυναμικού προγραμματισμού μας είναι ο απλοποιημένος γράφος συνενώσεων του ερωτήματος, που παρουσιάστηκε στο Σχήμα

3.3 της ενότητας 3.2. Κάθε ερώτημα τριάδας αντιπροσωπεύεται από μια κορυφή και τα ερωτήματα που μοιράζονται μια κοινή μεταβλητή συνδέονται με μια ακμή με ετικέτα το όνομα της μεταβλητής. Ο *DPccp* βασίζει τη διαδικασία απαρίθμησής του στην εξεύρεση όλων των *csg-cmp-pairs* του γράφου του ερωτήματος [Moerkotte 08]. Τα *csg-cmp-pairs* είναι ζεύγη που περιέχουν ένα συνδεδεμένο υπογράφο (csg) του ερωτήματος και έναν συνδεδεμένο συμπληρωματικό υπογράφο (cmp).

Definition (csg-cmp-pair). Έστω $G = (V, E, c)$ ένα γράφος ερωτήματος και S1, S2 δυο υποσύνολα του V ώστε τα S1 $\subseteq$ V και S2 $\subseteq$ (V $\setminus$ S1) να ορίζουν έναν συνδεδεμένο υπογράφο και ένα συνδεδεμένο συμπληρωματικό υπογράφο του G αντίστοιχα. Αν υπάρχει μια ακμή (u, v)$\in$E ώστε το u$\in$S1 και το v$\in$S2, καλούμε τα (S1, S2) ένα *csg-cmp-pairs*.

Η απαρίθμηση των *csg-cmp-pairs* περιορίζει τον *DPccp* σε 2-way πλάνα συνενώσεων αφού κάθε *csg-cmp-pair* αντιστοιχεί σε μια 2-way συνένωση μεταξύ του csg και του cmp γράφου. Για να εξερευνήσει multi-way πλάνα συνενώσεων ο αλγόριθμός μας απαριθμεί label connected, connected complement subgraph lists(*cmp-lc-list*).

Definition (cmp-lc-list). Έστω $G = (V, E, c)$ ένα γράφος ερωτήματος, L μια ετικέτα ακμής και S={S_1,. . . ,S_n} μια λίστα από σύνολα όπου τα $S_1 \subseteq$V, $S_2 \subseteq$(V $\setminus$ S_1), ..., $S_n \subseteq$ (V $\setminus$ $S_1, \ldots, S_{n-1}$) είναι συνδεδεμένοι υπογράφοι του G. Αν για κάθε ζευγάρι (S_i, S_j) $\in$ S υπάρχει μια ακμή (u, v) $\in$ E με ετικέτα L έτσι ώστε u $\in$ S_i και v $\in$ S_j, ονομάζουμε τη λίστα S cmp-lc-list.

Κάθε *cmp-lc-list* αντιστοιχεί σε ένα multi-way join όλων των υπογράφων που ανήκουν στη λίστα με βάση την κοινή μεταβλητή που αντιστοιχεί στην ετικέτα L. Παρατηρήστε ότι αν το S={S_1, . . . , S_n} είναι μια *cmp-lc-list*, τότε κάθε αναδιάταξή του είναι επίσης μια *cmp-lc-list*. Περιορίζουμε την απαρίθμηση των *cmp-lc-list* επεκτείνοντας τον τρόπο που παρουσιάστηκε στο [Moerkotte 08]. Απαριθμούμε μόνο τις λίστες που ικανοποιούν την ιδιότητα min(S_1) $\prec$ min(S_2) $\prec \cdots \prec$ min(S_n), όπου το $\prec$ είναι μια ολική διάταξη των κορυφών του G που ορίζεται από IDs τους και

$$min(S_i) = \{v | v \in S_i, \forall v' \in S_i : v' \neq v \implies v \prec v'\} \tag{3.1}$$

Υποθέτοντας ότι οι multi-way συνενώσεις είναι αντιμεταθετικές, δηλαδή ότι η σειρά των συνόλων της λίστας δεν επηρεάζει το κόστος της συνένωσης, η απαρίθμηση όλων των αναδιατάξεων θα ήταν άσκοπη. Άρα η εφαρμογή του παραπάνω περιορισμού εγγυάται ότι δεν θα απαριθμηθούν πολλαπλές φορές οι ίδιες *cmp-lc-lists* από τον αλγόριθμο δυναμικού προγραμματισμού μας. Επιπλέον, για να βρεί το βέλτιστο πλάνο εκτέλεσης, ένας αλγόριθμος θα πρέπει να εξετάσει όλα τα *cmp-lc-lists*, κάθε ένα από τα οποία αντιστοιχεί σε μια διαφορετική multi-way συνένωση, καθιστώντας τον αλγόριθμό μας βέλτιστο με βάση τον αριθμό των εξεταζόμενων πλάνων εκτέλεσης.

3.4.2 Αλγόριθμος δυναμικού προγραμματισμού για εύρεση βέλτιστου πλάνου εκτέλεσης

Σε αυτή την ενότητα, παρουσιάζουμε τον ψευδοκώδικα του αλγορίθμου δυναμικού προγραμματισμού μας. Επικεντρωνόμαστε κυρίως στις αλλαγές που έγιναν σε σχέση με τον *DPccp* αλγόριθμο και διατηρούμε τους ίδιους συμβολισμούς που χρησιμοποιούνται στο [Moerkotte 08], επιτρέποντας στους ενδιαφερόμενους αναγνώστες να συγκρίνουν τις λεπτομέρειες των αλγορίθμων. Ο προτεινόμενος αλγόριθμος δυναμικού προγραμματισμού τρέχει πάνω από τον απλοποιημένο γράφο συνενώσεων που περιγράφεται στην ενότητα 3.2. Η κύρια μέθοδός του είναι η *solve*(V), όπου V είναι το σύνολο των κόμβων του απλοποιημένου γράφου συνενώσεων και παρουσιάζεται στον Αλγόριθμο 5. Η μέθοδος *mergeAll* συγχωνεύει δύο πλάνα συνενώσεων που αναφέρονται στον ίδιο υπογράφο διατηρώντας μόνο το καλύτερο πλάνο για κάθε ξεχωριστή ταξινόμηση του αποτελέσματος.

Algorithm 5: $solve(V)$

```
1  for v ∈ V do
2      //αρχικοποίηση του dpTable
3      if v.isStar() then
4          dpTable[{v}].mergeAll(getPlanForStar(v));
5      else
6          dpTable[{v}].mergeAll(indexScans(v));
7      end
8  end
9  for v ∈ V descending according to ≺ do
10     emitCsg({v});
11     enumerateCsgRec({v}, B_v);
12 end
13 return dpTable[V];
```

Αρχικά, η *solve*(V) αρχικοποιεί τον dpTable με τα πλάνα εκτέλεσης όλων των κορυφών του απλοποιημένου γράφου συνενώσεων. Το απλοποιημένο διάγραμμα περιέχει κορυφές που αντιπροσωπεύουν είτε ερωτήματα τριάδας σκελετού ή υπογράφους αστέρων. Στην περίπτωση των ερωτημάτων σκελετού, προσθέτουμε στο dpTable όλες τις πιθανές σαρώσεις ευρετηρίων που μπορούν να μας δώσουν τα αποτελέσματα του ερωτήματος. Για παράδειγμα, εάν χρησιμοποιείται hexastore ευρετηρίαση, τα δεδομένα ενός ερωτήματος τριάδας μπορεί να ανακτηθούν από πολλαπλά ευρετήρια με διαφορετικές διατάξεις. Όσον αφορά τους υπογράφους αστέρα, θα πρέπει να δημιουργήσουμε τα πλάνα εκτέλεσής τους και να τα αποθηκεύσουμε στο dpTable. Έχουμε ενσωματώσει το πλαίσιο κρυφής μνήμης μας με τη H_2RDF+ βάση δεδομένων RDF, η οποία όπως προαναφέρθηκε χρησιμοποιεί αποδοτικές multi-way συνενώσεις για την εκτέλεση ερωτημάτων αστέρα. Αυτό σημαίνει ότι το καλύτερο πλάνο εκτέλεσης για ένα υπογράφημα αστέρα αποτελείται από μια multi-way συνένωση με βάση την κοινή μεταβλητή. Έτσι

μπορούμε εύκολα να δημιουργήσουμε τα αρχικά πλάνα εκτέλεσης για όλες τις κορυφές του απλοποιημένου γράφου συνενώσεων. Στην περίπτωση συστημάτων που χρησιμοποιούν 2-way συνενώσεις, υπάρχουν αποτελεσματικές μέθοδοι παραγωγής καλών πλάνων εκτέλεσης για τα μοτίβα αστέρα, όπως η hierarchical characterisation και τα characteristic sets [Gubichev 14].

Στη συνέχεια, η *solve* μέθοδος καλεί δυο υπορουτίνες τις *emitCsg* και *enumerateCsgRec* για όλους τους κόμβους σε φθίνουσα σειρά με βάση το $\prec$. Για καλύτερη αναγνωσιμότητα, διατηρούμε τα ονόματα που χρησιμοποιούνταν στο [Moerkotte 08] αλλά για μας το *csg* σύνολο αντιστοιχεί στο πρώτο σύνολο μιας *cmp-lc-list*. Άρα η *emitCsg* μέθοδος βρίσκει το πρώτο σύνολο της *cmp-lc-list* και συνεχίζει με την απαρίθμηση των υπόλοιπων υποσυνόλων. Η συνάρτηση *enumerateCsgRec(S_1, X)* χρησιμοποιείται για να επεκτείνει ένα συνδεδεμένο υπογράφο S_1 σε ένα μεγαλύτερο συνδεδεμένο υπογράφο αποφεύγοντας κόμβους που ανήκουν στο σύνολο αποκλεισμού X. Το σύνολο αποκλεισμού χρησιμοποιείται για την αποφυγή της διπλής απαρίθμησης υπογράφων αφού αποτρέπει την επέκταση των υπογράφων με κόμβους που έχουν διάταξη μικρότερη του v σύμφωνα με το $\prec$. Αυτό επιτυγχάνεται με τη χρήση το $\mathcal{B}_v = \{w : w \prec v\} \cup \{v\}$ που αποτρέπει κόμβους με ID μικρότερο του v.

Algorithm 6: $enumerateCsgRec(S_1, X)$

```
1 for N ⊆ 𝒩(S1, X) : N ≠ ∅ do
2     emitCsg(S1 ∪ N);
3 end
4 for N ⊆ 𝒩(S1, X) : N ≠ ∅ do
5     enumerateCsgRec(S1 ∪ N, X ∪ 𝒩(S1, X));
6 end
```

Για να επεκτείνουμε έναν υπογράφο, χρησιμοποιούμε την *enumerateCsgRec* η οποία βασίζεται στη γειτονία του υπογράφου $\mathcal{N}(S_1, X)$ που περιέχει όλους τους κόμβους που είναι προσβάσιμοι από το S_1 με μια ακμή και δεν ανήκουν στο X. Για όλους αυτούς τους κόμβους καλούμε τις *emitCsg* και *enumerateCsgRec*.

Για να διευκολύνουμε την κατανόηση του αλγορίθμου, το Σχήμα 3.9 απεικονίζει την εκτέλεσή του για τον απλοποιημένο γράφο της ενότητας 3.2. Το επάνω μέρος της εικόνας απεικονίζει τη σειρά των κλήσεων των συναρτήσεων *emitCsg* και *emitCmpLcList*. Οι κλήσεις της συνάρτησης *emitCsg(S_1)* καταγράφονται ως $\{S_1\}$ ενώ για την *emitCmpLcList* χρησιμοποιούμε το $[var|\{S_1\}, \ldots, \{S_n\}]$. Η *solve* μέθοδος αρχίζει με το σύνολο {12} επειδή αυτός είναι ο μεγαλύτερος κόμβος σύμφωνα με την $\prec$. Αυτό το σύνολο δε θα δημιουργήσει καμία *cmp-lc-list* και δεν θα επεκταθεί περαιτέρω αφού το σύνολο αποκλεισμού του περιέχει όλους τους γείτονές του. Το ίδιο θα συμβεί και για το σύνολο {11}. Η μέθοδος *emitCsg* στη συνέχεια θα απαριθμήσει όλες τις *cmp-lc-lists* που έχουν το σύνολο {6} ως πρώτο σύνολο. Όταν αυτή η διαδικασία τελειώσει η *enumerateCsgRec* θα επεκτείνει το {6} καλώντας την *emitCsg* για το {6,12}. Το ίδιο

θα συμβεί για το σύνολο {5} που θα επεκταθεί σε {5, 6}, {5, 11}, {5, 12}, {5, 6, 11}, {5, 6, 12}, {5, 11, 12}, {5, 6, 11, 12}. Το σύνολο {4} θα επεκταθεί σε {4, 5}, {4, 6}, {4, 11}, {4, 5, 6}, {4, 5, 11}, {4, 6, 11}, {4, 5, 6, 11} από τον πρώτο βρόχο της *enumerateCsgRec* μεθόδου. Ο δεύτερος βρόχος θα προσθέσει αναδρομικά τον κόμβο 12 σε όλα τα παραπάνω σύνολα καλώντας την *emitCsg* για τα {4, 5, 12}, {4, 6, 12}, {4, 5, 6, 12}, {4, 5, 11, 12}, {4, 6, 11, 12} και {4, 5, 6, 11, 12}.

Algorithm 7: $emitCsg(S_1)$

```
1  if |S_1| ≥ 2 then
2      dpTable[S_1].mergeAll(checkCache(S_1)); //check cache and merge plans
3  end
4  X = S_1 ∪ B_{min(S_1)};
5  N_label = N_label(S_1, X);
6  for (l, S_2) ∈ N_label : S_2 ≠ ∅ do
7      X_1 = X ∪ S_2;
8      list.push(S_1);
9      enumerateCmpLcList(list, S_2, X_1, l);
10 end
```

Κατά τη διάρκεια της διαδικασίας απαρίθμησης, η συνάρτηση *emitCsg(S_1)* εκτελείται μόνο μια φορά για κάθε συνδεδεμένο υπογράφο του ερωτήματος, αναδεικνύοντάς την ως ένα καλό μέρος για την προσθήκη του μηχανισμού ελέγχου της κρυφής μνήμης. Επιπλέον, η *emitCsg(S_1)* καλείται πριν από οποιαδήποτε προσπάθεια για χρησιμοποίηση του S_1 σε μια συνένωση και άρα η χρήση των αποτελεσμάτων της κρυφής μνήμης που θα ανακτηθούν εδώ θα εξεταστεί για όλα τα πλάνα που περιέχουν το S_1.

Η συνάρτηση *emitCsg* συνεχίζει με την απαρίθμηση των *cmp-lc-lists* που παρουσιάστηκαν στην ενότητα 3.4.1. Εδώ εισάγουμε την έννοια της labelled γειτονιάς ενός υπογράφου $N_{label} = \mathcal{N}_{label}(S_1, X)$. Το N_{label} είναι ένα σύνολο από ζευγάρια (l, S_2), όπου το l είναι μια ετικέτα ακμής που αντιστοιχεί σε μια μεταβλητή συνένωσης και το S_2 είναι το σύνολο των κόμβων που δεν περιέχονται στο X και είναι προσβάσιμοι από το S_1 με μια ακμή ετικέτας l. Επίσης θέλουμε να επιβάλουμε την απαρίθμηση των multi-way συνενώσεων που περιέχουν *όλα* τα ερωτήματα τριάδας που μοιράζονται μια συγκεκριμένη μεταβλητή. Για να το πετύχουμε, το $N_{label}(S_1, X)$ απαιτείται να περιέχει μόνο ζευγάρια (l, S_2), έτσι ώστε το $S_1 \cup S_2$ να περιέχει *όλους* τους κόμβους που περιέχουν μια ακμή ετικέτας l. Για κάθε ζευγάρι της labelled γειτονιάς του S_1 απαριθμούμε όλες τις πιθανές *cmp-lc-lists* που περιέχουν το S_1 ως πρώτο σύνολο και επιβάλουμε τη διάταξη των συνόλων των *cmp-lc-list* που παρουσιάστηκε στην ενότητα 3.4.1 χρησιμοποιώντας τη συνάρτηση *enumerateCmpLcList*.

Η συνάρτηση *enumerateCmpLcList($list, S, X, l$)* σπάει αναδρομικά το σύνολο S σε δυο σύνολα: το S_1 που περιέχει το μικρότερο κόμβο του S σύμφωνα με το $\prec$ και το $S_2 = S \setminus S_1$. Το S_1 εισάγεται στην *cmp-lc-list* και η διαδικασία απαρίθμησης συνεχίζει μέχρι το S_2 να μην έχει άλλους κόμβους. Αυτή η διαδικασία εξασφαλίζει ότι η *cmp-lc-lists* ικανοποιεί την ιδιότητα

Algorithm 8: $enumerateCmpLcList(list, S, X, l)$

```
1  if S = ∅ then
2      extendCmpLcList(list, 1, X, l);
3      return;
4  end
5  m = min_≺{v ∈ S};
6  for S_1 ⊆ S \ m do
7      S_1 = S_1 ∪ m; S_2 = S \ S_1; list.push(S_1);
8      enumerateCmpLcList(list, S_2, X, l);
9      list.pop();
10 end
```

διάταξης που παρουσιάστηκε στην ενότητα 3.4.1. Τέλος, η *enumerateCmpLcList* μέθοδος καλεί την συνάρτηση *extendCmpLcList* για να επεκτείνει αναδρομικά όλα τα υποσύνολα μιας *cmp-lc-list*.

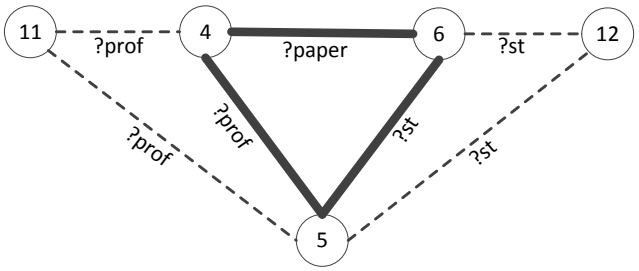
Algorithm 9: $extendCmpLcList(list, i, X, l)$

```
1  if i < list.size() − 1 then
2      extendCmpLcList(list, i + 1, X, l);
3  else
4      emitCmpLcList(list, l);
5  end
6  S = list.get(i);
7  for N ⊆ 𝒩(S, X) : N ≠ ∅ do
8      S_1 = S ∪ N; list.put(i, S_1);
9      extendCmpLcList(list, i, X ∪ 𝒩(S, X), l);
10 end
11 list.put(i, S);
```

Η συνάρτηση *extendCmpLcList(*$list, i, X, l$*)* απαριθμεί όλες τις πιθανές επεκτάσεις του i-οστού υπογράφου μιας *cmp-lc-list*. Ο πρώτος υπογράφος έχει επεκταθεί αναδρομικά από την συνάρτηση *enumerateCsgRec* και άρα η διαδικασία επέκτασης ξεκινάει από το δεύτερο στοιχείο της λίστας $i = 1, \ldots, list.size() - 1$. Η συνάρτηση *emitCmpLcList(*$list, l$*)* είναι το τελικό στάδιο του αλγορίθμου δυναμικού προγραμματισμού μας και είναι υπεύθυνη για τον υπολογισμό του κόστους εκτέλεσης της multi-way συνένωσης που περιγράφεται από το $(list, l)$, χρησιμοποιώντας το μοντέλο κόστους εκτέλεσης συνενώσεων, και για την ανανέωση της αντίστοιχης εγγραφής του πίνακα δυναμικού προγραμματισμού για τον γράφο του αποτελέσματος.

Στο παράδειγμά μας, η *emitCsg*({5}) συνάρτηση θα καλέσει την *enumerateCmpLcList* μια φορά για κάθε labelled γειτονιά του {5} ([st,{6,12}]). Σε αυτή την περίπτωση, οι λίστες δεν μπορούν να επεκταθούν περαιτέρω λόγω των περιορισμών διάταξης και έτσι η *emitCmpLcList* θα κληθεί δυο φορές: [st|{5}, {6}, {12}], [st|{5}, {6, 12}]. Στη περίπτωση του {4}, η labelled γειτονιά

{12}, {11}, {6}, {6, 12}, {5}, [st|{5}, {6}, {12}], [st|{5}, {6, 12}], {5, 6}, [st|{5, 6}, {12}], {5, 11}, [st|{5, 11}, {6}, {12}], [st|{5, 11}, {6, 12}], {5, 12}, [st|{5, 12}, {6}], {5, 6, 11}, [st|{5, 6, 11}, {12}], {5, 6, 12}, {5, 11, 12}, [st|{5, 11, 12}, {6}], {5, 6, 11, 12}, {4}, [prof|{4}, {5}, {11}], [prof|{4}, {5, 6}, {11}], [prof|{4}, {5, 12}, {11}], [prof|{4}, {5, 6, 12}, {11}], [prof|{4}, {5, 11}], [prof|{4}, {5, 6, 11}], [prof|{4}, {5, 11, 12}], [prof|{4}, {5, 6, 11, 12}], [paper|{4}, {6}], [paper|{4}, {5, 6}], [paper|{4}, {5, 6, 11}], [paper|{4}, {6, 12}], [paper|{4}, {5, 6, 12}], [paper|{4}, {5, 6, 11, 12}], {4, 5}, [prof|{4, 5}, {11}], [paper|{4, 5}, {6}], [paper|{4, 5}, {6, 12}], [st|{4, 5}, {6}, {12}], [st|{4, 5}, {6, 12}], {4, 6}, [prof|{4, 6}, {5}, {11}], [prof|{4, 6}, {5, 12}, {11}], [prof|{4, 6}, {5, 11}], [prof|{4, 6}, {5, 11, 12}], [st|{4, 6}, {5}, {12}], [st|{4, 6}, {5, 11}, {12}], [st|{4, 6}, {5, 12}], [st|{4, 6}, {5, 11, 12}], {4, 11}, [prof|{4, 11}, {5}], [prof|{4, 11}, {5, 6}], [prof|{4, 11}, {5, 12}], [prof|{4, 11}, {5, 6, 12}], [paper|{4, 11}, {6}], [paper|{4, 11}, {5, 6}], [paper|{4, 11}, {6, 12}], [paper|{4, 11}, {5, 6, 12}], {4, 5, 6}, [prof|{4, 5, 6}, {11}], [st|{4, 5, 6}, {12}], {4, 5, 11}, [paper|{4, 5, 11}, {6}], [paper|{4, 5, 11}, {6, 12}], [st|{4, 5, 11}, {6}, {12}], [st|{4, 5, 11}, {6, 12}], {4, 6, 11}, [prof|{4, 6, 11}, {5}], [prof|{4, 6, 11}, {5, 12}], [st|{4, 6, 11}, {5}, {12}], [st|{4, 6, 11}, {5, 12}], {4, 5, 6, 11}, [st|{4, 5, 6, 11}, {12}], {4, 5, 12}, [prof|{4, 5, 12}, {11}], [paper|{4, 5, 12}, {6}], [st|{4, 5, 12}, {6}], {4, 6, 12}, [prof|{4, 6, 12}, {5}, {11}], [prof|{4, 6, 12}, {5, 11}], [st|{4, 6, 12}, {5}], [st|{4, 6, 12}, {5, 11}], {4, 5, 6, 12}, [prof|{4, 5, 6, 12}, {11}], {4, 5, 11, 12}, [paper|{4, 5, 11, 12}, {6}], [st|{4, 5, 11, 12}, {6}], {4, 6, 11, 12}, [prof|{4, 6, 11, 12}, {5}], [st|{4, 6, 11, 12}, {5}], {4, 5, 6, 11, 12}



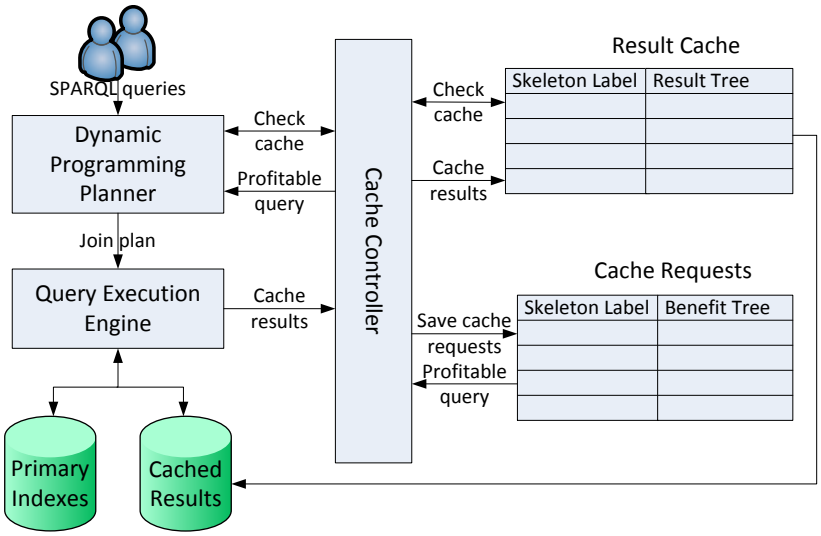
Σχήμα 3.9: *Εκτέλεση του αλγορίθμου βελτιστοποίησης για το ερώτημα Q_e*

είναι ([prof,{5,11}],[paper,{6}]). Εξετάζοντας την γειτονιά [prof,{5,11}], η αναδρομική απαρίθμηση της *enumerateCmpLcList* θα καλέσει την *extendCmpLcList* για τις ακόλουθες λίστες: [{4}, {5}, {11}], [{4}, {5,11}]. Τέλος, η *extendCmpLcList* θα επεκτείνει όλα τα υποσύνολα αυτών των λιστών καλώντας την *emitCmpLcList* για τις: i) [prof|{4}, {5}, {11}], [prof|{4}, {5, 6}, {11}], [prof|{4}, {5, 12}, {11}], [prof|{4}, {5, 6, 12}, {11}] ii) [prof|{4}, {5, 11}], [prof|{4}, {5, 6, 11}], [prof|{4}, {5, 11, 12}], [prof|{4}, {5, 6, 11, 12}].

3.5 Κρυφή μνήμη SPARQL αποτελεσμάτων

Στην ενότητα αυτή, περιγράφουμε το σύστημα κρυφής μνήμης SPARQL ερωτημάτων. Όπως απεικονίζεται στο Σχήμα 3.10, η ανάλυση του ερωτήματος ξεκινά από τον *Βελτιστοποιητή Πλάνου Εκτέλεσης Δυναμικού Προγραμματισμού*, που περιγράφηκε στην ενότητα 3.4, ο οποίος

εξετάζει όλα τα πιθανά πλάνα για τον προσδιορισμό του βέλτιστου πλάνου εκτέλεσης του ερωτήματος. Μέτα-δεδομένα των προσωρινά αποθηκευμένων αποτελεσμάτων αποθηκεύονται στην κύρια μνήμη χρησιμοποιώντας τις κανονικοποιημένες ετικέτες ως κλειδί του πίνακα *Result Cache*. Ενώ ο βελτιστοποιητής πλάνου εκτέλεσης εξετάζει όλους τους συνδεδεμένους υπογράφους του ερωτήματος, εξετάζει και τη χρησιμοποίηση των προσωρινά αποθηκευμένων αποτελεσμάτων χρησιμοποιώντας τις κανονικοποιημένες ετικέτες που παρουσιάστηκαν στην ενότητα 3.3.2. Το όφελος όλων των πιθανά χρησιμοποιούμενων αποτελεσμάτων εξετάζεται από το μοντέλο κόστους κατά την εκτέλεση του βελτιστοποιητή και άρα το βέλτιστο πλάνο εκτέλεσης μπορεί να περιέχει, εν μέρει, προσωρινά αποθηκευμένα αποτελέσματα. Ο *Ελεγκτής Κρυφής Μνήμης* είναι υπεύθυνος για την παρακολούθηση των αιτημάτων κρυφής μνήμης και τη διατήρηση αναλυτικών εκτιμήσεων για το όφελος των διαφόρων μοτίβων αποτελεσμάτων που μπορούν να αποθηκευτούν στην κρυφή μνήμη, καθώς και για το κόστος εκτέλεσής τους. Αυτές οι πληροφορίες αποθηκεύονται στον πίνακα *Cache Requests* και χρησιμοποιούνται για να την εκτέλεση και προσωρινή αποθήκευση *κερδοφόρων* ερωτημάτων (συχνά εμφανιζόμενων αλλά όχι αποθηκευμένων ερωτημάτων), προκειμένου να ενισχύσουν την χρήση της κρυφής μνήμης.



Σχήμα 3.10: *Αρχιτεκτονική του συστήματος*

3.5.1 Αφαίρεση σταθερών ερωτήματος

Σε αυτή την ενότητα περιγράφουμε πως μπορούμε να εντοπίσουμε και να εξετάσουμε την χρήση των αποθηκευμένων αποτελεσμάτων που μπορούν να προσφέρουν κάποιο όφελος στην εκτέλεση του τρέχοντος ερωτήματος. Ας εξετάσουμε το παρακάτω ερώτημα:

?prof worksFor "MIT" .
?prof emailAddress ?email .
?prof name "Mike" .

Χρησιμοποιώντας ακριβές ταίριασμα, ο βελτιστοποιητής μας θα εκτελούσε αιτήματα κρυφής μνήμης για τους υπογράφους του ερωτήματος που περιέχουν όλες τις σταθερές και τα URI του αρχικού ερωτήματος και οι οποίοι φαίνονται παρακάτω:

?prof worksFor "MIT" . ?prof name "Mike" .	?prof worksFor "MIT" . ?prof emailAddress ?email .
?prof emailAddress ?email . ?prof name "Mike" .	?prof worksFor "MIT" . ?prof emailAddress ?email . ?prof name "Mike" .

Όμως, μπορούμε να παρατηρήσουμε ότι το ακόλουθο προσωρινά αποθηκευμένο αποτέλεσμα θα μπορούσε επίσης να χρησιμοποιηθεί αποτελεσματικά για την απάντηση του ερωτήματος, μετατρέποντάς το σε μη απλή αναζήτηση ενός ευρετηρίου:

{ ?prof worksFor ?univ .
?prof emailAddress ?email
?prof name ?name } index by ?univ ?name

Για να αντιμετωπιστεί αυτό το σενάριο, θα πρέπει να εξετάσουμε επίσης πιο γενικές μορφές γράφων ερωτήματος, καθώς και τις ιδιότητες ευρετηρίασής τους, που μπορεί να βοηθήσουν στη μείωση του κόστους αναζήτησης του αποτελέσματος. Για παράδειγμα, στην παραπάνω περίπτωση, η προσωρινή αποθήκευση του γενικού αποτελέσματος, χωρίς καμία ευρετηρίαση, θα ήταν άσκοπη αν το μέγεθός του ήταν σημαντικά μεγαλύτερο από το μέγεθος των φιλτραρισμένων αποτελεσμάτων, αφού θα έπρεπε να διαβάσουμε όλα τα δεδομένα του για να βρούμε αυτά που ταιριάζουν με το τρέχον ερώτημα. Επιπλέον, η χρηστικότητα ενός προσωρινά αποθηκευμένου αποτελέσματος εξαρτάται από τις συνενώσεις που απαιτείται να εκτελεστούν σε αυτό. Για παράδειγμα, εάν πρέπει να συνενώσουμε αυτό το αποτέλεσμα με ένα άλλο ερώτημα τριπλέτας σύμφωνα με τη μεταβλητή *?prof*, θα θέλαμε να το έχουμε ταξινομημένο ώστε να μπορέσουμε να εκτελέσουμε μια merge συνένωση και όχι μια πιο πολύπλοκη hash ή sort-merge συνένωση. Επίσης, πρέπει να λάβουμε υπόψη την περίπτωση της προσωρινής αποθήκευσης αποτελεσμάτων που περιέχουν πιο σύνθετα φίλτρα για τις μεταβλητές, προβολές και ομαδοποιήσεις. Για την επίτευξη όλων αυτών των στόχων, θα πρέπει να βρούμε έναν αποτελεσματικό τρόπο για να εξετάζουμε ποια από τα προσωρινά αποθηκευμένα αποτελέσματα μπορούν να συνεισφέρουν στην ανάκτηση των αποτελεσμάτων ενός υπογράφου του ερωτήματος, και να τα ελέγξουμε ώστε να ανακαλύψουμε αυτό που απαιτεί το μικρότερο δυνατό κόστος.

Για την αντιμετώπιση αυτού του προβλήματος, αφαιρούμε τις σταθερές από το γράφο του ερωτήματος. Αφαιρούμε όλους τους δεσμευμένους κόμβους του ερωτήματος που βρίσκονται στις θέσεις subject και object ενός ερωτήματος τριάδας και τις αντικαθιστούμε με μεταβλητές, μαζί με τα αντίστοιχα φίλτρα. Στο παραπάνω παράδειγμα, το ερώτημα μετατρέπεται σε:

{ ?prof ub:worksFor ?univ .
?prof ub:emailAddress ?email .
?prof ub:name ?name } filter(?univ="MIT", ?name="Mike")

Χρησιμοποιούμε αυτή την αναπαράσταση για το γράφο του ερωτήματος για την εκτέλεση του βελτιστοποιητή δυναμικού προγραμματισμού μας και έτσι μπορούμε να ελέγξουμε όλους υπογράφους της εκτελώντας αιτήματα κρυφής μνήμης. Κάθε αίτημα κρυφής μνήμης έχει ως κλειδί την κανονικοποιημένη ετικέτα σκελετού του απλοποιημένου και αφηρημένου υπογράφου του ερωτήματος και συνοδεύεται από τα φίλτρα, τις προβολές και τις ομαδοποιήσεις που απαιτούνται, καθώς και από ένα αίτημα για μια μεταβλητή συνένωσης. Όπως αναφέρθηκε στο κεφάλαιο 3.3 τα φίλτρα, οι προβολές και οι ομαδοποιήσεις δεν έχουν χρησιμοποιηθεί για να δημιουργηθεί η κανονικοποιημένη ετικέτα και επομένως, τα ερωτήματα με την ίδια αφηρημένη δομή αλλά διαφορές στις υπόλοιπες ιδιότητες ομαδοποιούνται με βάση την ετικέτα τους.

3.5.2 Δέντρο Αποτελεσμάτων

Η χρήση των κανονικοποιημένων ετικετών για την πρόσβαση στην κρυφή μνήμη αποτελεσμάτων, *Result Cache*, μειώνει τα αποτελέσματα που πρέπει να εξετάσουμε για κάθε υπογράφο του ερωτήματος, αλλά χρειαζόμαστε ακόμα έναν αποτελεσματικό τρόπο για να διαλέξουμε το καλύτερο αποτέλεσμα από τη λίστα όλων των αποτελεσμάτων που μοιράζονται την ίδια αφηρημένη δομή σκελετού. Κάθε εγγραφή κρυφής μνήμης, που σχετίζεται με μια αφηρημένη ετικέτα σκελετού, ομαδοποιεί όλα τα προσωρινά αποθηκευμένα αποτελέσματα χρησιμοποιώντας μια δομή δέντρου, που ονομάζεται *Result Tree*. Η δομή αυτή μπορεί στη συνέχεια να χρησιμοποιηθεί για: i) να εκτελεί αποτελεσματικά δοκιμές ισομορφισμού υπογράφου σκελετού-αστέρα και ii) για να μπορούμε να αναζητούμε αποτελεσματικά το καλύτερο αποτέλεσμα σε σχέση με τις επιπρόσθετες πληροφορίες του αιτήματος, auxiliary info (φίλτρα, προβολές, ομαδοποιήσεις). Το Σχήμα 3.11 αναπαριστά αυτή τη δομή για τον αφηρημένο γράφο του ερωτήματος σκελετού που χρησιμοποιείται στην ενότητα 3.2:

?3 author ?2 .
?3 advisor ?1 .
?1 author ?2 .

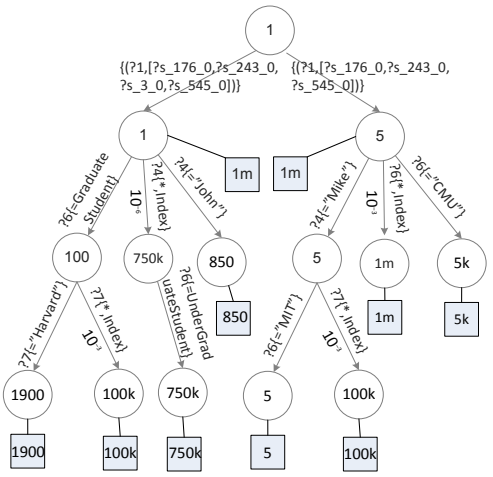
Για την καλύτερη κατανόηση, χρησιμοποιούμε τα κανονικοποιημένα ονόματα για τις μεταβλητές, τα οποία παρουσιάστηκαν στην ενότητα Section 3.3.1. Σε αυτό το παράδειγμα υποθέτουμε ότι η κρυφή μνήμη μας περιέχει διάφορα προσωρινά αποθηκευμένα αποτελέσματα με τις παρακάτω αφηρημένες ετικέτες αστέρων:

?1 fullName ?4 .
?1 emailAddress ?5 .
?1 studiesIn ?7 .
?1 type ?6 .

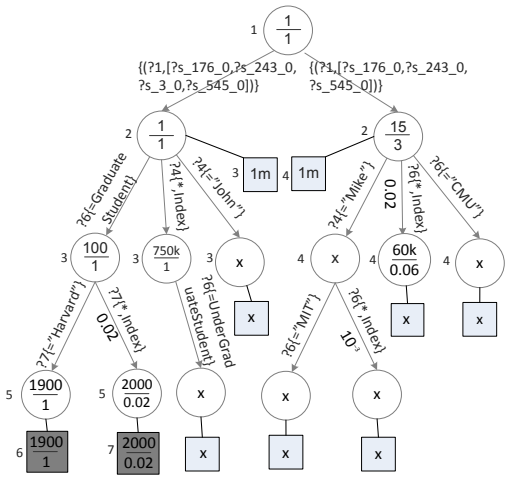
?1 fullName ?4 .
?1 emailAddress ?5 .
?1 studiesIn ?6 .

Οι πρώτες ακμές του δέντρου καταγράφουν τις αφηρημένες ετικέτες αστέρων των αποθηκευμένων αποτελεσμάτων. Κάθε άλλη ακμή περιέχει πληροφορίες φίλτρων και ευρετηρίασης που σχετίζονται με μια συγκεκριμένη μεταβλητή του ερωτήματος. Κάθε φύλο του δέντρου περιγράφει ένα προσωρινά αποθηκευμένο αποτέλεσμα και άρα το μονοπάτι που το συνδέει με την κορυφή του δέντρου καταγράφει όλες τις επιπρόσθετες πληροφορίες του. Για παράδειγμα, μια ακμή με ετικέτα *?4{"*, Index"}* σημαίνει ότι το αποτέλεσμα δεν έχει κανένα φίλτρο σε αυτή τη μεταβλητή και επίσης υπάρχει ένα ευρετήριο για αυτή. Μια ακμή με ετικέτα *?6{="MIT"}* δηλώνει ότι το αποτέλεσμα περιέχει ένα φίλτρο *?6="MIT"*. Μια ακμή "Π" σημαίνει ότι η συγκεκριμένη μεταβλητή έχει αφαιρεθεί μέσω προβολής από το αποτέλεσμα. Αν το ερώτημα περιέχει ομαδοποιήσεις αυτές καταγράφονται ως ακμές τελευταίου επιπέδου στο δέντρο αποτελεσμάτων. Για να ελέγξουμε τη χρησιμότητα των προσωρινά αποθηκευμένων αποτελεσμάτων για ένα αίτημα κρυφής μνήμης, διασχίζουμε το δέντρο αποτελεσμάτων από την κορυφή και βρίσκουμε τα αποτελέσματα που μπορούν να χρησιμοποιηθούν για την εκτέλεσή του. Όταν διασχίζουμε τις ακμές των ετικετών αστέρα εκτελούμε μια εξέταση ισομορφισμού υπογράφου σκελετού-αστέρα ώστε να ακολουθήσουμε μόνο τα υποδέντρα που μπορούν να χρησιμοποιηθούν για την απάντηση του ερωτήματος.

Εκτός από τον έλεγχο της χρηστικότητας των προσωρινά αποθηκευμένων αποτελεσμάτων με βάση ένα αίτημα κρυφής μνήμης, θα πρέπει επίσης να είμαστε σε θέση να αξιολογήσουμε το κόστος χρήσης των αποτελεσμάτων και να εντοπίσουμε το αποτέλεσμα που ταιριάζει καλύτερα στο αίτημα της κρυφής μνήμης. Ως κόστος ενός προσωρινά αποθηκευμένου αποτελέσματος, αναφερόμαστε στην ποσότητα των δεδομένων που πρέπει να διαβαστούν για τη χρήση του. Για να εκτιμηθεί το κόστος αυτό υπό την παρουσία πολλαπλών ευρετηρίων μεταβλητών και φίλτρων αποτελεσμάτων, πρέπει να επεκτείνουμε τη δομή δέντρου μας με εκτιμήσεις επιλεκτικότητας και μεγέθη αποτελεσμάτων. Η διαδικασία εισαγωγής ενός προσωρινά αποθηκευμένου αποτελέσματος σε ένα δέντρο αποτελεσμάτων φαίνεται στον Αλγόριθμο 10. Η μέθοδος *treeInsert* προσθέτει αναδρομικά ένα νέο κόμβο φύλλο, που αντιπροσωπεύει το τρέχον αποτέλεσμα, μαζί με τον αριθμό των εγγραφών του. Η τιμή *minResults* για κάθε κόμβο του δένδρου είναι ορατή μέσα σε κάθε κόμβο του Σχήματος 3.11 και αποτελεί μια εκτίμηση



Σχήμα 3.11: *Δέντρο προσωρινά αποθηκευμένων αποτελεσμάτων*



Σχήμα 3.12: *Αναζήτηση του δέντρου αποτελεσμάτων για το ερώτημα Q_r*

του χαμηλότερου κόστους που μπορεί να επιτευχθεί από τα αποτελέσματα του αντίστοιχου υποδένδρου. Έχοντας εγγυήσεις για το χαμηλότερο κόστος ενός υποδένδρου, μπορούμε να εκτελέσουμε την αναζήτηση για ένα αίτημα κρυφής μνήμης χρησιμοποιώντας έναν αποτελεσματικό αλγόριθμο A* αναζήτησης. Ο αλγόριθμος αυτός μπορεί να χρησιμοποιήσει τις παραπάνω εκτιμήσεις για την αποφυγή εξερεύνησης ολόκληρων υποδένδρων που δεν περιέχουν χρήσιμα αποτελέσματα. Για να δημιουργήσουμε τις ελάχιστες εκτιμήσεις κόστους, κάθε κόμβος φύλλο περιέχει το πραγματικό μέγεθος του αποτελέσματος και η τιμή αυτή διαδίδεται προς τα πάνω κρατώντας κάθε φορά την ελάχιστη τιμή μεταξύ όλων των κόμβων παιδιών. Στην περίπτωση ακμών ευρετηρίασης, εφαρμόζουμε μια εκτίμηση επιλεκτικότητας πριν μεταφέρουμε την τιμή από το παιδί στον κόμβο πατέρα. Αν μια αίτηση κρυφής μνήμης περιέχει ένα φίλτρο σε μια μεταβλητή που έχει ευρετήριο, θα πρέπει να μειώσουμε το κόστος του αποτελέσματος ανάλογα με την αναμενόμενη επιλεκτικότητα του φίλτρου πάνω στο ευρετήριο. Οι ακμές που δεν αντιστοιχούν σε ευρετήρια δεν αλλάζουν το κόστος του αποτελέσματος αφού θα πρέπει να ανακτήσουμε όλα τους τα δεδομένα για να εκτελέσουμε τη λειτουργία φιλτραρίσματος.

Η υλοποίηση της κρυφής μνήμης μας δεν εξαρτάται από το συγκεκριμένο τρόπο που μια μηχανή εκτέλεσης SPARQL ερωτημάτων χειρίζεται τις εκτιμήσεις επιλεκτικότητας. Για να δημιουργήσουμε το δέντρο αποτελεσμάτων απαιτείται μόνο η μέγιστη επιλεκτικότητα που μπορεί να επιτευχθεί από μια λειτουργία φιλτραρίσματος σε ένα συγκεκριμένο ευρετήριο. Για κάθε ακμή ευρετηρίου ενός προσωρινά αποθηκευμένου αποτελέσματος, χρησιμοποιούμε μια τιμή επιλεκτικότητας $s = minRecords/totalRecords$ η οποία απεικονίζεται κατά μήκος των ακμών ευρετηρίων του σχήματος 3.11. Στο παράδειγμά μας, ένα ευρετήριο για το γενικό αποτέλεσμα (1 εκατομμύριο εγγραφές), στη μεταβλητή *?7 (?univ)* δίνει ένα ελάχιστο ποσό 1.000 εγγραφών και έτσι μέγιστη επιλεκτικότητα του είναι $s = 1000/1m = 10^{-3}$. Για να χειριστούμε

Algorithm 10: $cacheResult$

```
1  function cacheResult(V, E, aux, size)
2    //V, E : κόμβοι και avertices and edges of the abstract query graph
3    //aux : επιπρόσθετες πληροφορίες ερωτήματος (filters, projections, etc)
4    //size : το μέγεθος του αποτελέσματος σε αριθμό εγγραφών
5    (skeletonLabel, starLabel) ← canonicalLabel(V, E);
6    resultTree ← ResultCache.get(skeletonLabel);
7    aux.addStarLabel(starLabel);
8    treeInsert(resultTree.root, aux, size);
9  function treeInsert(node, aux, size)
10   if aux.isEmpty() then
11       createLeaf(size);
12   else
13       aInfo ← aux.getNext();
14       if ((edge, child) = node.getEdge(aInfo)) = null) then
15           //Η ακμή δεν υπάρχει, δημιουργούμε μια νέα
16           (edge, child) = newEdge(aInfo);
17       end
18       treeInsert(child, aux, size);
19       if aInfo.isIndexed() then
20           //υπολογίζουμε την μέγιστη επιλεκτικότητα του ευρετηρίου
21           maxSel ← maxSelectivity(edge);
22           results = child.minResults * maxSel;
23       else
24           results = child.minResults;
25       end
26       if results < node.minResults then
27           if results ≤ 1 then
28               results ← 1;
29           end
30           node.minResults = results;
31       end
32   end
```

την εκτίμηση των πολλαπλών λειτουργιών φιλτραρίσματος σε διαφορετικές μεταβλητές χρησιμοποιούμε την αρχή της ανεξαρτησίας, δηλαδή, η επιλεκτικότητα των δύο λειτουργιών φιλτραρίσματος με επιλεκτικότητα s_1 και s_2 είναι $s = s_1 \cdot s_2$. Αυτή η ιδιότητα μας επιτρέπει να ακολουθήσουμε μονοπάτια φίλτρων και να διατηρήσουμε μια συνολική εκτίμηση χρησιμοποιώντας τον πολλαπλασιασμό των ατομικών εκτιμήσεων κατά τη διέλευσή μας από τις ακμές ευρετηρίων. Η *treeInsert* μέθοδος (Αλγόριθμος 10) ξεκινά με την προσθήκη των αντίστοιχων φύλλων μαζί με τον αριθμό εγγραφών τους. Μπορούμε στη συνέχεια να προχωρήσουμε από το φύλλο προς τον κόμβο ρίζα και να ενημερώσουμε τις ελάχιστες τιμές των κόμβων των υψηλότερων επιπέδων. Κατά τη διέλευση από μια ακμή ευρετηρίου εφαρμόζουμε τη μέγιστη εκλεκτικότητα πολλαπλασιάζοντάς τη με τον αριθμό εγγραφών του κόμβου παιδιού. Θέτουμε επίσης

μια ελάχιστη τιμή 1 για το κόστος των κόμβων. Η πολυπλοκότητα της λειτουργίας εισαγωγής ενός αποτελέσματος στην κρυφή μνήμη έχει γραμμική πολυπλοκότητα σε σχέση με τον αριθμό τον επιπρόσθετων πληροφοριών του ερωτήματος, μέγεθος που περιορίζεται από το συνολικό αριθμό των μεταβλητών του αφηρημένου ερωτήματος.

Algorithm 11: $CheckCache$

```
1  function checkCache(V, E, aux, k)
2      //V, E : κόμβοι και ακμές του αφηρημένου ερωτήματος
3      //aux : επιπρόσθετες πληροφορίες (filters, projections, etc)
4      //k : αναζήτηση για τα top-k αποτελέσματα
5      (skeletonLabel, starLabel) ← canonicalLabel(V, E);
6      resultTree ← ResultCache.get(skeletonLabel);
7      aux.addStarLabel(starLabel);
8      return searchTree(resultTree, aux, k);
9  function searchTree(resultTree, aux, k)
10     results ← {};
11     //openNodes : priority queue with pairs (node, cost)
12     openNodes ← {(resultTree.root, 1)};
13     while openNodes ≠ {} do
14         //παίρνουμε τον ανοιχτό κόμβο με το ελάχιστο κόστος
15         n ← openNodes.removeHead();
16         if (results.size = k)and(n.cost ≥ results.maxCost) then
17             //Εχουμε βρει k αποτελέσματα και όλοι οι ανοιχτοί κόμβοι έχουν μεγαλύτερο κόστος
18             return results;
19         end
20         processNode(n, openNodes, results, aux, k);
21     end
22     return results;
23 function processNode(n, openNodes, results, aux, k)
24     for (edge, child) ∈ n.children() do
25         if s = selectivity(edge, aux) > 0 then
26             child.selectivity ← s * n.selectivity;
27             child.cost ← child.minResults * child.selectivity;
28             if child.isLeaf() then
29                 //διατηρούμε τα k καλύτερα αποτελέσματα
30                 if results.size < k then
31                     results.add(child);
32                 elseif results.maxCost > child.cost then
33                     results.removeMaxAndAdd(child);
34                 end
35             else
36                 openNodes.add({child, child.cost});
37             end
38         end
39     end
```

Όπως αναφέρθηκε προηγουμένως, χρησιμοποιούμε τις εκτιμήσεις ελάχιστου κόστους για να εκτελέσουμε για A* αναζήτηση και να αποφύγουμε την εξερεύνηση υποδέντρων που έχουν μεγάλα κόστη. Η αναζήτηση ξεκινά από την ρίζα του δέντρου και εξετάζει όλους τους κόμβους με χρήση μίας best-first αναζήτησης, σύμφωνα με τις εκτιμήσεις κόστους. Το Σχήμα 3.12 απεικονίζει την εκτέλεση της μεθόδου *searchTree* (Αλγόριθμος 11) καθώς ψάχνουμε την κρυφή μνήμη για τα αίτημα (Q_r):

{ ?3 author ?2 .
?3 advisor ?1 .
?1 author ?2 .
?1 fullName ?4 .
?1 emailAddress ?5 .
?1 studiesIn ?7 .
?1 type ?6 .
} filter(?6=GraduateStudent, ?7="Harvard")

Στο Σχήμα 3.12 κάθε κόμβος περιέχει 2 αριθμούς: το εκτιμώμενο κόστος του (επάνω αριθμός) και η επιλεκτικότητά του (κάτω αριθμός). Όταν διασχίζουμε τις ακμές του δέντρου ο Αλγόριθμός μας ελέγχει τις παρακάτω ιδιότητες:

- Αν μια συγκεκριμένη ακμή μπορεί να χρησιμοποιηθεί για την ανάκτηση των αποτελεσμάτων του ερωτήματος. Για παράδειγμα, η ακμή με ετικέτα *?4="Mike"* δε θα ακολουθηθεί αφού είναι πιο περιοριστική από το ερώτημα.

- Όταν διασχίζουμε ακμές ευρετηρίων που ταιριάζουν με φίλτρα του ερωτήματος πρέπει να εφαρμόσουμε την επιλεκτικότητά τους. Για παράδειγμα, καθώς διασχίζουμε την ακμή *(?7: "*,Indexed")* πρέπει να εφαρμόσουμε την επιλεκτικότητα του φίλτρου *?7="Harvard"*. Για να το πετύχουμε αυτό, χρησιμοποιούμε έναν εκτιμητή επιλεκτικότητας για το αφηρημένο ερώτημα σκελετού-αστέρα και για τη συγκεκριμένη μεταβλητή. Χρησιμοποιούμε το αφηρημένο αποτέλεσμα για να εκτιμήσουμε τις επιλεκτικότητες γιατί μια ακμή του δέντρου μπορεί να ανήκει σε πολλαπλά αποτελέσματα με διαφορετικά μεγέθη και χρειαζόμαστε μια εκτίμηση της μέγιστης επιλεκτικότητας. Σε αυτή την περίπτωση, η επιλεκτικότητα του *?7="Harvard"* είναι 0.02 επειδή περιμένουμε να επιστρέψει 20 χιλιάδες αποτελέσματα και το αφηρημένο αποτέλεσμα περιέχει 1 εκατομμύριο εγγραφές.

- Όταν διασχίζουμε τις ακμές που αντιστοιχούν στη μεταβλητή συνένωσης πολλαπλασιάζουμε την επιλεκτικότητα με 2, αν η ακμή δεν είναι ακμή ευρετηρίου. Αυτό προκύπτει από το γεγονός ότι το κόστος μιας hash ή sort-merge συνένωσης σε σχέση με μια merge

συνένωση μπορεί να πορσεγγιστεί από το χρόνο που χρειαζόμαστε για να διαβάσουμε τα δεδομένα εισόδου δυο φορές [Neumann 10a].

- Όσον αφορά τις ακμές ετικετών αστέρα, θα πρέπει να εκτελέσουμε μια μια δοκιμή ισομορφισμού υπογράφου σκελετού-αστέρα προκειμένου να ελεγχθεί η δυνατότητα χρήσης των υφιστάμενων υποδένδρων των αποτελεσμάτων. Αν η ετικέτα αστέρα ταιριάζει ακριβώς με την ετικέτα του αιτήματος κρυφής μνήμης δεν αλλάζουμε την εκτίμηση της επιλεκτικότητας. Σε περίπτωση προσωρινής αποθήκευσης αποτελεσμάτων που είναι υπογράφοι του αιτήματος, πρέπει να αλλάξουμε την επιλεκτικότητα τους, προκειμένου να αντικατοπτριστεί το κόστος της συνένωσής τους τα υπόλοιπα ερωτήματα τριάδας των μοτίβων αστέρα. Πρέπει, επίσης, να αλλάξουμε την κανονικοποιημένη αρίθμηση των μεταβλητών το συγκεκριμένο υποδέντρο, διότι ορισμένες από τις μεταβλητές αστέρα αλλάζουν όνομα λόγω τις έλλειψης των ερωτημάτων τριάδας. Στο παράδειγμά μας, όταν διασχίζουμε το δεξί υποδέντρο η ?univ μεταβλητή έχει όνομα ?6 και όχι ?7. Όσον αφορά την εκτίμηση της επιλεκτικότητας, την ορίζουμε ως 2 συν τον αριθμό των ερωτημάτων τριάδας που λείπουν. Ο αριθμός αυτός προσεγγίζει το κόστος της εκτέλεσης μιας sort-merge συνένωσης, η οποία θα διαβάσει δύο φορές τα δεδομένα του αποτελέσματος και μια φορά τα ταξινομημένα αποτελέσματα των ερωτημάτων τριάδας.

Για να εκτιμηθεί η επιλεκτικότητα πολλαπλών διαδοχικών ακμών διατηρούμε μια εκτίμηση επιλεκτικότητας για κάθε ανοιχτό κόμβο και μεταφέρουμε τις τιμές στους κόμβους παιδιά πολλαπλασιάζοντας τη όταν διασχίζουμε ακμές ευρετηρίων που συνδυάζονται με φίλτρα. Η εκτίμηση του ελάχιστου κόστους για κάθε κόμβο (επάνω αριθμός) υπολογίζεται με τον πολλαπλασιασμό της εκλεκτικότητας με την τιμή *minResults* και απεικονίζεται στο Σχήμα 3.11. Η εκτίμηση της επιλεκτικότητας μιας ακμής με βάση τις επιπρόσθετες πληροφορίες του αιτήματος κρυφής μνήμης περιγράφεται λεπτομερώς στη συνάρτηση $selectivity(edge, aux)$ του Αλγορίθμου 12.

Οι εκτιμήσεις κόστους μας μπορεί να αποκλίνουν από τις πραγματικές γεγονός που οφείλεται στη χρήση των εκτιμητών των αφηρημένων αποτελεσμάτων και την υπόθεσή της ανεξαρτησίας για τις πράξεις φιλτραρίσματος. Ως εκ τούτου, εκτελούμε μια αναζήτηση top-k για τα αποτελέσματα και στη συνέχεια εξετάζουμε περαιτέρω το κόστος του κάθε αποτελέσματος, μέσα στον αλγόριθμο *DPccp*, χρησιμοποιώντας την αναλυτική μέθοδο εκτίμησης κόστους εκτέλεσης της χρησιμοποιούμενης RDF βάσης δεδομένων. Στο Σχήμα 3.12 εκτελούμε μια top-2 αναζήτηση, απεικονίζοντας δίπλα σε κάθε κόμβο το βήμα εκτέλεσης κατά το οποίο τον ανοίξαμε. Οι κόμβοι που σημειώνονται με ένα '$\times$' δεν εξετάστηκαν και οι κόμβοι με γκρι χρώμα απεικονίζουν τα αποτελέσματα της αναζήτησης. Παρατηρούμε ότι απαιτούνται 7 βήματα για να βρούμε τα top-2 καλύτερα αποτελέσματα και η αναζήτηση μας κατάφερε να αποφύγει την

Algorithm 12: *Selectivity estimation*

```
1  function selectivity(edge, aux)
2    //Υπολογισμός επιλεκτικότητας για το aux info
3    selectivity ← 1;
4    if edge.isStarLabel() then
5      if edge.skeletonStarSubgraphIsomorphic(aux) then
6        aux.setNewCanonicalVariableMapping();
7        selectivity ← 2 + patternsToBeJoined();
8      else
9        return 0;
10     end
11   else
12     for a ∈ aux.get(edge.variable) do
13       //έλεγχος χρήστικότητας της ακμής
14       if edge.subsumes(a) then
15         if a.isFilter then
16           if edge.isIndexed then
17             //επιλεκτικότητα φίλτρου με χρήση του εκτιμητή του αφηρημένου
                 αποτελέσματος
18             selectivity ← selectivity * filterSelectivity(a);
19           end
20         else if a.isJoinVariable and edge.isNotIndexed then
21           selectivity ← selectivity * 2;
22         end
23       else
24         return 0;
25       end
26     end
27   end
28   return selectivity;
```

εξερεύνηση ενός μεγάλου μέρους του δέντρου αναζήτησης, λόγω των επιπρόσθετων πληροφοριών τους και των ελάχιστων εκτιμήσεων κόστους.

Η εξάρτηση της A* αναζήτησης από τις εκτιμήσεις κόστους και από τα αποτελέσματα του δέντρου δεν μας επιτρέπει να δώσουμε ένα χρήσιμο άνω όριο πολυπλοκότητας για αυτόν τον αλγόριθμο. Φυσικά, ένα άνω φράγμα για τον αλγόριθμο είναι το μέγιστο μέγεθος του δέντρου των αποθηκευμένων αποτελεσμάτων μέσα σε μια εγγραφή της κρυφής μνήμης, αλλά αυτό το όριο δεν λαμβάνει υπόψη την έξυπνη μείωση του χώρου αναζήτησης που επιτυγχάνεται. Στη χειρότερη περίπτωση, η αναζήτηση του δέντρου αποτελεσμάτων θα έχει πολυπλοκότητα O(r), όπου r είναι το μέγιστο ποσό των προσωρινά αποθηκευμένων αποτελεσμάτων που μοιράζονται την ίδια αφηρημένη δομή σκελετού. Σημειώνουμε εδώ ότι το r φράσσεται επίσης από τους περιορισμούς του μεγέθους της κρυφής μνήμης, όπως θα συζητήσουμε στην επόμενη ενότητα, και ως εκ τούτου μπορούμε να περιμένουμε ότι δεν μπορεί να αυξηθεί απεριόριστα. Η *checkCache*

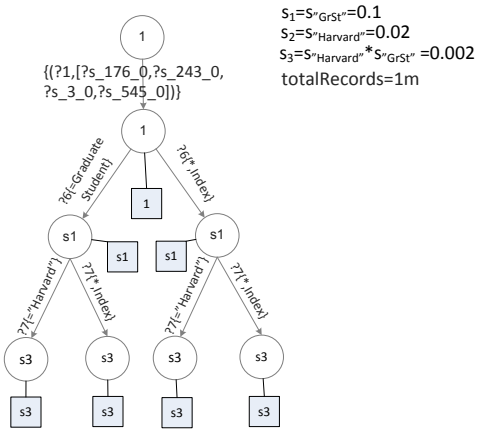
μέθοδος (Αλγόριθμος 11), δημιουργεί μια κανονικοποιημένη ετικέτα και, στη συνέχεια, εκτελεί μια αναζήτηση στο αντίστοιχο δέντρο αποτελεσμάτων. Έτσι, ο χρόνος που απαιτείται για την κανονικοποιημένη δημιουργία ετικέτας θα κυριαρχήσει, στη χειρότερη περίπτωση, στην πολυπλοκότητα της checkCache μεθόδου λόγω της εκθετική πολυπλοκότητάς της. Ωστόσο, όπως φαίνεται στην πειραματική αξιολόγηση μας, ο μηχανισμός checkCache είναι πρακτικός και παρουσιάζει αποδεκτή χρονική πολυπλοκότητα για πολύπλοκους γράφους ερωτημάτων.

3.6 Ελεγκτής Κρυφής Μνήμης

Σε αυτή την ενότητα, περιγράφουμε τη λειτουργία του ελεγκτή της κρυφής μνήμης μας. Τα καθήκοντά του είναι η παραγωγή και η προσωρινή αποθήκευση των *κερδοφόρων* προτύπων ερωτημάτων και η στρατηγική αντικατάστασης της προσωρινής μνήμης. Ως *κερδοφόρα* μοτίβα ερωτημάτων ορίζουμε ερωτήματα που, ακόμα και αν δεν έχουν ακριβώς ζητηθεί από τα ερωτήματα των χρηστών ή χρησιμοποιούνται ως ενδιάμεσα αποτελέσματα, θα μπορούσαν να ωφελήσουν την εκτέλεση των ερωτημάτων του φόρτου εργασίας, αν αποθηκευτούν. Για παράδειγμα, σκεφτείτε ένα φόρτο εργασίας με ερωτήματα που έχουν την ίδια αφηρημένη δομή ερωτήματος, αλλά τυχαία ID στις σταθερές τους (π.χ. όνομα καθηγητή). Η αποθήκευση μόνο των ενδιάμεσων αποτελεσμάτων αυτών των ερωτημάτων δεν θα προσέφερε μεγάλο κέρδος στην εκτέλεση του φόρτου εργασίας, διότι θα χρησιμοποιούσε την κρυφή μνήμη μόνο για ερωτήματα που ήταν ακριβώς ίδια με τα προηγούμενά τους. Σε αντίθεση, η προσωρινή αποθήκευση του αφηρημένου μοτίβου του ερωτήματος και η ευρετηρίασή του με βάση τη μεταβλητή των επιλεκτικών αναγνωριστικών εισάγει οφέλη για όλα τα ερωτήματα του φόρτου εργασίας καθώς τα μετατρέπει σε σαρώσεις ευρετηρίου. Εκτός από τον εντοπισμό των αφηρημένων αποτελεσμάτων και την ευρετηρίαση τους, ο ελεγκτής κρυφής μνήμης μπορεί να προσδιορίσει συχνά εμφανιζόμενους υπογράφους του φόρτου εργασίας και να τους ευρετηριάσει με βάση: 1) τις πιο κοινές μεταβλητές φιλτρων τους και 2) τις πιο κοινές μεταβλητές συνενώσεων.

3.6.1 Εύρεση κερδοφόρων ερωτημάτων

Όπως συζητήθηκε στην προηγούμενη ενότητα, ο αλγόριθμος δυναμικού προγραμματισμού μας εκτελεί αιτήσεις κρυφής μνήμης για όλους τους υπογράφους του αφηρημένου ερωτήματος. Επιπλέον, τα αιτήματα κρυφής μνήμης εκτελούνται ταυτόχρονα με τη βελτιστοποίηση του πλάνου εκτέλεσης του ερωτήματος και επομένως, μπορούν να καταγραφούν μαζί με τις εκτιμήσεις για τις επιπτώσεις τους στο χρόνο εκτέλεσης του αντίστοιχου ερωτήματος. Η διατήρηση ενός τόσο λεπτομερούς αρχείου καταγραφής των αιτημάτων κρυφής μνήμης παρέχει πολύτιμες πληροφορίες σχετικά με το ποια πρότυπα ερωτημάτων μπορούν να παρέχουν



Σχήμα 3.13: *Δέντρο Εκτίμησης Οφέλους για το αίτημα κρυφής μνήμης (Q_r)*

το μεγαλύτερο όφελος στον φόρτο εργασίας. Ο ακριβής υπολογισμός του οφέλους κάθε αποθηκευμένου αποτελέσματος $Q_i = (V_i, E_i)$ στην εκτέλεση ενός ερωτήματος $Q = (V, E)$ θα απαιτούσε την εκτέλεση του αλγορίθμου *DPccp* για κάθε ένα από τα αποτελέσματα. Για να αποφευχθεί αυτό, χρησιμοποιούμε την ακόλουθη ευριστική συνάρτηση εκτίμησης του οφέλους, που πολλαπλασιάζει το κόστος του ερωτήματος με το κλάσμα των ερωτημάτων τριάδας που καλύπτονται.

$$B(Q_i|Q) = \frac{|V_i|}{|V|} \cdot cost(Q) \tag{3.2}$$

Η εκτίμηση οφέλους απαιτεί το βέλτιστο υπολογισμό του κόστος του ερωτήματος Q και μπορεί να υπολογιστεί στο τέλος της διαδικασίας υπολογισμού του πλάνου εκτέλεσης. Ως εκ τούτου, κατά τη διάρκεια της διαδικασίας σχεδιασμού καταγράφουμε όλα τα μη ικανοποιημένα αιτήματα κρυφής μνήμης (Q_i) και στο τέλος δημιουργούμε έναν κατάλογο αιτημάτων, μαζί με τα αντίστοιχα οφέλη τους και τον στέλνουμε στον ελεγκτή κρυφής μνήμης για περαιτέρω επεξεργασία. Αυτό σημαίνει ότι το κόστος της απόδοσης οφέλους για όλα τα ερωτήματα που θα μπορούσαν ενδεχομένως να χρησιμοποιηθούν δεν επηρεάζει την εκτέλεση και το σχεδιασμό των ερωτημάτων. Η επεξεργασία των αιτημάτων κρυφής μνήμης γίνεται από το ξεχωριστό νήμα εκτέλεσης του ελεγκτή κρυφής μνήμης. Ο ελεγκτής διατηρεί μια δομή Cache Requests η οποία περιέχει εκτιμήσεις για το όφελος ερωτημάτων και χρησιμοποιεί σαν κλειδί τις κανονικοποιημένες ετικέτες σκελετού. Κάθε εγγραφή του πίνακα περιέχει μια δομή δέντρου εκτίμησης οφέλους που αναφέρεται σε ερωτήματα με την ίδια ετικέτα.

Το Σχήμα 3.13 απεικονίζει αυτό το δέντρο εκτίμησης οφέλους, που παράγεται από τη μέθοδο *addBenefit* (Αλγόριθμος 14), για το αίτημα κρυφής μνήμης (Q_r) του προηγούμενου κεφαλαίου με όφελος $B = 3sec$. Κάθε φύλλο του δέντρου αντιπροσωπεύει ένα μοτίβο ερωτήματος

Algorithm 13: $executeQuery$

```
1  function executeQuery(query)
2      (q, aux) ← abstractQuery(query);
3      //q: abstract query, aux: auxiliary query info
4      cacheRequests ← {};
5      plan ← DPccp(q, aux, cacheRequests);
6      results ← execute(plan); //RDF engine
7      //handled offline by the CacheController thread
8      CacheController.cacheResults(results);
9      CacheController.addRequestBenefits(cacheRequests, qID);
10     CacheController.addResultBenefit(q, plan);
11 function cacheResults(results)
12     //cache computed results
13     for result ∈ results do
14         //get the respective benefit from the cache requests
15         request ← CacheRequests.get(result);
16         result.benefit ← request.benefit;
17         result.queryIDs ← request.queryIDs;
18         cache(result, result.benefit);
19     end
20 function addRequestBenefits(cacheRequests, qID)
21     for (req, benefit) ∈ cacheRequests do
22         addBenefit(req.skeletonLabel, req.starLabel, req.aux, benefit, qID);
23     end
24 function addResultBenefit(q, plan)
25     //update the benefit of utilized cached results
26     for result ∈ plan.usedCachedResults do
27         newPlan ← DPccpWithoutResult(q, result);
28         result.benefit+ = (newPlan.cost − plan.cost);
29     end
```

που μπορεί ενδεχομένως να χρησιμοποιηθεί για την αίτηση της κρυφής μνήμης. Για να αποδώσουμε οφέλη σε όλα τα ενδεχομένως χρήσιμα αποτελέσματα, για όλα τα φίλτρα του ερωτήματος δημιουργούμε τουλάχιστον την ακμή *"*, Index"* και την ακμή που περιέχει το αντίστοιχο φίλτρο. Επιπλέον, εάν η εγγραφή του πίνακα Cache Requests περιέχει ήδη οφέλη για αποτελέσματα με φίλτρα που μπορούν να χρησιμοποιηθούν για το τρέχον αίτημα τα ακολουθούμε. Για παράδειγμα, αν ένα προηγούμενο ερώτημα είχε ένα φίλτρο $?univ = \text{“}H*\text{”}$ θα αποδίδαμε επίσης οφέλη στο υποδένδρο του. Όσον αφορά τις ακμές των ετικετών αστέρα, αν το υπάρχον δέντρο περιέχει ισομορφικά υποδέντρα αστέρα τα ακολουθούμε προκειμένου να αποδόσουμε τα αντίστοιχα οφέλη στα φύλλα τους. Οι τιμές εντός των κόμβων του δέντρου αντιπροσωπεύουν τις εκτιμήσεις εκλεκτικότητας για το αντίστοιχο μοτίβο ερωτήματος. Για να δημιουργήσουμε αυτές τις εκτιμήσεις χρειαζόμαστε μόνο τον εκτιμητή επιλεκτικότητας και τον συνολικό αριθμό των εγγραφών του αφηρημένου αποτελέσματος. Χρησιμοποιήσετε πάλι την ιδιότητα της ανεξαρτησίας για την εκτίμηση της επιλεκτικότητας πολλαπλών φίλτρων σε διαφορετικές

Algorithm 14: $addBenefit$

```
1  function addBenefit(skeletonLabel, starLabel, aux, benefit, qID)
2    //skeletonLabel, starLabel : the canonical labels of the abstract query graph
3    //aux : auxiliary query info(filters, projections, etc)
4    //benefit : the estimated benefit for the query
5    //qID : the query ID
6    aux.addStarLabel(starLabel);
7    benefitTree ← CacheRequests.get(skeletonLabel);
8    treeAddBenefit(benefitTree.root, aux, benefit, 1, qID);
9  function treeAddBenefit(node, aux, benefit, s, qID)
10   //s : parent node selectivity
11   if aux.isEmpty() then
12       addChild(benefit, s, qID);
13   else
14       aInfo ← aux.next();
15       if aInfo.isStarLabel() then
16           newEdgeIfNotExists(aInfo.getLabel());
17       else
18           addChild(benefit, s, qID);
19           newEdgeIfNotExists("*, Index");
20           newEdgeIfNotExists(aInfo);
21       end
22       for (edge, child) ∈ node.children() do
23           //check usability, selectivity of existing edges
24           selectivity ← s * selectivity(edge, aInfo);
25           //prune subtrees with benefit ≤ 0
26           if (benefit − selectivity * R/thr) > 0 then
27               treeAddBenefit(child, aux, benefit, selectivity);
28           end
29       end
30   end
```

μεταβλητές. Ως εκ τούτου, πρέπει απλά να μεταφέρουμε την εκτίμηση επιλεκτικότητας από τους γονικούς κόμβους προς τον κόμβο παιδί πολλαπλασιάζοντάς τη με την επιλεκτικότητα των αντίστοιχων ακμών. Το όφελος για κάθε χρησιμοποιήσιμο αποτέλεσμα, κόμβος φύλλο, είναι:

$$b = B - s \cdot R/thr \tag{3.3}$$

όπου το B είναι το συνολικό όφελος του αιτήματος, s είναι η επιλεκτικότητα του αποτελέσματος, R είναι ο αριθμός των εγγραφών του αφηρημένου ερωτήματος, και thr ταχύτητα ανάγνωσης εγγραφών (π.χ. 100k εγγραφές/sec). Το δεύτερο μέρος της εξίσωσης αντιπροσωπεύει το κόστος ανάγνωσης του αποτελέσματος. Στο Σχήμα 3.13, το όφελος για ένα αποτέλεσμα με επιλεκτικότητα s_3 είναι $b_3 = 3sec - s_3 * 1m/100k \simeq 3sec$. Μπορούμε επίσης να αγνοήσουμε τα οφέλη που είναι μικρότερα από 0. Για παράδειγμα, το μη ευρετηριασμένο αφηρημένο αποτέλεσμα, υψηλότερο φύλλο, έχει επιλεκτικότητα 1 και όφελος $b = 3sec - 1 * 1m/100k = -7sec$.

Algorithm 15: $CacheControllerPeriodicProcess$

```
1  // methods that run in configurable time or query intervals;
2  function updateBenefits()
3      //OrderedResults : benefit ordered list of cached results
4      for result ∈ ResultCache do
5          //decrease benefit with time using decay paramater 0 < a < 1
6          result.benefit = result.benefit * a;
7          OrderedResults.insert(result);
8      end
9      //OrderedRequests : benefit ordered list with max size
10     for request ∈ CacheRequests do
11         //decrease benefit with time
12         request.benefit = request.benefit * a;
13         OrderedRequests.insert(request);
14     end
15     //remove cache requests that are not in OrderedRequests
16     removeFromCacheRequests(OrderedRequests);
17     for request ∈ OrderedRequests do
18         //estimate cost for best requests using DPccp
19         request.benefit = request.benefit/estimateCost();
20     end
21 function profitableQueryGeneration()
22     //iterate in decreasing order of benefit/cost
23     for req ∈ OrderedRequests do
24         //proactively check cache replacement
25         evict ← evictions(estimateSize(req), req.benefit);
26         if evict.satisfied then
27             result ← executeQuery(req);
28             return;
29         end
30     end
```

Όταν τα οφέλη όλων των μοτίβων υπολογιστούν, ο ελεγκτής κρυφής μνήμης αθροίζει τις προϋπάρχουσες με τις νέες τιμές οφέλους και αποθηκεύει το αποτέλεσμα στο δέντρο του πίνακα Cache Requests. Ο ελεγκτής, τρέχοντας τον αλγόριθμο σχεδιασμού πλάνου εκτέλεσής μας, υπολογίζει το κόστος εκτέλεσης των πιο κερδοφόρων μοτίβων ερωτημάτων και διατηρεί μια διατεταγμένη λίστα ζευγών (request, benefit/cost) (Αλγόριθμος 15) που χρησιμοποιείται από τη μέθοδο *profitableQueryGeneration* για να προκαλέσει την εκτέλεση και προσωρινή αποθήκευση των πιο *κερδοφόρων* ερωτημάτων.

Algorithm 16: $cachingPolicy$

```
1  function cache(result, benefit)
2      evict ← evictions(result.size, benefit));
3      if evict.satisfied then
4          removeCachedResults(evict);
5          cacheResult(result.V, result.E, result.aux, result.size);
6          decreaseBenefits(result.queryIDs);
7          return;
8      end
9  function evictions(size, benefit)
10     //cache replacement policy
11     evict ← {};
12     if availableCacheSize ≥ size then
13         evict.satisfied = true;
14         return evict;
15     end
16     //iterate cached results in decreasing benefit order
17     for result ∈ OrderedResults do
18         if result.benefit ≤ benefit then
19             evict.add(result);
20             if evict.totalSize ≥ size then
21                 break;
22             end
23         end
24     end
25     if evict.totalSize ≥ size then
26         evict.satisfied = true;
27     else
28         evict.satisfied = false;
29     end
30     return evict;
31 function evictions(size, benefit)
32     for request ∈ CacheRequests do
33         oldSize ← request.queryIDs.size;
34         //remove common query IDs
35         request.queryIDs = request.queryIDs \ queryIDs;
36         newSize ← request.queryIDs.size;
37         request.benefit = request.benefit ∗ newSize/oldSize;
38     end
```

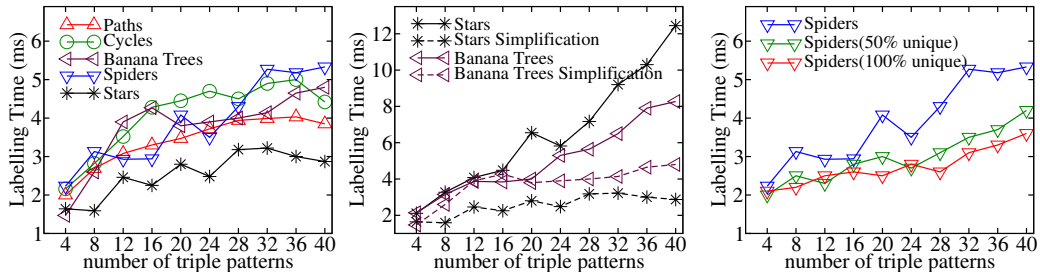
3.7 Πειράματα

Σε αυτή την ενότητα, παρουσιάζουμε μια λεπτομερή αξιολόγηση των επιδόσεων της προτεινόμενης υλοποίησης κρυφής μνήμης για SPARQL ερωτήματα. Επιλέγουμε να ενσωματώσουμε το σύστημά μας με την H_2RDF+ βάση δεδομένων RDF [Papailiou 13, Papailiou 14], που συνδυάζει το Hadoop και την HBase για την ευρετηρίαση και εκτέλεση ερωτημάτων SPARQL.

Το H_2RDF+ δημιουργεί ευρετήρια HBase για όλες τις αναδιατάξεις των RDF τριάδων, δηλαδή *spo, pso, pos, ops, osp* και *sop* [Weiss 08], καθώς και συγκεντρωτικά ευρετήρια στατιστικών που μπορούν να προσφέρουν ένα αναλυτικό μοντέλο κόστους συνενώσεων. Το σύστημα κρυφής μνήμης μας είναι ανεξάρτητο από τη μηχανή εκτέλεσης ερωτημάτων και έτσι μικρές αλλαγές απαιτούνται ώστε το H_2RDF+, καθώς και άλλες βάσεις δεδομένων RDF να το υποστηρίξουν. Η κρυφή μας μνήμη εξαρτάται από την ευρετηρίαση των προσωρινά αποθηκευμένων αποτελεσμάτων για: 1) την εκτέλεση ενεργειών φιλτραρίσματος, 2) την ενίσχυση των επιδόσεων εκτέλεσης συνενώσεων. Οι περισσότερες βάσεις δεδομένων RDF, συμπεριλαμβανομένου και του H_2RDF+, χρησιμοποιούν ταξινομημένα ευρετήρια που μπορούν να εκτελέσουν και τις δυο παραπάνω λειτουργίες. Δημιουργούμε ευρετήρια φορτώνοντας τα προσωρινά αποθηκευμένα αποτελέσματα σε πίνακες HBase, όμοιους με αυτούς των ευρετηρίων του H_2RDF+. Τα αποτελέσματα που δεν έχουν ευρετήρια αποθηκεύονται σε απλά αρχεία HDFS.

3.7.1 Δημιουργία κανονικοποιημένων ετικετών για SPARQL ερωτήματα

Στην ενότητα αυτή, εξετάζουμε την απόδοση του αλγορίμου δημιουργία κανονικοποιημένων SPARQL ετικετών. Αξιολογούμε τον προτεινόμενο αλγόριθμο κανονικοποίησης σκελετούαστέρα για διάφορους τύπους ερωτημάτων και τον συγκρίνουμε με τον βασικό αλγόριθμο κανονικοποιημένης επισήμανσης [Papailiou 15] που χρησιμοποιεί ολόκληρο το γράφο συνενώσεων του ερωτήματος. Η απόδοση της διαδικασίας επισήμανσης είναι στενά συνδεδεμένη με την απόδοση του *Bliss*, με ένα πρόσθετο κόστος για τον μετασχηματισμό του ερωτήματος.



Σχήμα 3.14: *Δημιουργία κανονικοποιημένων ετικετών για SPARQL ερωτήματα. Αξιολόγηση απόδοσης για διαφορετικά μοτίβα ερωτημάτων και μεγέθη ερωτήματος*

Αρχικά δημιουργούμε ένα σύνολο από μοτίβα SPARQL ερωτημάτων χρησιμοποιώντας μόνο ένα είδος ερωτήματος τριάδας π.χ., *?v1 ub:takesCourse ?v2*. Αυτό σημαίνει ότι όλες οι κορυφές του μετασχηματισμένου γράφου, που χρησιμοποιείται για την κανονικοποιημένη επισήμανση και απεικονίζεται στο Σχήμα 3.5, θα έχουν την ίδια ετικέτα. Τόσο ο *Bliss* όσο και ο

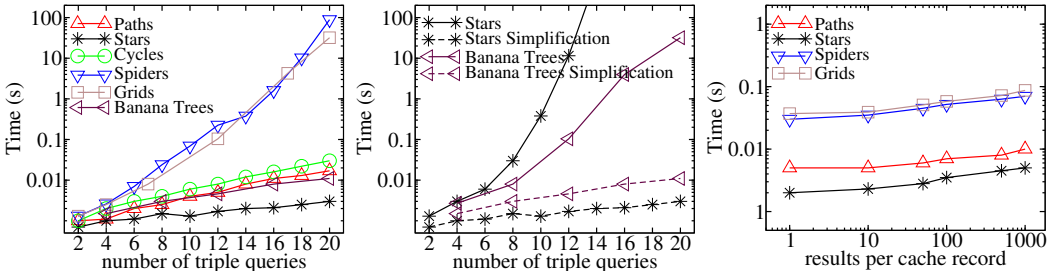
nauty χρησιμοποιούν τις πληροφορίες του χρώματος των κορυφών και τη δομή του γραφήματος για να σπάσουν τη συμμετρία του γράφου και να μειώσουν το χώρο αναζήτησης ισομορφισμών. Κατά συνέπεια, για τα ερωτήματα SPARQL που περιέχουν μόνο έναν είδος ερωτήματος τριάδας, ο αλγόριθμος κανονικοποιημένης επισήμανσης μπορεί να επωφεληθεί μόνο από τη δομή του γράφου τους. Οι χρόνοι απόκρισης που απαιτούνται για την επισήμανση ερωτημάτων με μεταβλητό αριθμό ερωτημάτων τριάδας απεικονίζονται στο πρώτο γράφημα του Σχήματος 3.14. Μεταβάλλοντας το μέγεθος του γράφου του ερωτήματος από 4 έως 40 κόμβους, παρατηρούμε ότι ο αλγόριθμος κανονικοποιημένης επισήμανσής μας μπορεί να δημιουργήσει ετικέτες για όλα τα ερώτημα σε λιγότερο από 6ms. Τα γραφήματα αράχνης και οι κύκλοι αποδεικνύονται οι πιο δύσκολες περιπτώσεις, διότι περιέχουν περισσότερους αυτομορφισμούς και δεν μπορούν να απλοποιηθούν χρησιμοποιώντας την τεχνική απλοποίησης σκελετού-αστέρων. Αντίθετα, τα ερωτήματα αστέρα απαιτούν την μικρότερη ποσότητα χρόνου επισήμανσης λόγω της τεχνικής απλοποίησής μας η οποία εκτελεί μόνο μία λειτουργία ταξινόμησης για την επισήμανσή τους. Το αποτέλεσμα της τεχνικής απλοποίησης σκελετού-αστέρα είναι πιο ορατό στο δεύτερο γράφημα του Σχήματος 3.14. Αυτό το γράφημα απεικονίζει το χρόνο επισήμανσης που απαιτείται για την επισήμανση του αρχικού γράφου συνενώσεων του ερωτήματος καθώς και του απλοποιημένου γράφου σκελετού-αστέρων. Τα μοτίβα γράφων αστέρα και μπανανιάς μπορούν να απλοποιηθούν αποτελεσματικά με χρήση της τεχνικής μας ς και άρα οι χρόνοι επισήμανσή τους μειώνονται αντίστοιχα. Στην περίπτωση των ερωτημάτων αστέρα, ο μετασχηματισμός τους σε κλίκες στο γράφο συνενώσεων απαιτεί εκθετικό χρόνο επισήμανσης χρησιμοποιώντας τον *Bliss*. Η τεχνική μας απλοποιεί ολόκληρο το γράφημα αστέρα και εκτελεί μόνο μια λειτουργία ταξινόμησης, χωρίς την εκτέλεση του *Bliss*. Σημειώνουμε ότι, για τις δύο κατηγορίες ερωτημάτων ο αλγόριθμός μας έχει σχεδόν σταθερή πολυπλοκότητα της τάξης των 2-4 ms.

Η τρίτη γραφική παράσταση του Σχήματος 3.14 απεικονίζει την επίδραση της ύπαρξης διαφορετικών τύπων ερωτημάτων τριάδας στα ερωτήματα. Τα διαφορετικά ερωτήματα τριάδας δίνουν στον αλγόριθμο κανονικοποιημένης επισήμανσης περισσότερες πληροφορίες για τη μείωση του χώρου αναζήτησης των ισομορφισμών. Απεικονίζουμε το χρόνο που απαιτείται για να δημιουργήσουμε ετικέτες για τους δύσκολους γράφους αράχνης μεταβάλλοντας τον αριθμό των κόμβων καθώς και το ποσοστό των διακριτών ερωτημάτων τριάδας. Εξετάζουμε ερωτήματα με 0%, 50% και 100% των ερωτημάτων τριάδας να είναι μοναδικά, ενώ τα υπόλοιπα είναι όμοια. Παρατηρούμε ότι ο απαιτούμενος χρόνος επισήμανσης μειώνεται όταν τα ερωτήματα περιέχουν περισσότερα μοναδικά ερωτήματα τριάδας που βοηθούν τον αλγόριθμο μας να σπάσει τη συμμετρία του γράφου. Τέλος, αυτό το κέρδος είναι εκθετικό, ιδιαίτερα στην περίπτωση των ερωτημάτων με πολλούς αυτομορφισμούς, και οδηγεί σε σχεδόν γραμμική πολυπλοκότητα επισήμανσης για ερωτήματα με 100% μοναδικά ερωτήματα τριάδας.

3.7.2 Βελτιστοποιητής Δυναμικού Προγραμματισμού

Στην ενότητα αυτή, αξιολογούμε τον βελτιστοποιητή δυναμικού προγραμματισμού μας, που επεκτείνει τον αλγόριθμο *DPccp* [Moerkotte 06] ενσωματώνοντας τον αλγόριθμο κανονικοποιημένης επισήμανσής μας και την αναζήτηση προσωρινά αποθηκευμένων αποτελεσμάτων στην κρυφή μνήμη. Η πρώτη γραφική παράσταση του Σχήματος 3.15 απεικονίζει το χρόνο που απαιτείται για το σχεδιασμό του πλάνου εκτέλεσης και τη δημιουργία κανονικοποιημένων ετικετών για όλους τους υπογράφους του ερωτήματος. Χρησιμοποιούμε μια σειρά από διαφορετικούς γράφους ερωτημάτων που αποτελείται από μονοπάτια, κύκλους, αστέρες, αράχνες, μπανανιές και πλέγματα και να μεταβάλουμε τον αριθμό των ερωτημάτων τριάδας τους. Η πολυπλοκότητα του βελτιστοποιητή μας εξαρτάται σε μεγάλο βαθμό από την ποσότητα των συνδεδεμένων υπογράφων ενός ερωτήματος. Τα μονοπάτια και οι κύκλοι δεν περιέχουν πολλούς υπογράφους και ως εκ τούτου ο βελτιστοποιητής μας είναι σε θέση να βρεί το πλάνο εκτέλεσής τους για ερωτήματα με έως και 20 σχέσεις σε λιγότερο από 30ms. Τα ερωτήματα αράχνης και τα ερώτημα πλέγματος, λόγω της πιο περίπλοκης δομής τους και του γεγονότος ότι δεν έχουν απλοποιηθεί από την τεχνική απλοποίησής μας, παρουσιάζουν υψηλότερη πολυπλοκότητα από τα μονοπάτια και τους κύκλους με αποτέλεσμα να απαιτούνται χρόνοι απόκρισης της τάξης του δευτερολέπτου για ερωτήματα με έως και 14 ερωτήματα τριάδας. Τέλος, τα ερωτήματα μορφής αστέρα και μπανανιάς είναι απλοποιούνται επαρκώς από την τεχνική μας και είμαστε σε θέση να βελτιστοποιήσουμε τέτοια ερωτήματα με έως και 20 σχέσεις σε λιγότερο από 20 ms. Η επίδραση της τεχνικής απλοποίησής μας απεικονίζεται στο δεύτερο γράφημα του Σχήματος 3.15. Αυτό το γράφημα παρουσιάζει τους χρόνους εκτέλεσης του βελτιστοποιητή μας χρησιμοποιώντας τον αρχικό καθώς και τον απλοποιημένο γράφο συνενώσεων. Σαφώς, οι μη απλοποιημένοι γράφοι συνενώσεων μορφή αστέρα απαιτούν εκθετικό χρόνο βελτιστοποίησης λόγω της μορφής κλίκας τους. Και στις δυο περιπτώσεις όμως, η τεχνική απλοποίησής μας καταφέρνει να προσφέρει εκθετική μείωση της πολυπλοκότητας βελτιστοποίησης του ερωτήματος. Σημειώνουμε ότι μπορούμε να χειριστούμε τα δύο ερωτήματα αστέρων και μπανανιάς με έως και 20 σχέδια σε λιγότερο από 10 ms, ενώ ο αλγόριθμος βελτιστοποίησης που τρέχει στον αρχικό γράφο συνενώσεων απαιτεί περισσότερα από 100 sec για ερωτήματα αστέρων με 14 σχέσεις και πάνω από 12 δευτερόλεπτα για μπανανιές με 20 σχέσεις.

Μια άλλη σημαντική παράμετρος που επηρεάζει την πολυπλοκότητα του σχεδιαστή μας είναι η αναζήτηση του δέντρου αποτελεσμάτων. Η διαδικασία αυτή εξαρτάται κυρίως από την ποσότητα των αποτελεσμάτων τα οποία είναι αποθηκευμένα μέσα σε μια εγγραφή του πίνακα της κρυφής μνήμης. Όπως εξηγείται στην ενότητα 3.5.1, μπορούμε να εκτελέσουμε μια top-k A* αναζήτηση για να βρούμε τα καλύτερα αποτελέσματα. Η τρίτη γραφική παράσταση του



Σχήμα 3.15: *Απόδοση βελτιστοποιητή πλάνου εκτέλεσης*

Σχήματος 3.15, απεικονίζει την επίδραση του ποσού των προσωρινά αποθηκευμένων αποτελεσμάτων στο χρόνο εκτέλεσης του βελτιστοποιητή, χρησιμοποιώντας μια top-3 αναζήτηση. Για να δοκιμαστεί η επίδραση του ποσού των προσωρινά αποθηκευμένων αποτελεσμάτων στην εκτέλεση του σχεδιαστή μας, επιλέγουμε μια σειρά ερωτημάτων που αποτελούνται από 10 ερωτήματα τριάδας και μεταβάλουμε τον αριθμό αριθμός των προσωρινά αποθηκευμένων αποτελεσμάτων ανά εγγραφή του πίνακα της κρυφής μνήμης. Δημιουργούμε τυχαία 1 έως 1000 αποτελέσματα για κάθε εγγραφή της κρυφής μνήμης. Τα πειραματικά μας αποτελέσματα δείχνουν ότι η απόδοση του σχεδιαστή μας κλιμακώνει λογαριθμικά με τον αριθμό των προσωρινά αποθηκευμένων αποτελεσμάτων για όλα τα είδη ερωτήματος. Αυτό σημαίνει ότι η A* αναζήτησή μας καταφέρνει να μειώσει αποτελεσματικά το χώρο αναζήτησης. Ο χρόνος που απαιτείται για το σχεδιασμό του ίδιου ερωτήματος υπό την παρουσία 1 και 1000 αποθηκευμένων αποτελεσμάτων αυξάνει μόνο κατά ένα παράγοντα 2 για όλα τα ερωτήματα. Συνοψίζοντας, ο σχεδιαστής δυναμικού προγραμματισμού μας είναι σε θέση να χρησιμοποιήσει αποτελεσματικά την κρυφή μνήμη και να δημιουργήσουν βέλτιστα σχέδια εκτέλεσης σε λιγότερο από ένα δευτερόλεπτο για σύνθετα ερωτήματα SPARQL. Αυτό περιλαμβάνει όλα τα ερωτήματα αναφοράς που χρησιμοποιούνται στις ακόλουθες ενότητες.

3.7.3 Απόδοση κρυφής μνήμης

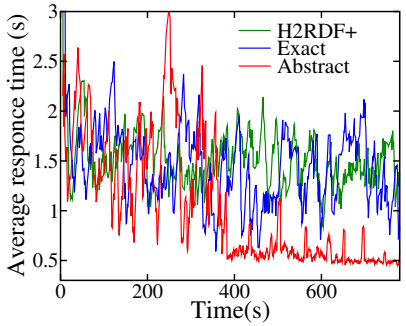
Σε αυτή την ενότητα αξιολογούμε την ικανότητα του ελεγκτή κρυφής μνήμης μας να βρεί και να αποθηκεύσει *κερδοφόρα* ερωτήματα για τη μείωση των χρόνων εκτέλεσης διαφόρων SPARQL συνόλων ερωτημάτων.

Συστοιχία πειραμάτων και σύνολα δεδομένων

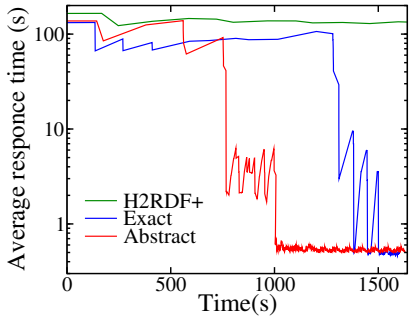
Η συστοιχία των πειραμάτων μας αποτελείται απο 10 κόμβους, κάθε ένα εκ των οποίων διαθέτει έναν 2 Quad-Core E5405 Intel Xeon®επεξεργαστή CPU στα 2.00GHz, 8 GB μνήμης RAM και 500GB σκληρό δίσκο. Κάθε κόμβος τρέχει 5 ταυτόχρονους mappers και 5 reducers,

κάθε ένας από τους οποίους καταναλώνει 512MB RAM. Στα πειράματά μας χρησιμοποιούμε το Hadoop v1.1.2 και την HBase v0.94.5.

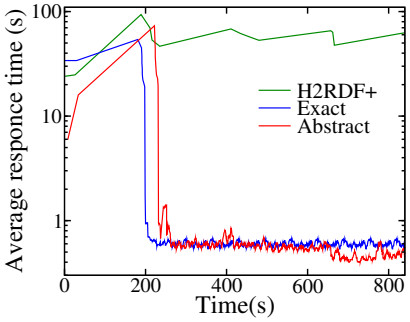
Χρησιμοποιούμε το τη γεννήτρια RDF δεδομένων LUBM [Guo 05] που δημιουργεί σύνολα δεδομένων με πληροφορίες ακαδημαϊκού τομέα, επιτρέποντας ένα μεταβλητό αριθμό RDF τριάδων ελέγχοντας τον αριθμό των πανεπιστημίων που θα περιέχονται στο σύνολο. Χρησιμοποιούμε δυο σύνολα δεδομένων στα πειράματά μας: το LUBM10k (10k universities, 1.38 billion triples και 250GB δεδομένων) και το LUBM20k (20k universities, 2.8 billion triples και 500GBδεδομένων).



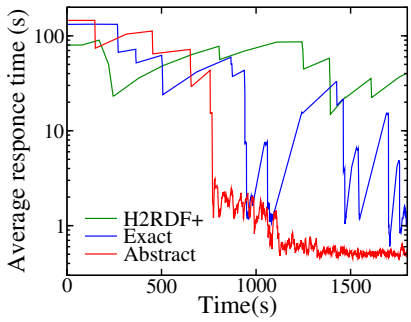
Σχήμα 3.16: *Μέσος χρόνος απόκρισης για το W1*



Σχήμα 3.17: *Μέσος χρόνος απόκρισης για το W2*



Σχήμα 3.18: *Μέσος χρόνος απόκρισης για το W3*



Σχήμα 3.19: *Μέσος χρόνος απόκρισης για το W4*

Απόδοση κρυφής μνήμης για φόρτους εργασίας SPARQL

Για να αξιολογήσουμε το σύστημα κρυφής μνήμης μας δημιουργούμε τα παρακάτω 4 σύνολα ερωτημάτων SPARQL χρησιμοποιώντας το LUBM:

Επιλεκτικά ερωτήματα (W1): Αυτό το σύνολο ερωτημάτων περιέχει ερωτήματα με επιλεκτικά μοτίβα όπως *?student ub:takesCourse <Course1>*. Αυτά τα επιλεκτικά μοτίβα μειώνουν το μέγεθος των δεδομένων που χρειάζεται να επεξεργαστούμε για την απάντησή τους. Αυτά τα ερωτήματα είναι:

LQ1: ?x rdf:type ub:GraduateStudent .
?x ub:takesCourse <GraduateCourse0>
LQ3: ?x rdf:type ub:Publication .
?x ub:publicationAuthor <AssistantProfessor0>
LQ4: ?x ub:worksFor <Department0.University0.edu> .
?x ub:name ?n . ?x ub:emailAddress ?em .
?x ub:telephone ?t . ?x rdf:type ub:Professor
LQ5: ?x rdf:type ub:GraduateStudent .
?x ub:memberOf <Department0.University0.edu>
LQ7: ?x rdf:type ub:Student . ?y rdf:type ub:Course .
?x ub:takesCourse ?y .
<AssociateProfessor0> ub:teacherOf ?y
LQ8: ?x rdf:type ub:Student . ?y rdf:type ub:Department .
?x ub:memberOf ?y . ?x ub:emailAddress ?em .
?y ub:subOrganizationOf <University0> .

Στον φόρτο εργασίας διαλέγουμε κάθε φορά με τυχαίο τρόπο τα επιλεκτικά department, university και course IDs.

Μη επιλεκτικά ερωτήματα (W2): Αυτό το σύνολο αποτελείται από μη επιλεκτικά ερωτήματα που έχουν μεγάλο μέγθος εισόδου ή σύνθετη δομή συνενώσεων. Τα ερωτήματα αυτής της κατηγορίας έχουν και μικρά και μεγάλα μεγέθη αποτελέσματος και είναι τα παρακάτω:

LQ2: ?z rdf:type ub:Department .
?x ub:memberOf ?z . ?x rdf:type ub:GraduateStudent .
?z ub:subOrganizationOf ?y . ?y rdf:type ub:University .
?x ub:undergraduateDegreeFrom ?y .
LQ9: ?x rdf:type ub:Student . ?y rdf:type ub:Professor .
?z rdf:type ub:Course . ?x ub:advisor ?y .
?y ub:teacherOf ?z . ?x ub:takesCourse ?z .
LQ15:?p rdf:type ?tp . ?p ub:worksFor ?d .
?s ub:takesCourse ?c . ?p ub:teacherOf ?c

Για παράδειγμα, το LQ2 έχει αρκετά μικρό αποτέλεσμα όμως αυτό προκύπτει μετά την εκτέλεση των συνενώσεων τριγωνικής δομής που απαιτούν μεγάλο μέγεθος εισόδου. Το LQ15 παράγει μεγάλο μέγεθος εξόδου γιατί συνδυάζει την πληροφορία όλων καθηγητών και των μαθημάτων. Για να κάνουμε αυτό το σύνολο ερωτημάτων πιο απαιτητικό διλέγουμε επίσης τυχαία

τους τύπους που χρησιμοποιούνται στα ερωτήματα τριάδας. Συνολικά, το σύνολο ερωτημάτων έχει 19 διαφορετικά SPARQL ερωτήματα.

Ερωτήματα κοινού υπογράφου (W3): Ένα δυνατό σημείο της τεχνικής κρυφής μνήμης μας είναι η δυνατότητα να ανακλύπτει cross-query συχνά εμφανιζόμενων υπογράφων και να τους αποθηκεύει γιατην βελτίωση των χρόνων εκτέλεσης των ερωτημάτων. Για να δείξουμε αυτή την περίπτωση δημιουργούμε ένα σύνολο ερωτημάτων που έχουν τον παρακάτω κοινό υπογράφο:
?x ub:memberOf ?z .
?z ub:subOrganizationOf ?y .
?x ub:undergraduateDegreeFrom ?y .
Κάθε ερώτημα αυτού του φόρτου εργασίας διαθέτει αυτόν τον υπογράφο σε συνδιασμό με άλλα τυχαία επιλεγμένα ερωτήματα τριάδας.

Γενικό σύνολο ερωτημάτων (W4): Αυτό το σύνολο ερωτημάτων περιέχει τα ερωτήματ όλων των προηγούμενων συνόλων. Αποτελείται από 14 διαφορετικούς τύπους ερωτημάτων και περιέχει επιλεκτικά, μη επιλεκτικά ερωτήματα και κοινούς υπογράφους μεταξύ των ερωτημάτων.

Τα Σχήματα 3.16-3.19 παρουσιάζουν το μέσο χρόνο απόκρισης των ερωτημάτων για τα 4 παραπάνω σύνολα ερωτημάτων. Για να αναδείξουμε τη συνεισφορά του συστήματος αφαίρεσης των σταθερών του ερωτήματος παρουσιάζουμε τα αποτελέσματα για: 1) τη βασική εκτέλεση στο H_2RDF+, 2) τη χρήση κρυφής μνήμης που δεν αφαιρεί τις σταθερές από τα ερωτήματα και 3) το πλήρως λειτουργικό σύστημα κρυφής μνήμης που παρουσιάσαμε. Σε όλες τις περιπτώσεις, χρησιμοποιούμε απεριόριστο μέγεθος τη κρυφής μνήμης.

ΚΕΦΑΛΑΙΟ 4

Ανάλυση δεδομένων κίνησης δικτύων

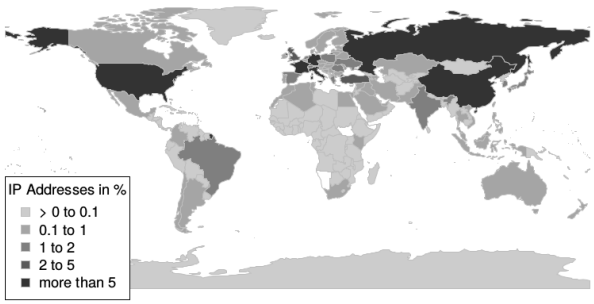
4.1 Εισαγωγή

Μια ακόμα πηγή μεγάλου όγκου μη δομημένων δεδομένων είναι και το Διαδίκτυο που έχει γίνει το κυρίαρχο κανάλι για την καινοτομία, το εμπόριο, και την ψυχαγωγία. Τόσο η κίνηση όσο και η διείσδυση του Διαδικτύου αυξάνεται με ρυθμό που καθιστά δύσκολο να παρακολουθείτε η ανάπτυξη και οι τάσεις του με έναν συστηματικό και επεκτάσιμο τρόπο. Πράγματι, πρόσφατες μελέτες δείχνουν ότι η κίνηση στο Διαδίκτυο αυξάνεται με ρυθμό μεγαλύτερο από 30% σε ετήσια βάση, όπως συμβαίνει και τα τελευταία 20 χρόνια. Αυτή η ανάπτυξη αναμένεται να συνεχιστεί με τον ίδιο ρυθμό και στο μέλλον [Cisco 13]. Ωστόσο, οι φορείς εκμετάλλευσης και οι διαχειριστές του αναγκάζονται να εκτελούν μεγάλης κλίμακας αναλύσεις πάνω στα δεδομένων του για τη βελτιστοποίηση παραμέτρων που αφορούν την δρομολόγηση, διαστασιολόγηση, και ασφάλεια του δικτύου.

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

Προφανώς, ένας IXP δρομολογεί κίνηση τάξης μεγέθους μεγαλύτερη σε σχέση με ένα συγκεκριμένο ISP. Μερικοί από τους πιο επιτυχημένους IXPs, συνδέουν περισσότερα από 600 δίκτυα και μεταφέρουν δεδομένα πολλαπλών TB ανά δευτερόλεπτο. Πιο συγκεκριμένα, κατά τη διάρκεια μιας μέσης εργάσιμης ημέρας του 2013, ένας από τους μεγαλύτερους IXPs, ο AMS-IX στο Άμστερνταμ, μετέφερε περίπου 25 PB ενώ η AT&T και η Deutsche Telekom μετέφεραν 33 PB και 16 PB δεδομένων αντίστοιχα [Chatzis 13a]. Πολλές σύγχρονες μελέτες έχουν δείξει ότι η δειγματοληπτική ανάλυση της κίνησης ενός IXP σε ένα βάθος χρόνου μερικών εβδομάδων μπορεί να εξάγει ενδιαφέροντα συμπεράσματα όχι μόνο για τα συγκεκριμένα AS που διασυνδέει, αλλά και για την κατάσταση ολόκληρου του διαδικτύου. Χρησιμοποιώντας το εργαλείο sFlow[1], οι προγούμενες μελέτες συγκέντρωσαν ένα δείγμα των πακέτων που δρομολογήθηκαν. Η ανάλυση των δειγμάτων απέδειξε ότι ένα IXP έχει τέλεια "ορατότητα" του συνολικού διαδικτύου, καθώς μέσα από αυτό περνάει κίνηση προς όλα σχεδόν τα υπάρχοντα AS και για όλα τα προθέματα δημόσιων δικτύων [Chatzis 13b].

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



Σχήμα 4.1: *Στατιστικά στοιχεία που προέρχονται από την παρακολούθηση της κίνησης ενός IXP*

Οι προηγούμενες μελέτες [Chatzis 13b] χρησιμοποίησαν παραδοσιακές κεντρικές τεχνικές επεξεργασίας των δεδομένων που συγκεντρώνονταν δειγματοληπτικά από τους IXPs. Είναι

[1] http://www.sflow.org/

προφανές ότι η ισχύς ενός μεμονωμένου υπολογιστή τοποθετεί ένα όριο στις δυνατότητες κλιμάκωσης της επεξεργασίας σε μεγαλύτερο αριθμό δεδομένων. Επίσης, το μέγιστο δυνατό μέγεθος των δεδομένων προς ανάλυση περιορίζεται από τις δυνατότητες του συγκεκριμένου υπολογιστή (συνήθως επηρεάζεται από το μέγεθος της φυσικής μνήμης του μηχανήματος).

Για να ξεπεραστούν τα προαναφερθέντα προβλήματα που σχετίζονται τόσο με την κλιμακωσιμότητα όσο και με την αποτελεσματική εκτέλεση ερωτημάτων, σχεδιάζουμε και υλοποιούμε το Datix: μια κλιμακώσιμη πλατφόρμα επεξεργασίας και ανάλυσης δεδομένων κίνησης δικτύων. Το σύστημά μας βασίζεται σε κατανεμημένες τεχνικές αποθήκευσης και επεξεργασίας δεδομένων, όπως το MapReduce [Dean 08] και είναι σε θέση να λύσει το πιο γενικό πρόβλημα της επεξεργασίας log αρχείων όπως περιγράφεται στο [Blanas 10]. Στόχος μας είναι η αποτελεσματική εκτέλεση κατανεμημένων συνενώσεων για το συνδυασμό μια κύριας πηγής πληροφορίας (log file), που στη δική μας περίπτωση είναι τα sFlow δεδομένα που συλλέγονται σε έναν IXP, με συμπληρωματικές πληροφορίες που παρέχονται από δευτερεύοντα σύνολα δεδομένων, όπως η χαρτογράφηση των διευθύνσεων IP, η αντιστοίχηση των IP με τα AS το DNS τους κτλ. [Durumeric 13]. Οι βασικές συνεισφορές αυτής της εργασίας είναι οι εξής:

- Παρουσιάζεται ένας έξυπνος τρόπος για pre-partitioning των δεδομένων ανάλογα με τις εγγραφές που περιέχουν, για να μπορεί να γίνει πιο αποδοτικά η επεξεργασία τους κατά τη διαδικασία της συνένωσης.

- Προτείνουμε μια αποδοτική υλοποίηση για τον αλγόριθμο συνένωσης της πληροφορίας, που κάνει χρήση της τεχνικής του map join [Blanas 10] και εξειδικευμένων συναρτήσεων UDF.

- Συνδυάζοντας τα δύο παραπάνω μπορούμε να εκτελέσουμε ειδικότερα ερωτήματα που αφορούν ένα περιορισμένο τμήμα των δεδομένων σε αρκετά μικρό χρονικό διάστημα έως μερικά λεπτά, όταν τα ερωτήματα που αφορούν όλο το dataset μπορεί να απαιτούν μερικές ώρες σε κάποιες περιπτώσεις.

4.2 Περιγραφή Συστήματος

Το Datix μπορεί να αξιοποιήσει υπολογιστικούς πόρους που προέρχονται από υπολογιστικά νέφη ή από ιδιωτικές συστοιχίες υπολογιστικών πόρων και εκτελεί ερωτήματα χρηστών πάνω σε δεδομένα που είναι αποθηκευμένα σε κατανεμημένα συστήματα αρχείων όπως το HDFS [Shvachko 10] ή η HBase [Chang 08]. Για την εκτέλεση κατανεμημένων SQL ερωτημάτων το Datix χρησιμοποιεί το Hive [Thusoo 09] ή το Shark [Zaharia 10]. Για την καλύτερη διαχείριση των δεδομένων, το Datix χωρίζει τα σύνολα δεδομένων σε δυο κατηγορίες. Η πρώτη κατηγορία είναι τα κεντρικά (*log*) σύνολα δεδομένων που στην δικιά μας περίπτωση είναι τα

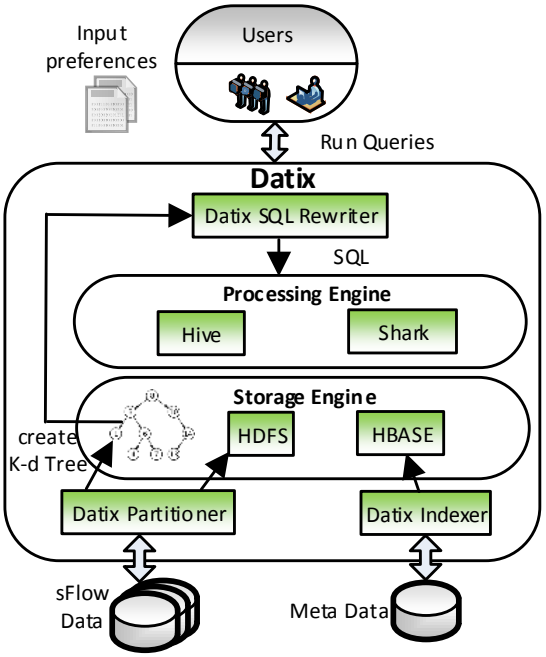
sflow αρχεία κίνησης του IXP. Αυτοί είναι οι κεντρικοί πίνακες που χρειάζεται να συνενωθούν με διάφορα επιμέρους σύνολα δεδομένων (*meta-datasets*) χρησιμοποιώντας ένα ή περισσότερα πεδία πληροφορίας. Στην περίπτωση των δεδομένων κίνησης δικτύων, όπως και στην γενική περίπτωση της επεξεργασίας log αρχείων [Blanas 10], το κεντρικό σύνολο δεδομένων αναμένεται να είναι τάξεις μεγέθους μεγαλύτερο από τα υπόλοιπα *meta-datasets*. Εκτός από το *log* σύνολο δεδομένων, το Datix υποστηρίζει την εισαγωγή πολλαπλών *meta-datasets*, τα οποία στην παρούσα μελέτη, είναι οι αντιστοιχίες IP με AS, IP με χώρα προέλευσης[2] και IP με DNS[3]. Έχουμε επιλέξει αυτά τα *meta-datasets* για την ανάλυσή μας επειδή είναι δημόσια διαθέσιμες πηγές πληροφορίας με διαφορετικά χαρακτηριστικά μεγέθους. Στη γενική περίπτωση, το Datix μπορεί να υποστηρίξει εισαγωγή αυθαίρετων *meta-datasets*, ανάλογα με τις προτιμήσεις του κάθε χρήστη.

Τα μεγέθη των *meta-datasets* μπορεί να μεταβάλλονται σημαντικά ανάλογα με τις πληροφορίες που παρέχουν. Για να χειριστούμε όλες τις περιπτώσεις *meta-datasets*, τα χωρίζουμε σε δύο κατηγορίες: (i) τα μικρού μεγέθους τα οποία είναι της τάξης των μερικών MB και μπορούν να χωρέσουν στην κύρια μνήμη ενός mapper (δηλαδή μιας ανεξάρτητης διεργασίας στην ορολογία του MapReduce) και (ii) τα μεγάλου μεγέθους τα οποία είναι μεγαλύτερα από το διαθέσιμο μέγεθος της κύριας μνήμης. Τα μικρού μεγέθους *meta-datasets* αποθηκεύονται σε αρχεία HDFS, ενώ τα μεγαλύτερα *meta-datasets* αποθηκεύονται και δεικτοδοτούνται με βάση τα πεδία συνένωσής τους, χρησιμοποιώντας πίνακες HBase.

Η συνένωση των δεδομένων κίνησης με τα διάφορα *meta-datasets* δίνει τη δυνατότητα εκτέλεσης καίριων ερωτημάτων λειτουργίας του δικτύου, όπως το ποιες είναι οι πιο δημοφιλείς IPs, πως επικοινωνούν τα AS μεταξύ τους, ποιες πόρτες εφαρμογών καταλαμβάνουν το μεγαλύτερο μέγεθος κίνησης. Επίσης σημαντικά ερωτήματα είναι η εκτίμηση του συνολικού μεγέθους κίνησης ανά AS ή IP, οι ερωτήσεις χρονικών παραθύρων, π.χ. σε ποια χώρα βρίσκονται οι διευθύνσεις IP που δραστηριοποιούνται σε μια χρονική περίοδο (κατά τη διάρκεια DDoS επιθέσεων) και είναι υπεύθυνες για περισσότερο από 1% της συνολικής κυκλοφορίας.

Το Σχήμα 4.2 απεικονίζει την αρχιτεκτονική του Datix. Ένα γραφικό εργαλείο βοηθά το χρήστη να δημιουργήσει το ερώτημα που θέλει να εκτελέσει. Στη συνέχεια, το Datix αναλαμβάνει να προσαρμόσει το ερώτημα του χρήστη (query rewriting) σε ένα ερώτημα που είναι συμβατό με τη HiveQL και μπορεί να εκτελεστεί αποτελεσματικά από το Hive ή το Shark (επάνω μέρος του Σχήματος 4.2). Στη συνέχεια τα ερωτήματα εκτελούνται με μια αλληλουχία κατανεμημένων εργασιών επεξεργασίας που λαμβάνουν είσοδο τα απαιτούμενα σύνολα δεδομένων του Datix τα οποία βρίσκονται αποθηκευμένα στο HDFS ή στην HBase (κάτω μέρος του Σχήματος 4.2). Πιο αναλυτικά το Datix αποτελείται από τα ακόλουθα τέσσερα στρώματα:

[2]http://dev.maxmind.com/geoip/legacy/geolite/
[3]http://scans.io/



Σχήμα 4.2: *Αρχιτεκτονική του Datix*

Datix Partitioner/Indexer: Αυτό το στρώμα είναι υπεύθυνο για το διαμοιρασμό (pre-partitioning) των συνόλων δεδομένων στους κόμβους της συστοιχίας σύμφωνα με τα πεδία διαμοιρασμού που έχει ορίσει ο χρήστης. Ο διαμοιρασμός των δεδομένων γίνεται με χρήση ενός K-d Tree [Louis 75] (Ενότητα 4.3.3). Τα πεδία διαμοιρασμού μπορεί να είναι πεδία συνένωσης (π.χ η IP πηγής και προορισμού των πακέτων) ή πεδία φιλτραρίσματος (π.χ η χρονοσφραγίδα του πακέτου, το πρωτόκολλο επικοινωνίας, η πόρτα της εφαρμογής κτλ.) Η λογική διαμοιρασμού μας στοχεύει στα παρακάτω:

- Τα μέτα-δεδομένα που απαιτούνται για την επεξεργασία ενός κομματιού του *log* αρχείου πρέπει να μπορούν να χωρέσουν στην κύρια μνήμη ενός mapper ώστε να είναι δυνατή η εκτέλεση αποδοτικών map-joins [Blanas 10].
- Τα μεγάλου μεγέθους *meta-datasets* πρέπει να είναι αποδοτικά δεικτοδοτημένα έτσι ώστε οι mappers να είναι σε θέση να ανακτήσουν τα απαραίτητα μέτα-δεδομένα με τη μικρότερη δυνατή επιβάρυνση.
- Εκτός από τον διαμοιρασμό με βάση τα πεδία συνένωσης, το *log* σύνολο δεδομένων πρέπει να διαμοιράζεται και με βάση πεδία φιλτραρίσματος που χρησιμοποιούνται για την αποδοτική εκτέλεση ερωτημάτων πάνω σε ένα υποσύνολο του *log* συνόλου δεδομένων.

Storage Engine: Αυτό το στρώμα του συστήματος είναι υπεύθυνο για την αποθήκευση και δεικτοδότηση όλων των απαραίτητων συνόλων δεδομένων. Κάθε partition του *log* αρχείου

αποθηκεύεται σε ένα ξεχωριστό HDFS αρχείο. Η πληροφορία του διαμοιρασμού αποθηκεύεται με χρήση ενός (K-d Tree) που επίσης αποθηκεύεται στο HDFS. Τα μικρού μεγέθους μέταδεδομένα αποθηκεύονται σαν ξεχωριστά αρχεία στο HDFS ενώ τα μεγάλου μεγέθους δεικτοδοτούνται με χρήση της Hbase.

Processing Engine: Αυτό το κομμάτι του συστήματος αναλαμβάνει την κατανεμημένη εκτέλεση SQL ερωτημάτων που χρησιμοποιούν ως είσοδο τα αντίστοιχα σύνολα δεδομένων του Datix. Συγκεκριμένα μπορούμε να χρησιμοποιήσουμε το Hive ή το Shark για την εκτέλεση των εργασιών που χρειάζονται για την επεξεργασία του ερωτήματος.

Datix SQL Rewriter: Το συγκεκριμένο κομμάτι του συστήματος είναι υπεύθυνο για τη μετάφραση του ερωτήματος του χρήστη σε ένα ερώτημα που μπορεί να εκτελεστεί αποδοτικά από το Hive και το Shark. Το Datix χρησιμοποιεί ειδικές συναρτήσει UDF ώστε να ενσωματώσει τον κωδικά του στην εκτέλεση των ερωτημάτων. Επίσης, παίρνει υπόψιν του την πληροφορία διαμοιρασμού των δεδομένων ώστε να εφαρμόσει τους περιορισμούς φιλτραρίσματος και συνένωσης. Έτσι καταφέρνει να μειώσει τα δεδομένα που χρειάζεται να επεξεργαστεί ένα ερώτημα (Ενότητα 4.3.3).

4.3 Αλγόριθμοι

Σε αυτή την ενότητα παρουσιάζουμε μια επισκόπηση των κατανεμημένων αλγορίθμων συνένωσης που χρησιμοποιούνται στο Datix. Προτείνουμε δύο νέους αλγορίθμους συνένωσης που διαχειρίζονται τις παρακάτω περιπτώσεις συνενώσεων: 1) όταν το σύνολο μέτα-δεδομένων είναι μικρό και χωράει στη μνήμη ενός map task (Ενότητες 4.3.1, 4.3.2), και 2) όταν το σύνολο μέτα-δεδομένων δεν χωράει στην κύρια μνήμη (Ενότητα 4.3.3). Αν και το Hive και το Shark υποστηρίζουν map-joins υπάρχουν κάποιες προϋποθέσεις που πρέπει να ισχύουν. Αρχικά, τα map-joins πρέπει να είναι συνενώσεις ισότητας (equi-joins) και δεύτερον ένας από τους δυο πίνακες πρέπει να χωράει στην κύρια μνήμη των mappers. Στις επόμενες ενότητες περιγράφουμε πως το Datix καταφέρνει και ξεπερνάει τους παραπάνω περιορισμούς.

4.3.1 Map συνενώσεις ισότητας (equi-joins)

Πρόβλημα: Δεδομένου δυο πινάκων, ενός μεγάλου πίνακα L και ενός μικρού πίνακα S, θέλουμε να εκτελέσουμε μια συνένωση ισότητας (equi-join) $L \bowtie_{L.c=S.c} S$ με βάση ένα συγκεκριμένο πεδίο (column) c των δυο πινάκων. Επίσης ισχύει ότι $|S| \ll |L|$, ώστε το S να χωράει στη μνήμη ενός mapper task. Αυτή η υπόθεση ισχύει για την περίπτωση των IP σε AS και IP σε χώρα αντιστοιχίσεων, οι οποίες καταλαμβάνουν χώρο λιγότερο από 12 MB.

Αντί να χρησιμοποιήσουμε το απλό αλγόριθμο του suffle join του Hive οποίος απαιτέι πολύ disk I/O και μεταφορά δεδομένων, χρησιμοποιούμε τον αλγόριθμο συνένωσης map-join

[Tang 11] και πιο συγκεκριμένα το Broadcast Join που περιγράφεται στο [Blanas 10]. Ο αλγόριθμος map equi-join εκτελεί τη συνένωση ανεξάρτητα σε κάθε map task χωρίς επικοινωνία. Κάθε κόμβος της συστοιχίας ανακτά όλο τον πίνακα S από το HDFS και τον αποθηκεύει στην κύρια μνήμη του. Στη συνέχεια κάθε διεργασία χρησιμοποιεί ένα hash-table για τη συνένωση ενός κομματιού του L με τις αντίστοιχες εγγραφές του S.

Στην αρχή κάθε map διεργασίας ο πίνακας S διαβάζεται από το HDFS και φορτώνεται σε ένα hash-table στην κύρια μνήμη. Στη συνέχεια για κάθε εγγραφή του πίνακα L η map συνάρτηση βρίσκει το πεδίο συνένωσης και ψάχνει το hash-table για τις αντίστοιχες εγγραφές του S. Παρατηρούμε ότι αυτός ο αλγόριθμος μεταφέρει μόνο τον πίνακα S σε όλους τους κόμβους του cluster και άρα αποφεύγει την ακριβή αναδιάταξη και μεταφορά των δεδομένων του L. Ωστόσο ένα πιθανό μειονέκτημα είναι ότι ο πίνακας S φορτώνεται πολλές φορές από κάθε διεργασία map. Αυτό μπορεί να αποφευχθεί με το να φορτώνουμε τον μικρό πίνακα μόνο μια φορά για κάθε φυσικό κόμβο του cluster. Οι ξεχωριστές διεργασίες του ίδιου κόμβου μπορούν να έχουν πρόσβαση στον πίνακα S μέσω μιας δομής μοιραζόμενης κύριας μνήμης.

4.3.2 Map συνενώσεις ανισότητας (theta-join)

Πρόβλημα: Παρόμοια με την Ενότητα 4.3.1 εξετάζουμε την περίπτωση στην οποία έχουμε δυο πίνακες L και S με την διαφορά ότι τώρα ο στόχος είναι η εκτέλεση μιας συνένωσης ανισότητας (theta-join) με βάση ένα συγκεκριμένο πεδίο. Άρα θέλουμε να εκτελέσουμε τη συνένωση $L \bowtie_{L.c \geq S.c_1 \wedge L.c \leq S.c_2} S$. Η λογική είναι ότι τα αρχεία που περιέχουν τις αντιστοιχίες IP με AS και IP με χώρα προέλευσης περιέχουν εύρη διευθύνσεων και όχι εγγραφές για κάθε μοναδική IP. Άρα απαιτείται η εκτέλεση συνενώσεων ανισότητας για την ανάκτηση των επιθυμητών αποτελεσμάτων.

Για την εκτέλεση συνενώσεων ανισότητας, η βασική διαφορά στην περίπτωση του map-join είναι ότι πρέπει να χρησιμοποιήσουμε αντί για ένα hash-table μια δομή δεδομένων που μπορεί να απαντήσει ερωτήματα εύρους. Μια τέτοια δομή είναι ένα δέντρο διάταξης, TreeMap. Η μεθοδολογία σε αυτή την περίπτωση αποτελείται από τα παρακάτω βήματα. Μεταφέρουμε τον πίνακα S σε όλους τους κόμβους και το φορτώνουμε σε μια δομή TreeMap στην κύρια μνήμη. Για να παράξουμε τα αποτελέσματα της συνένωσης, για κάθε εγγραφή του L βρίσκουμε το πεδίο συνένωσης και εκτελούμε μια ερώτηση εύρους στο TreeMap. Ενσωματώνουμε αυτές τις λειτουργίες τόσο στο Hive όσο και στο Shark με χρήση UDFs (user-defined functions).

4.3.3 Map συνενώσεις ισότητας (equi-joins) με μεγάλο σύνολο μέτα-δεδομένων

Πρόβλημα: Ο ορισμός αυτού του προβλήματος είναι παρόμοιος με αυτόν της Ενότητας 4.3.1 με τη διαφορά ότι τώρα ο πίνακας S είναι αρκετά μεγάλος ώστε να χωρέσει πλήρως στην κύρια μνήμη. Ένα τέτοιο παράδειγμα είναι η αντιστοιχία μεταξύ διευθύνσεων IP και DNS ονομάτων

το συνολικό μέγεθος της οποίας είναι γύρω στα 57 GB. Σε αυτή την περίπτωση πρέπει να ακολουθήσουμε μια διαφορετική στρατηγική για την υλοποίηση του map-join ώστε να αποφύγουμε τις περιττές μεταφορές δεδομένων. Συγκεκριμένα, η βασική ιδέα είναι ότι κάθε κομμάτι των sflow εγγραφών περιέχει περιορισμένο αριθμό μοναδικών IPs και άρα δεν χρειάζεται να μεταφέρουμε όλα το αρχείο αντιστοιχίας στη μνήμη των mappers. Αρκεί να μεταφέρουμε την πληροφορία που αναφέρεται σε αυτές τις μοναδικές διευθύνσεις (semi-join [Blanas 10]). Η δική μας συνεισφορά είναι ότι τα sflow δεδομένα διαμοιράζονται και αποθηκεύονται με τέτοιο τρόπο ώστε για κάθε partition να απαιτείται ένα εύρος των μέτα-δεδομένων. Άρα χρησιμοποιώντας τα διατεταγμένα ευρετήρια της HBase μπορούμε να ανακτήσουμε αποδοτικά τις αντίστοιχες εγγραφές.

Ο διαμοιρασμός των δεδομένων μπορεί να επιτευχθεί με διάφορες τεχνικές αλλά εμείς επικεντρωνόμαστε στις παρακάτω 2 μεθόδους:

Μέθοδος 1: Στατικός διαμοιρασμός Η πρώτη προσέγγιση για το διαμοιρασμό των δεδομένων είναι να χρησιμοποιήσουμε μια ομοιόμορφη κατάτμηση του χώρου των πεδίων συνένωσης (διευθύνσεις IP του *log* συνόλου δεδομένων). Τα μέτα-δεδομένα που απαιτούνται για κάθε κομμάτι του *log* (sFlow) αρχείου πρέπει να χωράνε στην κεντρική μνήμη ενός map task. Στην περίπτωσή μας, μπορούμε να χωρίσουμε το σύνολο δεδομένων IP-DNS σε κομμάτια που χωράνε στη μνήμη και να χρησιμοποιήσουμε τα επιλεχθέντα εύρη διευθύνσεων για το διαμοιρασμό των sflow δεδομένων. Εδώ πρέπει να λάβουμε υπόψιν μας ότι κάθε sflow εγγραφή περιέχει δυο IP διευθύνσεις τόσο τη διεύθυνση πηγής όσο και τη διεύθυνση προορισμού. Άρα χρειαζόμαστε έναν δισδιάστατο διαμοιρασμό. Η συγκεκριμένη υλοποίηση έχει δυο στάδια:

(i) Διαμοιρασμός: Μια εργασία MapReduce είναι υπεύθυνη για το διαμοιρασμό των δεδομένων sflow αφού έχουν βρεθεί στατικά τα εύρη διευθύνσεων από το σύνολο μέτα-δεδομένων. Αυτή η εργασία παράγει ένα ξεχωριστό αρχείο για κάθε partition σε συνδυασμό με ένα αρχείο που περιέχει τις μοναδικές IPs που περιέχονται σε κάθε partition. Ο διαμοιρασμός των δεδομένων γίνεται κατά τη διάρκεια εισαγωγής των δεδομένων στο σύστημα και πριν την εκτέλεση των ερωτημάτων.

(ii) Επερώτηση: Το δεύτερο βήμα είναι η ενσωμάτωση της πληροφορίας του διαμοιρασμού στην εκτέλεση των map-joins. Αυτό επιτυγχάνεται μέσω μιας συνάρτησης UDF η οποία ανακτά από την HBase μόνο τα αντίστοιχα εύρη διευθύνσεων IP και αφού τα φιλτράρει με βάση της μοναδικές IP του κάθε partition τα φορτώνει στην μνήμη του mapper.

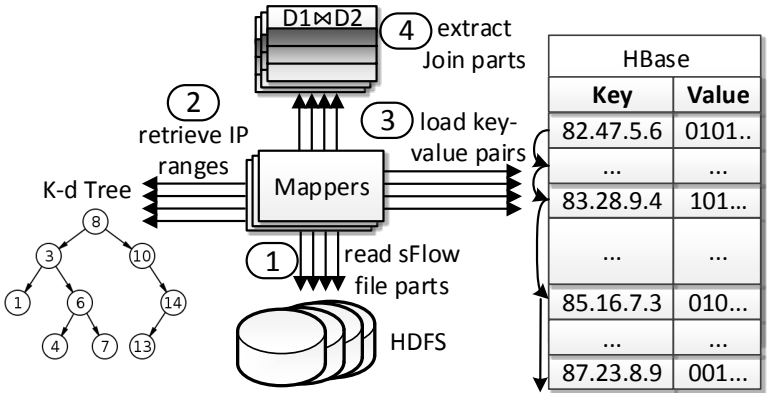
Η συγκεκριμένη τεχνική διαμοιρασμού είναι αρκετά απλή αλλά αποτυγχάνει να δημιουργήσει ομοιόμορφα μεγέθη κομματιών (partitions). Αυτό συμβαίνει γιατί λαμβάνει υπόψιν της μόνο το σύνολο των μέτα-δεδομένων και όχι την κατανομή των διευθύνσεων IP στις εγγραφές sflow. Έτσι περιοχές ζευγαριών διευθύνσεων IP που ανταλλάσουν μεγάλη κίνηση δεδομένων

θα δημιουργούν αντιστοίχως μεγάλα σε μέγεθος partition αρχεία. Επιπλέον, λόγω του ομοιόμορφου διαμοιρασμού μόνο μερικά partitions έχουν στην πραγματικότητα δεδομένα αφήνοντας πολλά από τα partition άδεια και οδηγώντας σε χαμηλή απόδοση.

Μέθοδος 2: Δυναμικός διαμοιρασμός. Η δεύτερη τεχνική διαμοιρασμού ξεπερνάει τα προβλήματα της προηγούμενης μεθόδου χρησιμοποιώντας μια δομή K-dimensional Tree (K-d Tree) [Louis 75] για τον διαμοιρασμό των δεδομένων. Αυτή η δομή ταιριάζει για το διαμοιρασμού ενός χώρου δεδομένων γιατί επιτυγχάνει τα παρακάτω:

- Διατηρεί την διάταξη των δεδομένων και δεν αφήνει κενά στο χώρο διαμοιρασμού.
- Εξασφαλίζει ότι όλα τα partition αρχεία θα περιέχουν έναν ομοιόμορφα κατανεμημένο αριθμό εγγραφών .
- Επιτρέπει τον διαμοιρασμό ενός συνόλου δεδομένων χρησιμοποιώντας πολλαπλές διαστάσεις.

Το τελευταίο χαρακτηριστικό του K-d Tree οδηγεί σε αποτελεσματική εκτέλεση φιλτραρίσματος πολλαπλών διαστάσεων. Αν και το K-d Tree δεν είναι αρκετά αποδοτικό σε μεγάλο αριθμό διαστάσεων, είναι κατάλληλο για την περίπτωσή μας αφού οι βασικές διαστάσεις που χρησιμοποιούνται για συνενώσεις και φιλτράρισμα στα sflow δεδομένα δεν ξεπερνούν τις 10. Αυτή η τεχνική διαμοιρασμού εκτελείται σε 3 στάδια:



Σχήμα 4.3: *Εκτέλεση συνενώσεων με χρήση K-d Tree.*

(i) Δειγματοληψία: Αρχικά δημιουργούμε ένα μικρό δείγμα των sflow δεδομένων (περίπου 1%) και το χρησιμοποιούμε για τη δημιουργία του K-d Tree. Ρυθμίζουμε το μέγιστο μέγεθος εγγραφών που περιέχονται σε ένα φύλλο του K-d Tree m με βάση το ποσοστό δειγματοληψίας καθώς και τους περιορισμούς κύριας μνήμης των mappers της συστοιχίας μας. Όλες οι εγγραφές του δείγματος εισάγονται κεντρικά σε μια δομή K-d Tree. Στο τέλος αυτής της διαδικασίας τα σημεία διαχωρισμού του K-d Tree χρησιμοποιούνται για τον διαμοιρασμό όλων των sflow δεδομένων.

Algorithm 17: Εκτέλεση συνενώσεων με χρήση K-d Tree διαμοιρασμού

```
1: Function evaluate()
2: if DnsMap == NULL then
3:    kd = readTreePartitionFile()
4:    kd.findBuckets(min, max, l)
5:    l.sort()
6:    partNum = l.indexOf(partNum)
7:    line = readLineFrom(uniqueIPFile)
8:    s = HBaseTable.getScanner(scan.setStartRow(line))
9:    result = s.next()
10:   while line! = NULL&&result! = NULL do
11:      if UniqueIP>ScanIP then
12:         result = s.seekTo(line, UniqueIP − ScanIP)
13:      else
14:         line = br.seekTo(result.getRowKey())
15:      end if
16:      if result.getRowKey().equals(line) then
17:         keyValue.putInHashMap()
18:      end if
19:   end while
20: end if
21: return DnsMap.get(ip)
```

(ii) Διαμοιρασμός: Το δεύτερο στάδιο είναι η χρήση της πληροφορίας του k-D Tree για τον διαμοιρασμό όλων των sflow εγγραφών. Αυτό επιτυγχάνεται με μια ξεχωριστή εργασία MapReduce που δημιουργεί ένα αρχείο που περιέχει τις εγγραφές που ανήκουν σε έναν υπερκύβο διαμοιρασμού του K-d Tree, σε συνδυασμό με ένα αρχείο που περιέχει τις μοναδικές IPs που περιέχονται σε κάθε partition.

(iii) Επερώτηση: Το τελευταίο βήμα είναι η δημιουργία μις συνάρτησης UDF που θα ενσωματώνει την παραπάνω λειτουργικότητα στα ερωτήματα. Το Σχήμα 4.3 απεικονίζει όλη τη διαδικασία που ακολουθείται. Η συνάρτηση παίρνει ως είσοδο το εύρος τιμών ενός συγκεκριμένου partition καθώς και το αρχείο που περιέχει τις μοναδικές IPs του. Χρησιμοποιεί αυτή την πληροφορία για να ανακτήσει αποδοτικά τα αντίστοιχα δεδομένα από την HBase με χρήση ενός range scan. Στην πραγματικότητα εκτελεί ένα merge-join μεταξύ των μοναδικών τιμών IP του αρχείου και του *meta-dataset*. Όταν ανακτηθούν τα απαραίτητα μέτα-δεδομένα μια συνένωση ισότητας εκτελείται παρόμοια με αυτή της ενότητας 4.3.1 (Αλγόριθμος 17).

4.4 Πειράματα

Στο κεφάλαιο αυτό θα παρουσιάσουμε την επίδραση διάφορων παραμέτρων στην απόδοση εκτέλεσης των ερωτημάτων. Πρώτα, εξετάζουμε τη διαφορά μεταξύ δύο συστημάτων εκτέλεσης MapReduce εργασιών, του Hive και του Spark αναλύοντας τα πλεονεκτήματα του

δεύτερου σε κάποιες περιπτώσεις. Στη συνέχεια, πειραματιζόμαστε με την κλιμάκωση των ερωτημάτων όσον αφορά τον αριθμό των κόμβων του cluster και το μέγεθος των συνολικών δεδομένων. Τέλος, δείχνουμε την επίδραση του αριθμού των διαστάσεων στο χρόνο εκτέλεσης των ερωτημάτων που αφορούν τα DNS ονόματα.

4.4.1 Περιγραφή του Dataset

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

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

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

Το τελικό μέγεθος των δεδομένων μετά από την προεπεξεργασία αυτή είναι περίπου 50 GB σε συμπιεσμένη μορφή. Σκοπός της ανάλυσής μας είναι να συνδυάσουμε την πληροφορία αυτή με επιπλέον σύνολα δεδομένων που περιέχουν meta- πληροφορίες όπως την κατάταξη της κάθε IP στο αυτόνομο σύστημα (AS) που ανήκει ή την αντιστοίχισή της με τη χώρα προέλευσης. Η πληροφορία αυτή συλλέχτηκε από τη βάση δεδομένων GeoLite, μία geolocation βάση δεδομένων που χρησιμεύει για την αντιστοίχιση των διευθύνσεων IP στο αυτόνομο σύστημα που ανήκουν και στη χώρα προέλευσης τους. Τα μεγέθη αυτών των αρχείων είναι 12 MB για το αρχείο που περιέχει τα αυτόνομα συστήματα και 7 MB για το αρχείο με τις χώρες. Τα δεδομένα αυτά ανανεώνονται στις αρχές κάθε μήνα, οπότε μπορούμε να χρησιμοποιούμε

διαφορετικά meta-data αναλόγως της ημερομηνίας που προέρχονται τα αρχικά δεδομένα μας. Τα αρχεία αυτά περιέχουν τα δεδομένα στη μορφή:

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

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

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

$$IP,\ DNSName$$

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

4.4.2 Περιγραφή του cluster

Για την εκτέλεση των πειραμάτων δημιουργήσαμε ένα cluster από υπολογιστές χρησιμοποιώντας την υποδομή που προσφέρει ο ∼okeanos [Koukis 13], ένα έργο που παρέχει υπηρεσίες Infrastructure as a Service (IaaS) στην Ελληνική ερευνητική και ακαδημαϊκή κοινότητα, που δημιουργήθηκε και συνεχίζει να αναπτύσσεται από το Ελληνικό Δίκτυο Έρευνας και Τεχνολογίας (ΕΔΕΤ). Αυτή η πλατφόρμα παρέχει στο χρήστη τη δυνατότητα για υπολογιστική ισχύ (μέσω εικονικών μηχανών), δικτύωσης και αποθήκευσης δεδομένων. Το cluster που δημιουργήσαμε αποτελείται από 1 master κόμβο και 14 slaves με τις εξής δυνατότητες:

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

Πίνακας 4.1: *Χαρακτηριστικά κόμβων του cluster*

Στο master κόμβο τρέχουν οι master διεργασίες όλων των tools και frameworks που χρησιμοποιούμε, δηλαδή ο JobTracker και ο NameNode του Hadoop και οι master της HBase και του

Spark. Επίσης, σε αυτό τον κόμβο είναι εγκατεστημένο το Hive και το Shark. Αντίστοιχα, στους slaves τρέχουν οι υπόλοιπες διεργασίες που αφορούν τα εργαλεία αυτά, όπως ο Datanode, ο TaskTracker, ο RegionServer και ο Worker.

4.4.3 Σύγκριση Συστημάτων

Σε αυτή την ενότητα συγκρίνουμε την απόδοση του Datix όταν εκτελεί τα ίδια ερωτήματα χρησιμοποιώντας το Hive ή το Shark. Αξιολογούμε την απόδοση αυτών των δύο συστημάτων με βάση το χρόνο εκτέλεσης του ερωτήματος και σχολιάζουμε σχετικά με τα πλεονεκτήματα και τα μειονεκτήματα του κάθε συστήματος.

Dataset Partitioning: Ο Πίνακας 4.2 δείχνει την επιβάρυνση του διαμοιρασμού των δεδομένων σε μεταβλητού μεγέθους σύνολα δεδομένων. Τόσο η δειγματοληψία όσο και ο διαμοιρασμός των δεδομένων έχουν προστεθεί στα αποτελέσματα. Όπως μπορούμε να παρατηρήσουμε, ο χρόνος εισαγωγής των δεδομένων κλιμακώνει γραμμικά σε σχέση με το μέγεθος των δεδομένων, ειδικότερα για μικρά μεγέθη δεδομένων. Για μεγαλύτερα σύνολα δεδομένων υπάρχει μια μικρή επιβάρυνση που οφείλεται κυρίως στην δρομολόγηση των map και reduce διεργασιών. Επιπλέον, ο διαμοιρασμός των δεδομένων απαιτεί περίπου 4 φορές περισσότερο χρόνο από την απλή ανάγνωση των δεδομένων.

Μέγεθος Dataset (%)	20%	40%	60%	80%	100%
Χρόνος εισαγωγής (min)	24	50	78	102	150

Πίνακας 4.2: *Χρόνος εισαγωγής δεδομένων σε σχέση με το μέγεθός τους*

Query Type	Python	Hive	Datix-Hive		Datix-Shark	
			2D	3D	2D	3D
topAS	58min	170min	50min	52min	15min	16min
topDNS	>24h	135min	35min	76min	30min	64min
topDNS 1 week	>24h	116min	30min	9min	28min	8min

Πίνακας 4.3: *Σύγκριση απλών συνενώσεων με το Datix*

Ο Πίνακας 4.3 περιέχει τους χρόνους εκτέλεσης δυο ειδών ερωτημάτων χρησιμοποιώντας μια κεντρική υλοποίηση σε Python, τις υλοποιήσεις συνενώσεων των Hive και Shark καθώς και τις υλοποιήσεις του Datix. Τα συγκεκριμένα ερωτήματα υπολογίζουν τα top-k AS ή DNS ζευγάρια, τα οποία ανταλλάσουν το μεγαλύτερο μέγεθος δικτυακής κίνησης. Αυτά τα ερωτήματα,

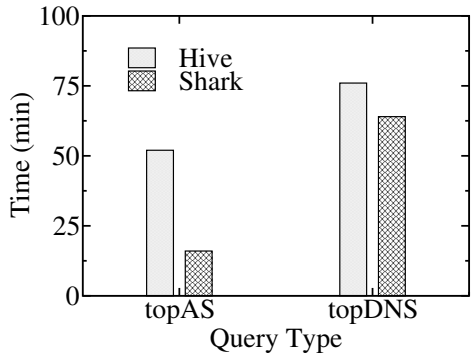
αν και είναι σχετικά απλά, καλύπτουν τις βασικές SQL λειτουργίες και μπορούν να χρησιμοποιηθούν για την εκτέλεση πιο σύνθετων ερωτημάτων. Συγκεκριμένα, τα ερωτήματα χρησιμοποιούν: συνενώσεις, ομαδοποίηση των δεδομένων, αθροιστικές συναρτήσει καθώς και λειτουργίες διάταξης. Συνδυάζονται μερικές ή όλες τις παραπάνω λειτουργίες πιο σύνθετα ερωτήματα μπορούν να δημιουργηθούν. Για παράδειγμα, ένα χρήσιμο ερώτημα θα ήτνα ο υπολογισμός της ημερήσιας κίνησης ενώς συγκεκριμένου εξυπηρετητή. Σε όλα τα πειράματα χρησιμοποιείται τριών διαστάσεων διαμοιρασμός των δεδομένων (source IP, destination IP, timestamp), εκτός και αν αναφέρεται διαφορετικά.

Η Python υλοποίησή μας τρέχει σε ένα μηχάνημα με επεξεργαστή Core i7-4820K CPU που διαθέτει 8 threads, 48GB RAM και 8TB δίσκο. Για τα ερωτήματα topAS, στα οποία το *meta-dataset* είναι μικρού μεγέθους και χωράει στη μνήμη των mappers, η Python υλοποίηση είναι 4 φορές πιο αργή από την εκτέλεση του Datix με χρήση του Shark ενώ ο χρόνος εκτέλεσής της είναι σχεδόν ίδιος με αυτόν του Hive. Υπάρχουν διάφοροι λόγοι που συμβαίνει αυτό. Πρώτα, το μηχάνημα που εκτελεί τον Python κώδικα έχει σαφώς καλύτερο επεξεργαστή και απόδοση από τα ανεξάρτητα VMs που χρησιμοποιούνται για την εκτέλεση του Hive. Δεύτερον το κόστος επικοινωνίας για τη μεταφορά των μέτα-δεδομένων επηρεάζει την εκτέλεση των ερωτημάτων. Διαπιστώνουμε λοιπόν ότι όταν τα δεδομένα χωράνε στη μνήμη τότε η κεντρική υλοποίηση είναι αρκετά ανταγωνιστική σε σχέση με τις αντίστοιχες κατανεμημένες.

Αντίθετα, το *meta-dataset* για την αντιστοιχία IP-DNS δεν χωράει στην κύρια μνήμη και άρα δεν είναι δυνατή η εκτέλεσή του με χρήση κάποιου hash-map. Μια συχνά χρησιμοποιούμενη λογική είναι η αποθήκευσή του σε μια βάση δεδομένων (π.χ. MySQL) και η επερώτησή της για κάθε sflow εγγραφή. Αυτή η τεχνική προσθέτει πολλαπλές επιβαρύνσεις στην εκτέλεση του ερωτήματος και απαιτεί μέχρι και μια ημέρα για την εκτέλεση του σε δεδομένα που αντιστοιχούν μόνο σε μια εβδομάδα. Σε αντίθεση, το Datix καταφέρνει να εκτελέσει τα συγκεκριμένα ερωτήματα παρέχοντας κλιμακωσιμότητα και χαμηλούς χρόνους εκτέλεσης.

Μια δεύτερη παρατήρηση είναι ότι οι αλγόριθμοι συνενώσεων του Datix έχουν καλύτερη απόδοση, τόσο για το topAS όσο και για το topDNS ερώτημα, από αυτούς που χρησιμοποιούνται στο Hive (70% βελτίωση) επειδή καταφέρνουν να αξιοποιούν αποτελεσματικά την κύρια μνήμη και να αποφεύγουν τις ακριβές μεταφορές δεδομένων μέσω δικτύου. Στην σύγκριση με το Shark η βασική υλοποίηση συνένωσης αποτυγχάνει να επεξεργαστεί το ερώτημα λόγω έλλειψης κύριας μνήμης. Ωστόσο το Shark με την βοήθεια των αλγορίθμων του Datix καταφέρνει να εκτελέσει όλα τα ερωτήματα παρουσιάζοντας τους καλύτερους χρόνους εκτέλεσης όπως φαίνεται στον Πίνακα 4.3. Μια ακόμα ενδιαφέρουσα παρατήρηση είναι η συμπεριφορά του συστήματός μας χρησιμοποιώντας διαμοιρασμό 2 (source IP, destination IP) και 3 διαστάσεων (source IP, destination IP, timestamp). Στην περίπτωση των τριών διαστάσεων παρατηρείται μια αύξηση του χρόνου εκτέλεσης των ερωτημάτων, ειδικά του ερωτήματος topDNS. Αυτό εξηγείται από το γεγονός ότι ο διαμοιρασμός σε περισσότερες διαστάσεις δημιουργεί

partitions με μεγαλύτερα εύρη IP. Αυτό απαιτεί την μεταφορά περισσότερων δεδομένων από την HBase και άρα επιβαρύνει αντίστοιχα την εκτέλεση των ερωτημάτων. Ωστόσο, αυτό δεν επηρεάζει σε μεγάλο βαθμό τα ερωτήματα topAS που μεταφέρουν συγκεκριμένο μέγεθος δεδομένων σε όλους τους mappers. Παρόλο που επιβαρύνει τα ερωτήματα που αφορούν όλο το σύνολο δεδομένων sflow, η χρήση 3D partitioning παρουσιάζει βελτιωμένη απόδοση σε ερωτήματα με φίλτρα όπως φαίνεται στον Πίνακα 4.3. Αυτό συμβαίνει γιατί ο διαμοιρασμός με βάση τα πεδία φιλτραρίσματος μας επιτρέπει να αποκλείσουμε ολόκληρα partitions από την εκτέλεση του ερωτήματος.

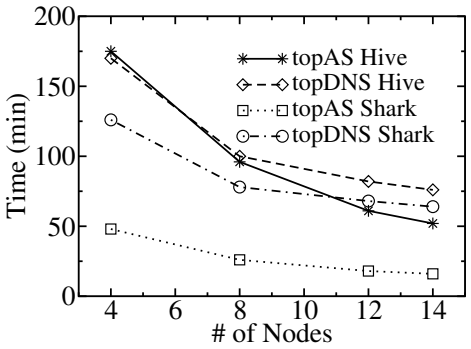


Σχήμα 4.4: *Σύγκριση του Hive με το Shark*

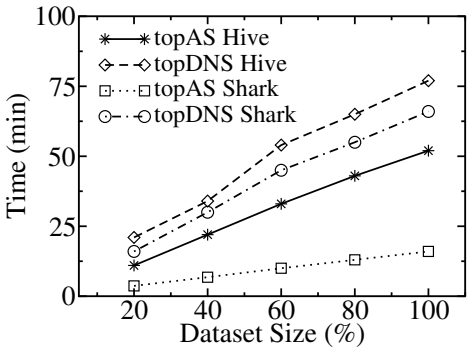
Έχοντας αναλύσει τα χαρακτηριστικά στα οποία διαφοροποιείται το Shark από το Hive παραθέτουμε τα αποτελέσματα από την εκτέλεση των ερωτημάτων για τα δύο συστήματα στο σχήμα 4.4. Παρατηρούμε ότι στην περίπτωση του ερωτήματος με τα αυτόνομα συστήματα το Shark είναι σημαντικά γρηγορότερο στην εκτέλεση από ότι το Hive. Παρόλα αυτά δεν συμβαίνει το ίδιο στην περίπτωση του άλλου ερωτήματος. Αυτό μπορεί να εξηγηθεί από το γεγονός ότι στη δεύτερη περίπτωση το σημαντικότερο ποσοστό του χρόνου εκτέλεσης μίας διεργασίας (περίπου το 90%) αφιερώνεται στη μεταφορά των απαιτούμενων δεδομένων από την HBase. Έτσι, στη δεύτερη περίπτωση το Shark δεν μπορεί να εκμεταλλευτεί πλήρως τα πλεονεκτήματα του. Αντίθετα, στην πρώτη περίπτωση που όλες οι λειτουργίες που απαιτούνται γίνονται in-memory φαίνεται η υπεροχή του Spark που οφείλεται κυρίως στον τρόπο που δρομολογεί τις διεργασίες και στο γεγονός ότι αποφεύγει να γράφει όλα τα ενδιάμεσα αποτελέσματα στο δίσκο.

4.4.4 Μελέτη κλιμακωσιμότητας του συστήματος

Τα σχήματα 4.6 και 4.5 δείχνουν την κλιμακωσιμότητα του συστήματός μας σε σχέση με το μέγεθος του συνόλου δεδομένων και τον αριθμό των διαθέσιμων κόμβων αντίστοιχα. Στην Εικόνα 4.6, μεταβάλλουμε τον όγκο των δεδομένων που υποβάλλονται σε επεξεργασία, ενώ

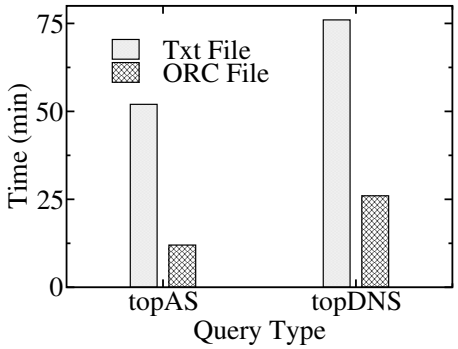


Σχήμα 4.5: *Κλιμακωσιμότητα με βάση τον αριθμό των υπολογιστικών πόρων*



Σχήμα 4.6: *Κλιμακωσιμότητα με βάση το μέγεθος των δεδομένων*

κρατάμε τον αριθμό των κόμβων σταθερό (14). Αντίθετα, στην Εικόνα 4.5 μεταβάλουμε τον αριθμό των κόμβων κατά την επεξεργασία ολόκληρου του συνόλου δεδομένων. Μια πρώτη παρατήρηση είναι ότι το σύστημά μας κλιμακώνει γραμμικά με το μέγεθος του συνόλου δεδομένων, ανεξάρτητα από το σύστημα εκτέλεσης που χρησιμοποιείται. Το Shark είναι ταχύτερο σε όλες τις περιπτώσεις σε σύγκριση με την αντίστοιχη υλοποίηση στο Hive. Αυτό οφείλεται κυρίως στο ακόλουθους λόγους. Κατ'αρχάς, το Shark αποφεύγει την εγγραφή δεδομένων στο δίσκο για όλα τα ενδιάμεσα αποτελέσματα των εργασιών MapReduce. Δεύτερον, χρησιμοποιεί ένα αποτελεσματικό αλγόριθμο δρομολόγησης των διεργασιών και, ως εκ τούτου, παρουσιάζει μικρότερη επιβάρυνση στο χρόνο εκτέλεσης του ερωτήματος. Επιπλέον, παρατηρούμε ότι για μικρό αριθμό κόμβων η κλιμακωσιμότητα του συστήματος είναι σχεδόν γραμμική, ενώ παρουσιάζεται μια μικρή υποβάθμιση στην απόδοση όσο οι διαθέσιμοι κόμβοι αυξάνονται. Τέλος, η διαφορά στο χρόνο εκτέλεσης με τη χρήση Hive και Shark είναι πολύ σημαντική για το ερώτημα topAS ανεξάρτητα από τον αριθμό τον κόμβων ή το μέγεθος του συνόλου δεδομένων.



Σχήμα 4.7: *Σύγκριση Text και ORC μορφής αρχείων*

Το Σχήμα 4.7 δείχνει την επιτάχυνση στον χρόνο εκτέλεσης για τους δύο τύπους ερώτημα όταν χρησιμοποιείτε η μορφή αρχείου ORC [Huai 14] αντί για την απλή μορφή κειμένου. Η μορφή αρχείου ORC χρησιμοποιεί μια προσέγγιση αποθήκευσης κατά στήλες και είναι σε θέση να ανακτήσει μόνο τα απαιτούμενα στοιχεία της στήλης για κάθε ερώτημα. Ως εκ τούτου, στα ερωτήματα που συνδυάζουν πληροφορίες μόνο από μερικές στήλες παρατηρούμε μια σημαντική επιτάχυνση στο συνολικό χρόνο εκτέλεσης.

Ως γενική παρατήρηση, η δύναμη του Datix έγκειται στις περιπτώσεις όπου το *meta-dataset* είναι αρκετά μεγάλο και η διαθέσιμη μνήμη RAM δεν είναι αρκετή για να εκτελέσει μια απλή map-join συνένωση. Ειδικότερα, το σχήμα διαμοιρασμού των δεδομένων μας έχει σχεδιαστεί για να ξεπεραστεί αυτός ο περιορισμός και μας δίνει τη δυνατότητα να απαντήσουμε αποτελεσματικά διάφορα ερωτήματα. Ωστόσο, όταν το *meta-dataset* είναι αρκετά μικρό, μια προσέγγιση Python που τρέχει σε έναν κεντρικό υπολογιστή με πολλούς πόρους (CPUs και μνήμη RAM) αναμένεται να αποφέρει συγκρίσιμη απόδοση σε σχέση με το Datix. Παρ 'όλα αυτά, το σύστημά μας αναδεικνύεται καλύτερο από τέτοιες μεθόδους και βελτιώνει σημαντικά την απόδοση του Hive και του Shark.

ΚΕΦΑΛΑΙΟ 5

Συσχετιζόμενες Εργασίες

Αυτό το κεφάλαιο παρουσιάζει τις συσχετιζόμενες με τη διατριβή εργασίες. Οργανώνουμε τις εργασίες σε δυο κατηγορίες: 1) αυτές που σχετίζονται με επεξεργασία RDF δεδομένων (Section 5.1), παρουσιάζοντας τόσο αναφορές σε RDF βάσεις δεδομένων όσο και σε κρυφές μνήμες για SPARQL ερωτήματα και 2) εργασίες που στοχεύουν στην επεξεργασία δεδομένων κίνησης δικτύων (Section 5.2).

5.1 RDF Βάσεις Δεδομένων

Σε αυτή την ενότητα, παρουσιάζουμε εργασίες ευρετηρίασης και επερώτησης RDF δεδομένων, διακρίνοντας τις σε δύο κατηγορίες, τα κεντρικά και τα κατανεμημένα συστήματα.

Κεντρικά Συστήματα

Το Hexastore [Weiss 08] είναι μια κεντρική λύση που βασίζεται στη δημιουργία έξι διαφορετικών ευρετηρίων, ένα για κάθε πιθανή διάταξη των subject-predicate-object μιας RDF τριπλέτας. Οι 6 αυτές αναδιατάξεις είναι οι *spo, sop, pso, pos, ops* και *osp*. Για παράδειγμα, το ευρετήριο *spo* περιέχει μια λίστα predicate για κάθε subject. Κάθε predicate *p* της προηγούμενης λίστας αναφέρεται σε ένα πίνακα που περιέχει όλα τα object που σχετίζονται με το συγκεκριμένο συνδυασμό *sp*. Τα ευρετήρια αυτά επιτρέπουν την ανάκτηση των δεδομένων που αντιστοιχούν σε κάθε δυνατό ερώτημα τριπλέτας με ελάχιστο κόστος.

Μια παρόμοια προσέγγιση ευρετηρίασης, μαζί με επιπρόσθετες στρατηγικές βελτιστοποίησης ερωτημάτων, ακολουθείται στο RDF-3X [Neumann 10a]. Το RDF-3X δημιουργεί 6 λεξικογραφικά ταξινομημένα RDF ευρετήρια (παρόμοια με το [Weiss 08]), καθώς και ευρετήρια που συλλέγουν στατιστικά στοιχεία για ζευγάρια και ανεξάρτητες RDF οντότητες, φτάνοντας συνολικά τα 15 ευρετήρια. Χρησιμοποιεί εκτενώς Merge συνενώσεις, ώστε να επιτύχει καλή απόδοση στα SPARQL ερωτήματα. Ωστόσο, η εκτέλεση του ερωτήματος εξαρτάται σε μεγάλο βαθμό από την ποσότητα της κύριας μνήμης που απαιτείται για την εκτέλεση των συνενώσεων, παρουσιάζοντας προβλήματα σε συνενώσεις με μικρή επιλεκτικότητα και μεγάλη είσοδο. Η χρήση ενός μόνο νήματος για την εκτέλεση του ερωτήματος, περιορίζει την επεκτασιμότητα του RDF-3X σε σύγχρονες αρχιτεκτονικές πολλαπλών πυρήνων.

Στο BitMat [Atre 08], οι RDF τριάδες αναπαριστώνται μέσω ενός 3-διαστάσεων *s, p, o* πίνακα bit. Κάθε στοιχείο του πίνακα είναι ένα bit που υποδηλώνει την παρουσία ή την απουσία της αντίστοιχης τριπλέτας. Το σύστημα διατηρεί πολλαπλές προβολές 2-διαστάσεων του αρχικού πίνακα, δημιουργώντας έτσι πολλαπλά ευρετήρια για όλους τους πιθανούς συνδυασμούς των subject-predicate-object. Ωστόσο, αυτή η προσέγγιση είναι αποτελεσματική σε κεντρικά υπολογιστικά περιβάλλοντα με μεγάλο μέγεθος κύριας μνήμης.

Άλλες συχνά χρησιμοποιούμενες κεντρικές βάσεις RDF δεδομένων είναι το Virtuoso [Erling 09], η Jena [Carroll 04] και το OWLIM [Kiryakov 05]. Παρόλα αυτά, όλες οι παραπάνω προσεγγίσεις λειτουργούν σε ένα μόνο μηχάνημα, περιορίζοντας την δυνατότητα αποθήκευσης μεγάλου όγκου δεδομένων καθώς την ικανότητα επεξεργασίας περίπλοκων ερωτημάτων.

Κατανεμημένα Συστήματα

Προκειμένου να αντιμετωπιστεί η πρόκληση των μεγάλου όγκου δεδομένων, η έρευνα έχει μετακινηθεί προς την κατεύθυνση της χρήσης κατανεμημένων συστημάτων για τη διαχείριση δεδομένων RDF. Μια πρώτη προσπάθεια προς αυτή την κατεύθυνση ήταν το 4store [Harris 09], το οποίο δημιουργεί ένα *pos* ευρετήριο κατανέμοντας τα δεδομένα του στους κόμβους ενός συμπλέγματος υπολογιστών. Ωστόσο, εκτός από τη μη αποδοτική αναζήτηση δεδομένων που προκύπτει από τη χρήση ενός μόνο RDF ευρετηρίου, το 4store δεν προσαρμόζει το είδος εκτέλεσης των συνενώσεων ανάλογα με την επιλεκτικότητα του κάθε ερωτήματος.

Το HadoopRDF [Husain 11] χρησιμοποιεί HDFS αρχεία (Hadoop Distributed File System) για την αποθήκευση των RDF ευρετηρίων του. Ουσιαστικά τα ονόματα των αρχείων χρησιμοποιούνται για τον διαχωρισμό των δεδομένων και τη δημιουργία ενός *pos* ευρετηρίου. Ωστόσο, αυτό δεν είναι ένα πλήρως λειτουργικό ευρετήριο δεδομένου ότι μπορεί να ανακτήσει μόνο συνδυασμούς subject-object για ένα δεδομένο predicate, αλλά όχι, για παράδειγμα, subject για ένα δεδομένο συνδυασμό predicate-object. Το HadoopRDF εκτελεί SPARQL συνενώσεις

χρησιμοποιώντας το προγραμματιστικό πλαίσιο MapReduce. Προτείνει έναν άπληστο αλγόριθμο εκτέλεσης συνενώσεων που προσπαθεί να μειώσει το συνολικό αριθμό των εργασιών MapReduce που χρειάζονται για την εκτέλεση ενός ερωτήματος. Αξίζει να σημειωθεί ότι, ο συγκεκριμένος άπληστος αλγόριθμος δεν λαμβάνει υπόψη την επιλεκτικότητα των συνενώσεων και των ερωτημάτων τριπλέτας. Τέλος, οι συνενώσεις εκτελούνται μόνο με MapReduce εργασίες, προκαλώντας μεγάλους χρόνους εκτέλεσης για επιλεκτικά ερωτήματα.

Πολλές ευρενητικές μελέτες έχουν γίνει και στον τομέα της βελτιστοποίησης της εκτέλεσης συνενώσεων με χρήση MapReduce [Blanas 10]. Σε αυτή την εργασία, οι συγγραφείς συγκρίνουν διαφορετικούς αλγόριθμους για την επεξεργασία μεγάλων πινάκων (log tables) που αποθηκεύονται σε αρχεία HDFS. Η κύρια διαφορά με το H_2RDF+ είναι η υλοποίηση συνενώσεων πάνω από ευρετήρια που αποθηκεύονται σε πίνακες HBase. Αυτό σημαίνει ότι οι αλγόριθμοί μας δεν χρειάζεται να επεξεργαστούν κάθε φορά το σύνολο των δεδομένων. Χρησιμοποιώντας τις δυνατότητες ευρετηρίασης της HBase επεξεργαζόμαστε μόνο το ποσό των δεδομένων που αντιστοιχεί στα ερωτήματα τριπλετών που θέλουμε να συνενώσουμε. Οι αλγόριθμοι που παρουσιάζονται στο [Blanas 10] δεν λαμβάνουν υπόψη την τυχόν προεπεξεργασία των δεδομένων και την ευρετηρίαση. Μπορούμε επίσης να χρησιμοποιήσουμε multi-way συνενώσεις που διαφέρουν από τις 2-way συνενώσεις που εφαρμόζονται στο [Blanas 10].

Η προηγούμενη έκδοση του συστήματός μας, H_2RDF [Papailiou 12], χρησιμοποιεί ένα σύστημα τριών ευρετηρίων και εξαρτάται από τον αλγόριθμο συνένωσης *Partial Input Hash-join*. Ο αλγόριθμος αυτός εκμεταλλεύεται τις δυνατότητες ευρετηρίασης της HBase εξετάζοντας αν μια συνένωση έχει μικρή εισόδου. Σε αυτή την περίπτωση, μόνο τα δεδομένα των επιλεκτικών ερωτημάτων τριπλέτας διαβάζονται από το ευρετήριο κατά τη map φάση. Τα υπόλοιπα ερωτήματα τριπλέτας συνενώνονται χρησιμοποιώντας τη reduce φάση της MapReduce εργασίας. Το H_2RDF χρησιμοποιεί επίσης προσαρμοστική κεντρική και κατανεμημένη εκτέλεση. Οι βασικές διαφορές με το H_2RDF+, βρίσκονται στους αλγορίθμους συνένωσης, στον αριθμό των ευρετηρίων (τρία έναντι έξι), στα πιο λεπτομερή στατιστικά στοιχεία και στο είδος και το μέγεθος των αναγνωριστικών (ID) που χρησιμοποιούνται για την αποθήκευση των δεδομένων.

Ένα εναλλακτικό σύστημα προτάθηκε στο [Huang 11]. Αυτή η μέθοδος ξεκινά διαμερίζοντας τον RDF γράφο σε ξεχωριστούς υπογράφους. Κάθε ξεχωριστή διαμέριση του γράφου αποθηκεύεται σε έναν κόμβο της συστοιχίας υπολογιστών, που χρησιμοποιεί το RDF-3X για την αποθήκευση των δεδομένων. Επιπλέον, προτάθηκε ένα σύστημα αντιγραφής των δεδομένων, με βάση το οποίο κάθε κόμβος διατηρεί εκτός από τις διαμερίσεις του επιπρόσθετες πληροφορίες σχετικά με τις τριπλέτες που βρίσκονται n βήματα μακριά (n-hop). Η διάταξη αυτή επιτρέπει την παράλληλη επεξεργασία των ερωτήσεων SPARQL που ικανοποιούν το n *hop guarantee*, δηλαδή έχουν διάμετρο μικρότερη από n. Σε περίπτωση που η εγγύηση αυτή δεν ικανοποιείται, η εκτέλεση του ερωτήματος γίνεται με χρήση του MapReduce. Το προτεινόμενο σύστημα πάσχει από τα ακόλουθα μειονεκτήματα: (1) Αργή εισαγωγή δεδομένων: εκτός

από την κεντρική διαμέριση του RDF γράφου, χρειάζεται επίσης ένα μεγάλο χρονικό διάστημα για να φορτώσει τις αντίστοιχες διαμερίσεις στις επιμέρους RDF-3X μηχανές (2) Οι συνενώσεις MapReduce που χρησιμοποιούνται εφαρμόζουν μη-βελτιστοποιημένους 2-way, hash-join αλγορίθμους. (3) Η υλοποίηση του n *hop guarantee* απαιτεί *εκθετικό*, σε σχέση με το n, μέγεθος αντιγραφής των RDF δεδομένων.

Στο [Zeng 13] παρουσιάζεται το Trinity.RDF, μια κατανεμημένη, βασισμένη στην κεντρική μνήμη του συστήματος βάση δεδομένων RDF. Οι συγγραφείς προτείνουν ένα μοντέλο εκτέλεσης του ερωτήματος που βασίζεται στην εξερεύνηση του RDF γράφου. Η τεχνική αυτή μπορεί να θεωρηθεί ως μια ακολουθία semi-join παρόμοια με την προσέγγιση που ακολουθείται στο BitMat. Το κύριο μειονέκτημα αυτού του συστήματος είναι ότι η απόδοση του δεσμεύεται από τη συνολική κύρια μνήμη του συμπλέγματος υπολογιστών. Αυτό θεωρούμε ότι δεν είναι μια επεκτάσιμη προσέγγιση, ιδίως αν ληφθεί υπόψη ότι οι συστοιχίες υπολογιστών που βασίζονται σε υπολογιστικά νέφη έχουν συνήθως περιορισμένες δυνατότητες κύριας μνήμης. Επιπλέον, τα αποτελέσματα της semi-join επεξεργασίας συγκεντρώνονται σε έναν κεντρικό κόμβο που είναι υπεύθυνος για την παραγωγή των τελικών αποτελεσμάτων. Αυτό μπορεί να αποτελέσει το δυσκολότερο μέρος της εκτέλεσης του ερωτήματος όταν: 1) τα ερωτήματα περιέχουν κύκλους. Η εκτέλεση ερωτημάτων με χρήση semi-join δεν μπορεί να μειώσει πλήρως το μέγεθος των αποτελεσμάτων για ερωτήματα που περιέχουν κύκλους [Bernstein 81], επιβαρύνοντας έτσι το τελευταίο βήμα της εκτέλεσης. 2) Η έξοδος του ερωτήματος είναι πραγματικά μεγάλη. Στην περίπτωση αυτή, ο τελευταίος διακομιστής θα πρέπει να επεξεργαστεί και να τυπώσει το σύνολο της εξόδου.

5.1.1 Κρυφές μνήμες για SPARQL ερωτήματα

Σε αυτή την ενότητα, παρουσιάζουμε μερικές από τις πιο σημαντικές ερευνητικές προσεγγίσεις για δημιουργία κρυφής μνήμης SPARQL αποτελεσμάτων. Ενώ αυτό είναι ένα δύσκολο πρόβλημα, υπάρχει περιορισμένος αριθμός σχετικών εργασιών. Μια πρώτη απόπειρα για εισαγωγή κρυφής μνήμης SPARQL έγινε στο [Martin 10], όπου μια σχεσιακή βάση δεδομένων κρατά μετά-δεδομένα για τα ερωτήματα που εκτελούνται και τα αποτελέσματά τους αποθηκεύονται προσωρινά. Ωστόσο, η προσέγγιση αυτή δεν μπορεί να αντιμετωπίσει το πρόβλημα του ισομορφισμού των γράφων των ερωτημάτων που παρουσιάζεται όταν το ίδιο SPARQL ερώτημα παρουσιάζει μικρές αποκλίσεις, όπως η αναδιάταξη των ερωτημάτων τριάδας, η μετονομασία των μεταβλητών, κ.α.

Μια πιο σύνθετη προσέγγιση παρουσιάστηκε στο [Yang 11], όπου τα κλειδιά της cache αποτελούνταν από ομαλοποιημένα δέντρα αλγεβρικών εκφράσεων, Algebra Expression Trees (AETs), που αντιστοιχούν σε αποθηκευμένα αποτελέσματα πλάνων εκτέλεσης συνενώσεων.

Ωστόσο, ένα προσωρινά αποθηκευμένο αποτέλεσμα χρησιμοποιείται μόνο σε πλάνα εκτέλεσης που το περιέχουν ακριβώς ως υπο-δέντρο. Αυτό επίσης δεν αποτελεί ένα γενικό πλαίσιο κρυφής μνήμης υπογράφων και οδηγεί σε χαμηλότερη αξιοποίηση της cache.

Στο [Lorey 13], οι συγγραφείς εισάγουν μια συνάρτηση ομοιότητας ερωτημάτων και έναν αλγόριθμο που μπορεί να ανιχνεύσει προσωρινά αποθηκευμένα ερωτήματα που μοιάζουν με το τρέχον ερώτημα. Ωστόσο, αυτή η άπληστη τεχνική δεν μπορεί να βρει όλα τα αποθηκευμένα αποτελέσματα που μπορούν να χρησιμοποιηθούν. Προτείνουν επίσης κάποιες ευριστικές μεθόδους που χρησιμοποιούνται για τη δημιουργία πιο γενικών ερωτημάτων με βάση αυτά που εκτελούνται. Η εκτέλεση και αποθήκευση τέτοιων SPARQL αποτελεσμάτων αυξάνει την αποδοτικότητα της κρυφής μνήμης και μοιάζει με την εκτέλεση *κερδοφόρων* ερωτημάτων που παρουσιάζουμε σε αυτή την εργασία. Η προσέγγισή αυτή βασίζεται και πάλι σε ευριστικές τεχνικές που δεν μπορούν να εξετάσουν το όφελος και την ευρετηρίαση όλων των πιθανώς χρήσιμων αποτελεσμάτων. Συνοψίζοντας, τα τρέχοντα πλαίσια προσωρινής αποθήκευσης αποτυγχάνουν να εντοπίσουν αποτελέσματα που αντιστοιχούν σε όλους τους υπογράφους του ερωτήματος και στη συνέχεια να τα χρησιμοποιήσουν για την παραγωγή βέλτιστων πλάνων εκτέλεσης.

Στο [Harbi 15], παρουσιάζεται η βάση δεδομένων AdHash. Το AdHash εφαρμόζει αρχικά μια hash κατανομή των RDF τριάδων και κατά την επεξεργασία του SPARQL ερωτήματος δυναμικά αναδιανέμει και να αντιγράφει τις απομακρυσμένες τριάδες που χρησιμοποιούνται κατά τη διαδικασία συνένωσης. Ως αποτέλεσμα, διαδοχικά ερωτήματα μπορούν να χρησιμοποιήσουν αυτή την προσαρμοστική αναδιάταξη για να εκτελεστούν παράλληλα χωρίς επικοινωνία δεδομένων. Αυτή η προσέγγιση δεν είναι ένα γενικό πλαίσιο κρυφής μνήμης, αλλά περισσότερο ένα σύστημα δυναμικής αναδιανομής και αντιγραφής που στοχεύει στην ελαχιστοποίηση του κόστους επικοινωνίας για διαδοχικά ερωτήματα. Αν για παράδειγμα εξετάσουμε την περίπτωση που ακριβώς το ίδιο ερώτημα εκτελείται δύο φορές, το δεύτερο ερώτημα, αν και δεν χρειάζεται μεταφορά δεδομένων, θα πρέπει να διαβάσει τα τοπικά δεδομένα και να εκτελέσει συνενώσεις. Ωστόσο, το σύστημά μας θα χρησιμοποιήσει κατ' ευθείαν το αποθηκευμένο αποτέλεσμα χωρίς να πραγματοποιήσει καμία επεξεργασία.

Εκτός από τις τεχνικές κρυφής μνήμης για SPARQL αποτέλεσμα υπάρχουν και άλλες σχετικές τεχνικές που προσπαθούν να αντιμετωπίσουν κάποια από τα μέρη του προβλήματος που αντιμετωπίζουμε σε αυτό το έγγραφο. Πολλές από τις προσεγγίσεις ευρετηρίασης δεδομένων γράφων προτείνουν τη χρήση ευρετηρίων βασισμένων σε συχνά μοτίβο γράφων [Zhao 07, Yan 04]. Τα συχνά εμφανιζόμενα μοτίβα γράφων ανακαλύπτονται και αποθηκεύονται κατά τη φάση εισαγωγής του συνόλου δεδομένων και μπορούν στη συνέχεια να χρησιμοποιηθούν για να απαντηθούν αποτελεσματικά τα ερωτήματα που τα περιέχουν. Επιπλέον, μεγάλο μέγεθος

έρευνας έχει γίνει στον τομέα της βελτιστοποίησης πολλαπλών ερωτημάτων [Roy 00] και επιλογής όψεων [Mistry 01] για σχεσιακές βάσεις δεδομένων. Προσεγγίσεις για τη βελτιστοποίηση πολλαπλών ερωτημάτων έχουν επίσης προταθεί για SPARQL ερωτήματα [Le 12]. Όλες αυτές οι τεχνικές εξαρτώνται είτε από τη γνώση του συνόλου των ερωτημάτων του φόρτου εργασίας ή σε ακριβές διαδικασίες εισαγωγής που βρίσκουν συχνά μοτίβα στο σύνολο δεδομένων. Αντίθετα, η προσέγγισή μας δεν απαιτεί καμία γνώση τόσο για το σύνολο δεδομένων όσο και για τα ερωτήματα του φόρτου εργασίας και στοχεύει στην προσαρμοστική ευρετηρίαση και την προσωρινή αποθήκευση των συχνών ερωτημάτων που παρατηρούνται στο φόρτο εργασίας.

5.2 Ανάλυση δεδομένων κίνησης δικτύων

Σε αυτή την ενότητα παρουσιάζουμε ερευνητικές εργασίες που σχετίζονται με την ανάλυση δεδομένων κίνησης δικτύων. Υπάρχει ένας αριθμός συστημάτων που έχουν προταθεί για την ανάλυση δεδομένων κίνησης δικτύων, καθένα από τα οποία αντιμετωπίζει μια ιδιαίτερη πτυχή αυτής της ευρείας ερευνητικά περιοχής. Στο [Lee 13], παρουσιάστηκε ένα σύστημα που χρησιμοποιεί τα libpcap αρχεία σε ένα κατανεμημένο περιβάλλον που συνδυάζει το MapReduce και το Hive για την επεξεργασία των δεδομένων. Οι συγγραφείς εφάρμοσαν ένα έξυπνο τρόπο ανάγνωσης των NetFlow αρχείων από πολλαπλούς υπολογιστικούς κόμβους με χρήση του Hadoop. Η δουλειά μας είναι συμπληρωματική με το [Lee 13], αφού οι αλγόριθμοι συνενώσεων που προτείνουμε μπορούν να ενσωματωθούν στο σύστημά τους και να χρησιμοποιηθούν για την εξαγωγή πρόσθετων πληροφοριών από *meta-datasets*.

Μια άλλη προσέγγιση προτείνεται στο [Li 13], όπου οι συγγραφείς χρησιμοποιούν τεχνικές μηχανικής μάθησης για την κατηγοριοποίηση των διαφόρων κόμβων του δικτύου με χρήση της πληροφορίας των sflow. Ουσιαστικά, αυτή η εργασία προσπαθεί να εξαγάγει πληροφορίες, όπως και η δικιά μας προσέγγιση, από την ανάλυση των αρχείων sFlow αξιοποιώντας το MapReduce και τις NoSQL βάσεις δεδομένων. Η διαφορά της προσέγγισής μας είναι ότι επιτρέπει τη χρήση αυθαίρετων και επιλεγμένων από το χρήστη *meta-datasets* οδηγώντας έτσι στην εξαγωγή πλουσιότερης πληροφορίας σχετικά με τη δρομολόγηση του δικτύου, τη διαστασιολόγηση και τα χαρακτηριστικά ασφαλείας.

Ένα σύστημα το οποίο μπορεί να εκτελέσει τόσο streaming όσο και batch επεξεργασία προκειμένου να αναλύσει τον συνεχώς αυξανόμενο όγκο των δεδομένων κίνησης του δικτύου παρουσιάζεται στο [Bumgardner 14]. Αντιμετωπίζει το πρόβλημα κλιμακωσιμότητας των υφιστάμενων συστημάτων με τη χρήση κατανεμημένων μεθοδολογιών όπως το MapReduce, αλλά παρ 'όλα αυτά δεν υποστηρίζει τη χρήση υψηλού επιπέδου γλωσσών επερωτήσεων. Επίσης οι συγκεκριμένες μεθοδολογίες είναι υλοποιημένες μόνο στο πλαίσιο του Hadoop και δεν μπορούν να εκτελεσθούν αποδοτικά πάνω από Spark.

Η εργασία [Herodotou 11] ασχολείται με την ιεραρχική κατάτμηση, και τη βελτιστοποίηση του πλάνου εκτέλεσης ενός ερωτήματος. Παρουσιάζεται μια δομή δέντρου κατάτμησης του χώρου διαστάσεων η οποία σε κάθε επίπεδο χρησιμοποιεί μια διαφορετική διάσταση. Επίσης αλλάζει τον τρόπο βελτιστοποίησης του πλάνου εκτέλεσης του ερωτήματος ώστε να λάβει υπόψιν του την κατάτμηση του χώρου. Αυτή η εργασία στοχεύει κυρίως σε παραδοσιακές κεντρικές βάσεις δεδομένων και δεν μπορεί να εφαρμοστεί άμεσα σε κατανεμημένα συστήματα. Επίσης η κατάτμηση του χώρου που προτείνουμε στο Datix έχει ως στόχο τον διαμοιρασμό των δεδομένων ώστε τα μέτα-δεδομένα να είναι σε θέση να χωρέσουν στην κεντρική μνήμη των map διεργασιών και να εκτελέσουμε αποδοτικά map-joins.

Στο [Johnson 15], οι συγγραφείς προτείνουν το TidalRace το οποίο χρησιμοποιείται για streaming δεδομένα και στοχεύει στην βελτιστοποίηση διεργασιών πάνω σε διαμοιρασμένα δεδομένα. Το Datix επικεντρώνεται σε επεξεργασία log αρχείων με batch τρόπο εκτέλεσης. Αντίθετα το TidalRace υποστηρίζει διαδοχικές ανανεώσεις των δεδομένων και της κατάτμησης καθώς και αναδιάταξη των δεδομένων.

Το DBStream [Bar 14] είναι μια βάση δεδομένων για streaming δεδομένα κίνησης δικτύων. Τα ερωτήματα που εκτελούμε σε αυτή την εργασία δεν μπορούν να εκτελεστούν στο DBStream αφού δεν επιτρέπετε η συνένωση των log δεδομένων με άλλα μέτα-δεδομένα. Το DBStream θα μπορούσε να επεκταθεί με τους αλγορίθμους που προτείνονται σε αυτή την εργασία ώστε να μπορέσει να χειριστεί μέτα-δεδομένα. Επίσης είναι υλοποιημένο πάνω από την κεντρική βάση δεδομένων PostgreSQL γεγονός που επηρεάζει την κλιμακωσιμότητά του στα μεγάλου όγκου δεδομένα. Το TicketDB [Baer 11], που είναι η εξέλιξη του DBStream επικεντρώνεται στην κατανεμημένη επεξεργασία αλλά συγκρίνεται μόνο με απλοϊκές υλοποιήσεις MapReduce ερωτημάτων. Επίσης δεν εφαρμόζει τεχνικές κατάτμηση και διαμοιρασμού των δεδομένων για την εκτέλεση αποδοτικών map-side συνενώσεων.

ΚΕΦΑΛΑΙΟ 6

Συμπεράσματα

Σε αυτή την εργασία, παρουσιάσαμε το H_2RDF+, μια πλήρως κατανεμημένη RDF βάση δεδομένων που μπορεί να αποθηκεύσει και να επεξεργαστεί αυθαίρετα μεγάλες ποσότητες RDF δεδομένων. Η κύρια συμβολή μας έγκειται στην κατανεμημένη εκτέλεση Merge και Sort-Merge συνενώσεων και τις προσαρμοστικές μας αποφάσεις για την κεντρική ή κατανεμημένη εκτέλεση συνενώσεων. Επιπλέον, βελτιστοποιήσαμε τόσο τη συμπίεση όσο και τις δυνατότητες ανάκτησης δεδομένων των HBase ευρετηρίων μας. Το H_2RDF+ είναι σε θέση να επιτύχει μεγάλες επιταχύνσεις και γραμμική κλιμάκωση στην φόρτωση και επεξεργασία δεδομένων RDF. Αυτά τα χαρακτηριστικά επιτρέπουν στο H_2RDF+ να επεξεργάζεται μη-επιλεκτικά ερωτήματα, σε ένα σύνολο δεδομένων μεγέθους 2.5TB, χρησιμοποιώντας μια μικρού μεγέθους συστοιχία, αποτελούμενη από 35 υπολογιστικούς κόμβους.

Προτείναμε επίσης ένα πλαίσιο κρυφής μνήμης για SPARQL ερωτήματα, το οποίο είναι σε θέση να επαναχρησιμοποιήσει αποτελεσματικά τα αποτελέσματα των εκτελούμενων ερωτημάτων. Παρουσιάσαμε έναν αλγόριθμο δημιουργίας κανονικοποιημένων ετικετών για SPARQL ερωτήματα. Επίσης προτείναμε μια νέα τεχνική απλοποίησης SPARQL ερωτημάτων που μπορεί να χρησιμοποιηθεί για να μειώσει την πολυπλοκότητα της βελτιστοποίησης και δημιουργίας κανονικοποιημένων ετικετών. Οι κανονικοποιημένες ετικέτες χρησιμοποιούνται για την αποτελεσματική αποθήκευση και ανάκτηση προσωρινά αποθηκευμένων αποτελεσμάτων. Επιπλέον, επεκτείναμε τον αλγόριθμο βελτιστοποίησης ερωτημάτων *DPccp* προσθέτοντας υποστήριξη για εξερεύνηση πλάνων που περιέχουν multi-way συνενώσεις και παραγωγή βέλτιστων πλάνων που λαμβάνουν υπόψιν την αξιοποίηση των προσωρινά αποθηκευμένων στην

κρυφή μνήμη αποτελεσμάτων. *Κερδοφόρα* ερωτήματα SPARQL ανακαλύπτονται και αποθηκεύονται προσωρινά, προκειμένου να μειωθούν οι χρόνοι απόκρισης για διάφορα είδη φόρτου εργασίας. Το πλαίσιο κρυφής μνήμης συνδέθηκε με την κατανεμημένη H_2RDF+ βάση δεδομένων παρουσιάζοντας μείωση του μέσου χρόνου απόκρισης κατά δύο τάξεις μεγέθους και προσφέροντας μικρούς χρόνους απόκρισης για σύνθετα σύνολα ερωτημάτων πάνω σε μεγάλου μεγέθους RDF δεδομένα.

Μια ακόμα συνεισφορά αυτής της διατριβής είναι το Datix, ένα πλήρως κατανεμημένο, ανοιχτού κώδικα σύστημα ανάλυσης δεδομένων κίνησης δικτύων. Το Datix βασίζεται σε τεχνικές έξυπνης κατανομής των δεδομένων, οι οποίες μπορούν να χρησιμοποιηθούν για την υποστήριξη γρήγορων συνενώσεων και αποδοτικών λειτουργιών επιλογής δεδομένων. Σαν αποτέλεσμα, το Datix πετυχαίνει να εκτελεί σε λίγα λεπτά ερωτήματα που απαιτούσαν έως και μέρες χρησιμοποιώντας τις υπάρχουσες τεχνολογίες κεντρικής επεξεργασίας. Επίσης παρουσιάζει έως και 70% μείωση χρόνου εκτέλεσης σε σχέση με αντίστοιχες δημοφιλείς πλατφόρμες κατανεμημένης επεξεργασίας, όπως το Hive και το Shark.

Συντμήσεις

AS	Autonomous System
BGP	Basic Graph Pattern
DNS	Domain Name System
HDFS	Hadoop Distributed File System
HFile	HBase file format
ISP	Internet Service Provider
IXP	Internet Exchange Points
NoSQL	Not only SQL
RAID	Redundant Array of Inexpensive Disks
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
TPC	Transaction Processing Performance Council
URI	Unique Resource Identifier

VM Virtual Machine

Δημοσιεύσεις

Περιοδικά

- D. Sarlis, **N. Papailiou**, I. Konstantinou, G. Smaragdakis and N. Koziris: "Datix: A System for Scalable Network Analytics." ACM SIGCOMM Computer Communication Review, 45(5), October 2015.
- T. Risse, E. Demidova, S. Dietze, W. Peters, **N. Papailiou**, K. Doka, Y. Stavrakas, V. Plachouras, P. Senellart, F. Carpentier, A. Mantrach, B. Cautis, P. Siehndel and D. Spiliotopoulos: "The ARCOMEM Architecture for Social- and Semantic-Driven Web Archiving" Future Internet Journal 2014, 6, 688-716.
- E. Demidova, N. Barbieri, S. Dietze, A. Funk, H. Holzmann, D. Maynard, **N. Papailiou** W. Peters, T. Risse and D. Spiliotopoulos: "Analysing and Enriching Focused Semantic Web Archives for Parliament Applications" Future Internet Journal 2014, 6, 433-456.

Συνέδρια

- **N. Papailiou**, D. Tsoumakos, P.Karras and N. Koziris: "Graph-Aware, Workload-Adaptive SPARQL Query Caching" In Proceedings of the 2015 ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD 2015), Melbourne, Australia
- K. Doka, **N. Papailiou**, D. Tsoumakos, C. Mantas and N. Koziris: "IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows." In Proceedings of the 2015

ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD 2015 Demo Track), Melbourne, Australia

- I. Giannakopoulos, D. Tsoumakos, **N. Papailiou** and N. Koziris: "PANIC: Modeling Application Performance over Virtualized Resources" In Proceedings of the 2015 IEEE International Conference on Cloud Engineering (IC2E 2015), 9-13 March, Tempe, AZ, USA
- I. Giannakopoulos, **N. Papailiou**, C. Mantas, I. Konstantinou, D. Tsoumakos and N. Koziris: "CELAR: Automated application elasticity platform" In proceedings of the 2014 IEEE International Conference on Big Data (BigData 2014), Washington DC, USA
- **N. Papailiou**, D. Tsoumakos, I. Konstantinou, P. Karras and N. Koziris: "Scalable Indexing and Adaptive Querying of RDF Data in the cloud" In proceedings of the 6th International Workshop on Semantic Web Information Management (SWIM 2014), Snowbird, Utah, USA
- **N. Papailiou**, D. Tsoumakos, I. Konstantinou, P.Karras and N. Koziris: "H2RDF+: An Efficient Data Management System for Big RDF Graphs." In Proceedings of the 2014 ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD 2014 Demo Track), Snowbird, Utah, USA
- **N. Papailiou**, I. Konstantinou, D. Tsoumakos, P.Karras and N. Koziris: "H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs." In proceedings of the 2013 IEEE International Conference on Big Data (BigData 2013), Santa Clara, CA, USA
- E. Demidova, N. Barbieri, S. Dietze, A. Funk, G. Gossen, D. Maynard, **N. Papailiou**, V. Plachouras, W. Peters, T. Risse, Y. Stavrakas and N. Tahmasebi: "Analysing Entities, Topics and Events in Community Memories." In proceedings of the 1st International Workshop on Archiving Community Memories, Lisbon, Portugal
- E. Angelou, **N. Papailiou**, I. Konstantinou, D. Tsoumakos and N. Koziris: "Automatic Scaling of Selective SPARQL Joins Using the TIRAMOLA System" In proceedings of the 4th International Workshop on Semantic Web Information Management (SWIM 2012), Scottsdale, Arizona, USA
- **N. Papailiou**, I. Konstantinou, D. Tsoumakos and N. Koziris: "H2RDF: Adaptive Query Processing on RDF Data in the Cloud" In Proceedings of the 21th International Conference on World Wide Web (WWW 2012 demo track), Lyon, France

Βιβλιογραφία

[Arvind 00] Vikraman Arvind & Johannes Köbler. *Graph isomorphism is low for zpp (np) and other lowness results*. In STACS. Springer, 2000.

[Atre 08] M. Atre, J. Srinivasan & J. Hendler. *BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries*. In ISWC, 2008.

[Baer 11] A. Baer, A. Barbuzzi, P. Michiardi & F. Ricciato. *Two Parallel Approaches to Network Data Analysis*. In LADIS, 2011.

[Bar 14] A. Bar, P. Casas, L. Golab & A. Finamore. *DBStream: an Online Aggregation, Filtering and Processing System for Network Traffic Monitoring*. In IWCMC, 2014.

[Bernstein 81] Philip A Bernstein & Dah-Ming W Chiu. *Using Semi-joins to Solve Relational Queries*. JACM, 1981.

[Blanas 10] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita & Yuanyuan Tian. *A Comparison of Join Algorithms for Log Processing in MaPreduce*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2010.

[Bonstrom 03] Valerie Bonstrom, Annika Hinze & Heinz Schweppe. *Storing RDF as a graph*. In Web Congress, 2003.

[Brickley 14] Dan Brickley & R.V. Guha. *RDF Schema 1.1*. W3C recommendation, 2014. http://www.w3.org/TR/rdf-schema/.

[Bumgardner 14] Vernon KC Bumgardner & Victor W Marek. *Scalable Hybrid Stream and Hadoop Network Analysis System*. In ACM SPEC, 2014.

[Carroll 04] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne & Kevin Wilkinson. *Jena: Implementing the Semantic Web Recommendations*. In WWW, 2004.

[Chang 08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes & Robert E Gruber. *Bigtable: A Distributed Storage System for Structured Data*. ACM TOCS 26(2), 2008.

[Chatzis 13a] N. Chatzis, G. Smaragdakis, A. Feldmann & W. Willinger. *There is More to IXPs than Meets the Eye*. CCR 45(5), 2013.

[Chatzis 13b] Nikolaos Chatzis, Georgios Smaragdakis, Jan Böttger, Thomas Krenc & Anja Feldmann. *On the Benefits of Using a Large IXP as an Internet Vantage Point*. In ACM IMC, 2013.

[Cisco 13] Cisco. *Cisco Visual Networking Index: Forecast and Methodology, 2013 – 2018*. Available at http://www.cisco.com, 2013.

[Codd 70] Edgar F Codd. *A relational model of data for large shared data banks*. Communications of the ACM, vol. 13, no. 6, pages 377–387, 1970.

[Darga 08] Paul T Darga, Karem A Sakallah & Igor L Markov. *Faster symmetry discovery using sparsity of symmetries*. In Proceedings of the 45th annual Design Automation Conference, pages 149–154. ACM, 2008.

[Dean 08] J. Dean & S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Comm. of the ACM 51(1), 2008.

[Durumeric 13] Z. Durumeric, E. Wustrow & J. A. Halderman. *ZMap: Fast Internet-Wide Scanning and its Security Applications*. In USENIX Security Symposium, 2013.

[Erling 09] Orri Erling & Ivan Mikhailov. *Virtuoso: RDF Support in a Native RDBMS*. In Semantic Web Information Management. 2009.

[Gallego 11] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto & Pablo de la Fuente. *An empirical study of real-world SPARQL queries*. In 1st International Workshop on Usage Analysis and the Web of Data (USEWOD), Hydebarabad, India, 2011.

[Gassner 93] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer & Yun Wang. *Query optimization in the IBM DB2 family*. IEEE Data Eng. Bull., vol. 16, no. 4, 1993.

[Gubichev 14] Andrey Gubichev & Thomas Neumann. *Exploiting the query structure for efficient join ordering in SPARQL queries.* In EDBT, pages 439–450, 2014.

[Guo 05] Yuanbo Guo, Zhengxiang Pan & Jeff Heflin. *LUBM: A Benchmark for OWL Knowledge Base Systems*. Web Semantics: Science, Services and Agents on the World Wide Web, vol. 3, no. 2, 2005.

[Harbi 15] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis & Nikos Mamoulis. *Evaluating SPARQL Queries on Massive RDF Datasets*. Proceedings of the VLDB Endowment, vol. 8, no. 12, 2015.

[Harris 09] S. Harris, N. Lamb & N. Shadbolt. *4store: The Design and Implementation of a Clustered RDF Store*. SSWS, 2009.

[Hartke 09] Stephen G Hartke & AJ Radcliffe. *Mckay's canonical graph labeling algorithm*. Communicating mathematics, vol. 479, 2009.

[Herodotou 11] Herodotos Herodotou, Nedyalko Borisov & Shivnath Babu. *Query Optimization Techniques for Partitioned Tables*. In ACM SIGMOD, 2011.

[Hoffart 11] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard de Melo & Gerhard Weikum. *YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages*. In WWW, 2011.

[Huai 14] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee & X. Zhang. *Major Technical Advancements in Apache Hive*. In ACM SIGMOD, 2014.

[Huang 11] Jiewen Huang, Daniel J. Abadi & Kun Ren. *Scalable SPARQL Querying of Large RDF Graphs*. PVLDB, vol. 4, no. 11, 2011.

[Husain 11] M. Husain, J. McGlothlin, M.M. Masud, L. Khan & B.M. Thuraisingham. *Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing*. TKDE, vol. 23, no. 9, 2011.

[Idreos 11] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson & Anastasia Ailamaki. *Here are my Data Files. Here are my Queries. Where are my Results?* In CIDR, 2011.

[Johnson 15] Theodore Johnson & Vladislav Shkapenyuk. *Data Stream Warehousing in Tidalrace.* In CIDR, 2015.

[Junttila 07] Tommi Junttila & Petteri Kaski. *Engineering an efficient canonical labeling tool for large and sparse graphs.* In David Applegate, Gerth Stølting Brodal, Daniel Panario & Robert Sedgewick, editeurs, Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics, pages 135–149. SIAM, 2007.

[Kiryakov 05] Atanas Kiryakov, Damyan Ognyanov & Dimitar Manov. *OWLIM - A Pragmatic Semantic Repository for OWL.* In WISE, 2005.

[Köbler 94] Johannes Köbler, Uwe Schöning & Jacobo Torán. The graph isomorphism problem: its structural complexity. 1994.

[Koukis 13] V. Koukis, C. Venetsanopoulos & N. Koziris. *∼okeanos: Building a Cloud, Cluster by Cluster.* IEEE Internet Computing 17(3), 2013.

[Le 12] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan & Feifei Li. *Scalable multi-query optimization for SPARQL.* In ICDE. IEEE, 2012.

[Lee 13] Yeonhee Lee & Youngseok Lee. *Toward Scalable Internet Traffic Measurement and Analysis with Hadoop.* CCR 43(1), 2013.

[Li 13] Bingdong Li, Mehmet Hadi Gunes, George Bebis & Jeff Springer. *A Supervised Machine Learning Approach to Classify Host Roles On Line Using sFlow.* In ACM HPDC, 2013.

[Lohr 12] Steve Lohr. *The age of big data.* New York Times, vol. 11, 2012.

[Lorey 13] Johannes Lorey & Felix Naumann. *Caching and Prefetching Strategies for SPARQL Queries.* In ESWC. 2013.

[Louis 75] Bentley Jon Louis. *Multidimensional Binary Search Trees Used for Associative Searching.* Comm. of the ACM 18(9), 1975.

[Manola 04] Frank Manola, Eric Miller, Brian McBride*et al.* *RDF primer.* W3C recommendation, vol. 10, no. 1-107, page 6, 2004.

[Manyika 11] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh & Angela H Byers. *Big data: The next frontier for innovation, competition, and productivity*. 2011.

[Martin 10] Michael Martin, Jörg Unbehauen & Sören Auer. *Improving the performance of semantic web applications with SPARQL query caching*. In The Semantic Web: Research and Applications. 2010.

[McKay 81] Brendan D McKay. Practical Graph Isomorphism. Department of Computer Science, Vanderbilt University, 1981.

[McKay 90] Brendan D McKay & Adolfo Piperno. *Nauty and Traces User's Guide*, 1990.

[McKay 14] Brendan D. McKay & Adolfo Piperno. *Practical graph isomorphism, {II}*. Journal of Symbolic Computation, vol. 60, no. 0, pages 94 – 112, 2014.

[Mistry 01] Hoshi Mistry, Prasan Roy, S Sudarshan & Krithi Ramamritham. *Materialized view selection and maintenance using multi-query optimization*. In ACM SIGMOD Record, 2001.

[Moerkotte 06] Guido Moerkotte & Thomas Neumann. *Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products*. In VLDB, 2006.

[Moerkotte 08] Guido Moerkotte & Thomas Neumann. *Dynamic programming strikes back*. In In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2008.

[Neumann 10a] Thomas Neumann & Gerhard Weikum. *The RDF-3X Engine for Scalable Management of RDF Data*. VLDBJ, vol. 19, no. 1, 2010.

[Neumann 10b] Thomas Neumann & Gerhard Weikum. *x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases*. PVLDB, vol. 3, no. 1-2, 2010.

[Neumann 11] Thomas Neumann & Guido Moerkotte. *Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins*. In IEEE 27th International Conference on Data Engineering (ICDE), pages 984–994. IEEE, 2011.

[Papailiou 12] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos & Nectarios Koziris. *H_2RDF: Adaptive Query Processing on RDF Data in the Cloud.* In WWW, 2012.

[Papailiou 13] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras & Nectarios Koziris. *H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs.* In IEEE BigData, 2013.

[Papailiou 14] Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras & Nectarios Koziris. *H2RDF+: An Efficient Data Management System for Big RDF Graphs.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2014.

[Papailiou 15] Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras & Nectarios Koziris. *Graph-Aware, Workload-Adaptive SPARQL Query Caching.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 1777–1792, 2015.

[Prud'hommeaux 06] Eric Prud'hommeaux & Andy Seaborne. *SPARQL Query Language for RDF.* W3C recommendation, 2006. http://www.w3.org/TR/rdf-sparql-query/.

[Roy 00] Prasan Roy, Sridhar Seshadri, S Sudarshan & Siddhesh Bhobe. *Efficient and extensible algorithms for multi query optimization.* ACM SIGMOD Record, 2000.

[Sahoo 10] Satya Sanket Sahoo. *Semantic Provenance: Modeling, Querying, and Application in Scientific Discovery.* PhD thesis, Wright State University, 2010.

[Shvachko 10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia & Robert Chansler. *The Hadoop Distributed File System.* In IEEE MSST, 2010.

[Stuckenschmidt 04] Heiner Stuckenschmidt, Richard Vdovjak, Geert-Jan Houben & Jeen Broekstra. *Index structures and algorithms for querying distributed RDF repositories.* In WWW, 2004.

[Tang 11] Liyin Tang & Namit Jain. *Join Strategies in Hive.* Hive Summit, 2011.

[Thusoo 09] Ashish Thusoo*et al. Hive: A Warehousing Solution over a Map-Reduce Framework.* VLDB, 2009.

[Tran 10] Thanh Tran & Günter Ladwig. *Structure index for RDF data*. In SemData, 2010.

[Tsialiamanis 12] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides & Peter Boncz. *Heuristics-based query optimisation for SPARQL*. In ICDT. ACM, 2012.

[W3C 15] W3C. *LargeTripleStores*. http://www.w3.org/wiki/LargeTripleStores, 2015.

[Weiss 08] Cathrin Weiss, Panagiotis Karras & Abraham Bernstein. *Hexastore: Sextuple Indexing for Semantic Web Data Management*. PVLDB, vol. 1, no. 1, 2008.

[Yan 04] Xifeng Yan, Philip S Yu & Jiawei Han. *Graph indexing: a frequent structure-based approach*. In SIGMOD, 2004.

[Yang 11] Mengdong Yang & Gang Wu. *Caching intermediate result of sparql queries*. In WWW, 2011.

[Zaharia 10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker & Ion Stoica. *Spark: Cluster Computing with Working Sets*. In USENIX conference on Hot topics in cloud computing, 2010.

[Zeng 13] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao & Zhongyuan Wang. *A Distributed Graph Engine for Web Scale RDF Data*. PVLDB, vol. 6, no. 4, 2013.

[Zhao 07] Peixiang Zhao, Jeffrey Xu Yu & Philip S Yu. *Graph indexing: tree+ delta<= graph*. In VLDB, 2007.