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

Μελέτη Απόδοσης Κατανεμημένων Βάσεων Δεδομένων Για Γράφους για Διαχείριση Διασυνδεδεμένων Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΖΑΝΑΚΗ ΒΑΣΙΛΙΚΗ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2016

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

Μελέτη Απόδοσης Κατανεμημένων Βάσεων Δεδομένων Για Γράφους για Διαχείριση Διασυνδεδεμένων Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΖΑΝΑΚΗ ΒΑΣΙΛΙΚΗ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31η Οκτωβρίου 2016.

Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επ. Καθηγητής Ε.Μ.Π.	Δημήτριος Τσουμάκος Επ. Καθηγητής Ιόνιο Πανεπιστήμιο

Αθήνα, Οκτώβριος 2016

Τζανάκη Βασιλική

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

Copyright © Τζανάκη Βασιλική, 2016.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

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

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

Περίληψη

Η πρόκληση του σημερινού κόσμου είναι τα δεδομένα. Ένας τεράστιος όγκος δεδομένων παράγεται κάθε δευτερόλεπτο και η δημιουργία τους συνεχίζει να αυξάνεται με ραγδαίο ρυθμό. Εικόνες, βίντεο, μουσική, έγγραφα, emails, αρχεία καταγραφής σφαλμάτων, δεδομένα από αισθητήρες και δορυφόρους, είναι μερικά μόνο από τα δεδομένα που παράγονται από τους ανθρώπους είτε από μηχανές, ως μέρος του Διαδικτύου των Πραγμάτων (Internet of Things). Όλα αυτά τα δεδομένα όμως δεν έχουν καμία αξία, αν δεν μπορούμε να τα μετατρέψουμε σε πολύτιμη πληροφορία και να τη χρησιμοποιήσουμε ώστε να βελτιώσουμε τις ζωές μας.

Ευτυχώς, τη σημερινή εποχή, διαθέτουμε πολύτιμα εργαλεία, τα οποία μπορούν να μας βοηθήσουν στην οργάνωση, στην αποθήκευση και στην ανάλυση όλων αυτών των δεδομένων. Η εισαγωγή του Υπολογιστικού Νέφους (Cloud Computing), καθώς και η ανάπτυξη διαφόρων κατανεμημένων συστημάτων και προγραμματιστικών μοντέλων μας έδωσε τη δυνατότητα να εξάγουμε ουσιαστική πληροφορία και να μετατρέψουμε όλα αυτά τα σκόρπια, φαινομενικά ασύνδετα μεταξύ τους δεδομένα, σε πολύτιμη γνώση.

Σε αυτή τη διπλωματική εργασία, επικεντρωνόμαστε περισσότερο στα δεδομένα με τη μορφή γράφων. Τα Διασυνδεδεμένα Δεδομένα (Linked Data), δηλαδή τα δεδομένα που δημοσιεύονται στις διάφορες ιστοσελίδες και συνδέονται μεταξύ τους, είναι μια κατηγορία δεδομένων που μπορεί να σχηματίσει τεράστιους γράφους. Αξιοποιούμε τις δυνατότητες που μας προσφέρει το Cloud Computing, καθώς και τα υπάρχοντα κατανεμημένα συστήματα, ώστε να φορτώσουμε δεδομένα αυτής της μορφής και να υλοποιήσουμε αλγορίθμους που απαντάνε σε διαφόρων ειδών ερωτήματα πάνω σε τέτοιου είδους δεδομένα. Στη συνέχεια, μελετάμε την απόδοση του συστήματός μας καθώς ο αριθμός των υπολογιστικών πόρων αυξάνεται και ελέγχουμε την εγκυρότητα των αποτελεσμάτων μας, με την εκτέλεση των ίδιων ερωτημάτων πάνω στα ίδια δεδομένα σε ένα κεντρικό σύστημα. Επιπλέον, ερευνούμε πώς ένα ερώτημα διαφορετικού είδους ή η αντικατάσταση διαφόρων παραμέτρων του, μπορεί να επηρεάσει το χρόνο υπολογισμό του.

Λέξεις κλειδιά

Hadoop, Giraph, Pregel, Openvirtuoso, Υπολογιστικό Νέφος, Κατανεμημένα συστήματα, MapReduce, Bulk Synchronous Model, RDF, Διασυνδεδεμένα Δεδομένα, γράφοι, DBpedia, SPARQL.

Abstract

Today's world challenge is data. A huge amount of data is generated every second and the pace of data creation continues to accelerate rapidly. Images, videos, music, documents, emails, machine logs, sensor data, satellite data are a few only the data produced either by people, or by machines as part of the Internt of Things (IoT). All this data though is worth nothing unless we are able to turn it into valuable information and use it in order to improve our lives.

Fortunately, nowadays we are holding in our hands precious tools, which can help us in the organization, storage and analysis of all this data. The introduction of the Cloud Computing model, as well as the development of various distributed frameworks and programming models has enabled us to extract meaningful information and transform all this scattered, seemingly unrelated data into valuable knowledge.

In this thesis, we are focusing more on graph data. Linked data, the data published in various websites and connected between them, are a category of data that can lead to huge graphs. We exploit the Cloud environment, as well as the existing distributed frameworks, in order to load data of this form and implement algorithms which allow us to answer various kinds of queries over such type of data. Then, we study the performance of our system as the number of computer resources increases and verify the validity of our results, by executing the same queries over the same datasets in a centralized system. Furthermore, we investigate how a different kind of query or the change of the various parameters involved, can influence the execution time.

Key words

Hadoop, Giraph, Openvirtuoso, Cloud Computing, Distributed systems, MapReduce, Bulk Synchronous Model, RDF, Linked data, graphs, DBpedia, Pregel, SPARQL.

Περιεχόμενα

Κατάλογος Σχημάτων

Κεφάλαιο 1

Εισαγωγή

1.1 Αντικείμενο της Διπλωματικής Εργασίας

Από τις πρώτες στιγμές της ανθρωπότητας κληθήκαμε να αντιμετωπίσουμε διάφορα προβλήματα, να τα καταλάβουμε, να τα αναλύσουμε και να βρούμε τρόπους και τεχνολογίες, που θα μας βοηθήσουν να επιβιώσουμε και να βελτιώσουμε τις ζωές μας. Μια από τις σημαντικότερες προκλήσεις της σημερινής εποχής είναι η αποδοτική διαχείριση των δεδομένων. Ένας τεράστιος όγκος δεδομένων παράγεται κάθε δευτερόλεπτο και ο ρυθμός της δημιουργίας τους συνεχίζει να αυξάνεται ραγδαία. Ο καθένας από εμάς, με τη χρήση ενός πλήθους ψηφιακών εφαρμογών και κοινωνικών δικτύων, παράγει καθημερινά δεδομένα σε διάφορες μορφές, όπως έγγραφα, ήχοι, βίντεο, φωτογραφίες, μηνύματα ηλεκτρονικού ταχυδρομείου, συμβάλλοντας έτσι σ'αυτήν την τεράστια αύξηση των δεδομένων. Όμως τα δεδομένα δεν παράγονται μόνο από τους ανθρώπους. Ο αριθμός των δεδομένων που παράγεται από τις μηχανές θα γίνει ακόμα μεγαλύτερος. Εξυπηρετητές (servers), αναγνώστες αναγνωριστικού ραδιοσυχνοτήτων (Radio Frequency ID readers), συσκευές καταγραφής κίνησης, ιατρικές συσκευές, δορυφόροι και αισθητήρες είναι μερικά μόνο παραδείγματα της τεράστιας παραγωγής δεδομένων από μηχανές με υψηλή ταχύτητα.

Όλα τα δεδομένα που παράγονται δεν μπορούν να αξιοποιηθούν και απαιτείται περαιτέρω επεξεργασία για την εξόρυξη χρήσιμης πληροφορίας. Επομένως, μια μεγάλη πρόκληση είναι η εύρεση προηγμένων συστημάτων για την οργάνωση και αποθήκευσή τους. Οι μεμονωμένοι υπολογιστές, ανεξάρτητα από το πόσο ισχυροί είναι, δεν είναι ικανοί να επεξεργαστούν αυτόν τον τεράστιο όγκο διαθέσιμης πληροφορίας μέσα σε εύλογο χρονικό διάστημα. Συνεπώς, τα κατανεμημένα συστήματα γίνονται όλο και πιο δημοφιλή, και μαζί με αυτά αναπτύσσονται νέα εργαλεία λογισμικού και προγραμματιστικά μοντέλα που μπορούν να αξιοποιήσουν αυτό το κατανεμημένο περιβάλλον.

Ένα από τα υπολογιστικά μοντέλα που που επιτρέπει την κατανεμημένη επεξεργασία δεδομένων είναι το Υπολογιστικό Νέφος (Cloud Computing). Έχει αποκτήσει μεγάλη δημοτικότητα στις μέρες μας, καθώς μπορεί να παρέχει υπηρεσίες που δεν υπήρχαν ποτέ πριν και στους παρόχους και στους χρήστες του 'Λογισμικό ως Υπηρεσίας' (Software as a Service, Saas). Ο επίσημος ορισμός του Υπολογιστικού Νέφους, όπως διατίθεται από το Εθνικό Ινστιτούτο Προτύπων και Τεχνολογίας (National Institute of Standards and Technology) [7] είναι ο ακόλουθος:

Το Υπολογιστικό Νέφος είναι ένα μοντέλο που επιτρέπει τη συνεχή, εύκολη και κατ'απαίτηση πρόσβαση σε μια ομάδα (pool) από παραμετροποιήσιμους υπολογιστικούς πόρους (όπως δίκτυα, εξυπηρετητές, αποθηκευτικού χώρους, εφαρμογές και υπηρεσίες) που μπορούν να δεσμευθούν και να απελευθερωθούν με ελάχιστη προσπάθεια διαχείρισης ή αλληλεπίδρασης με τον πάροχο των υπηρεσιών.

Τα πλεονεκτήματα που προέρχονται από αυτή την ελαστικότητα των πόρων είναι πρωτοφανή στην ιστορία της Τεχνολογίας της Πληροφορίας (IT), και έχουν μεταμορφώσει ένα τεράστιο μέρος της IT βιομηχανίας καθιστώντας το Λογισμικό ως Υπηρεσία ακόμα πιο ελκυστικό και αλλάζοντας ολοκληρωτικά τον τρόπο με τον οποίο το υλικό των υπολογιστών (hardware) σχεδιάζεται και αγοράζεται[8].

Οι προγραμματιστές και οι εταιρείες δε χρειάζεται να ανησυχούν για τις μεγάλες κεφαλαιουχικές δαπάνες σε hardware ώστε να αναπτύξουν τις υπηρεσίες τους ή το ανθρώπινο κόστος που απαιτείται για τη λειτουργία τους. Μπορούν να αρχίσουν να υλοποιούν καινοτόμες ιδέες ή υπηρεσίες χωρίς να υπάρχει λόγος να ανησυχούν για τις περιπτώσεις όπου η δημοτικότητα των υπηρεσιών τους δεν ανταποκρίνεται στις προσδοκίες τους ή αντίθετα αυξάνεται ξαφνικά, αφού μπορούν να δεσμεύουν και να απελευθερώνουν τους επιθυμητούς πόρους ανά πάσα στιγμή, χωρίς την ανθρώπινη αλληλεπίδραση με τον κάθε πάροχο υπηρεσιών. Επίσης, όλων των ειδών οι υπηρεσίες (π.χ. αποθήκευση, επεξεργασία, μεταφορά δεδομένων, εύρεση ενεργών λογαριασμών χρηστών) μπορούν να παρακολουθούνται, να ελέγχονται και να μετριούνται και για τον πάροχο και τον χρήστη των χρησιμοποιούμενων υπηρεσιών, προσφέροντας έτσι τη δυνάτητα για χρέωση ανάλογα με τη χρήση.

Η εισαγωγή του μοντέλου του Υπολογιστικού Νέφους απαιτεί να αλλάξει ο τρόπος με τον οποίο το λογισμικό γράφεται μέχρι τώρα. Ένα πρόγραμμα δεν χρειάζεται να εκτελείται σε ένα μεμονωμένο υπολογιστή για πολλές ώρες, ούτε υπάρχει η ανάγκη για την προμήθεια υπερυπολογιστών από την πλευρά των εταιρειών. Ένας μεγάλος αριθμός εμπορικών υπολογιστών, που επικοινωνούν μεταξύ τους μέσω ενός δικτύου, μπορεί να λύσει το ίδιο πρόβλημα σε πολύ μικρότερο χρονικό διάστημα. Για να εκμεταλλευτούμε αυτή τη δυνατότητα, πρέπει να γράψουμε προγράμματα που να μπορούν να είναι κλιμακώσιμα. Αυτό σημαίνει, ότι μεγάλα προβλήματα πρέπει να χωριστούν σε διάφορα απλούστερα, όπου το καθένα από αυτά υπολογίζεται στους μεμονωμένους κόμβους ενός κατανεμημένου συστήματος.

Χωρίς αμφιβολία, η υλοποίηση κατανεμημένων αλγορίθμων είναι πολύ πιο πολύπλοκη από αυτή των σειριακών, χρειάζεται έρευνα και απαιτεί σωστή διαχείρηση μιας συστοιχίας υπολογιστών (cluster). Παρόλα αυτά, τα πλεονεκτήματα της μείωσης του κόστους, της κλιμακωσιμότητας και της αξιοπιστίας είναι αυτά που καθιστούν τον κατανεμημένο προγραμματισμό πολύ πιο ελκυστικό. Έτσι έχουν οριστεί εγγενώς παραλληλοποιήσιμα προγραμματιστικά μοντέλα και έχουν αναπτυχθεί διεπαφές που επιτρέπουν στο προγραμματιστή να υλοποιήσει τον αλγόριθμο του, καθώς και συστήματα τα οποία μπορούν να εκτελούν το πρόγραμμα και να παρέχουν ανοχή σε σφάλματα, χωρίς κάποια επιπλέον προσπάθεια. Ένα από τα προγραμματιστικά μοντέλα που έχουν αποκτήσει μεγάλη δημοτικότητα στις μέρες μας είναι το MapReduce, με το Hadoop να είναι το σύστημα το οποίο το υλοποιεί με αποδοτικό τρόπο. Το Hadoop, παρόλα αυτά, μπορεί να θεωρηθεί απλώς ως ένα σημείο εκκίνησης, καθώς έχουν αναπτυχθεί πολλά άλλα συστήματα, όπως το Spark [9] και το Storm [10], καθένα από τα οποία επιχειρεί να προσφέρει το δικό του τρόπο για την επεξεργασία των λεγόμενων ‘Μεγάλων Δεδομένων’ (Big Data).

Τέτοια εργαλεία μαζί με πολλά άλλα που έχουν αναπτυχθεί και αυτά που πρόκειται να αναπτυχθούν στο μέλλον, μας δίνουν τη δυνατότητα για πρώτη φορά να επεξεργαστούμε ένα μεγάλο μέρος αυτού του ωκεανού από δεδομένα που μας περιτριγυρίζει. Τέτοιου είδους επεξεργασία μπορεί να μας παρέχει πληροφορία, που έχει τεράστια επίδραση στις ζωές μας, από τον τρόπο που ζούμε και εργαζόμαστε, στον τρόπο που διευθύνουμε επιχειρήσεις, πόλεις ή ακόμα και κυβερνήσεις. Οι εταιρείες μπορούν να χρησιμοποιήσουν αυτή την πληροφορία για να προβλέψουν τάσεις και συμπεριφορές καταναλωτών, οι κυβερνήσεις για να γίνουν πιο αποδοτικές, να εξοικονομήσουν χρήματα, να εντοπίσουν απάτες, είτε ακόμα και να ματαιώσουν τρομοκρατικές επιθέσεις. Ακόμα, και ο καθένας από εμάς θα μπορέσει να δει βελτιώσεις σε όλους τους τομείς της ζωής του, όπως στην υγεία, στην ιατρική, στην οικονομία, στην έρευνα, στην εκπαίδευση, στην πρόγνωση του καιρού, στον αθλητισμό, στις τηλεποικοινωνίες, στη μόδα, στην γεωργία και σε πολλούς άλλους.

Η πληθώρα των εργαλείων που έχουμε σήμερα στη διάθεσή μας, μετατρέπει την επεξεργασία του τεράστιου όγκου δεδομένων σε μια απλή διαδικασία. Η πραγματική πρόκληση όμως, δεν είναι η επεξεργασία των δεδομένων αυτή καθαυτή, αλλά το πώς τα δεδομένα μπορούν να μεταφραστούν σε πολύτιμη γνώση. Ο τρόπος με τον οποίο τα συλλέγουμε, τα επεξεργαζόμαστε, και τα ερμηνεύουμε μπορεί να παίξει κρίσιμο ρόλο στα συμπεράσματα που καταλήγουμε. Τις περισσότερες φορές, τα δεδομένα που διαχειριζόμαστε ως απομονωμένες νησίδες πληροφορίας δεν έχουν να προσφέρουν τί-

ποτα χρήσιμο. Ο τρόπος όμως με τον οποίο αυτά συνδέονται μεταξύ τους, μπορεί να μας αποκαλύψει πολλά πράγματα για τη δομή και τη φύση τους. Δεδομένα που διασυνδέονται με διάφορους τρόπους, μπορούμε να βρούμε σε κάθε είδους δίκτυο, όπως κοινωνικά δίκτυα, σημασιολογικά δίκτυα, δίκτυα μεταφορών, βάσεις γνώσεων και άλλα. Τα δεδομένα αυτά, μπορούν να σχηματίσουν τεράστιους γράφους, οι οποίοι μετά από κατάλληλη επεξεργασία, μπορούν να μας προσφέρουν πολύ μεγαλύτερη κατανόηση των δικτύων από τα οποία εξήχθησαν.

Η επεξεργασία των προαναφερθέντων γράφων δεδομένων σε ισχυρά υπολογιστικά κέντρα (data centers) μπορεί να παίξει σημαντικό ρόλο σε πολλούς τομείς. Αυτό άλλωστε ήταν και το κίνητρο για το γράψιμο αυτής της διπλωματικής εργασίας: η εύρεση ενός αποδοτικού τρόπου για την εξαγωγή χρήσιμης πληροφορίας από δεδομένα σε μορφή γράφων. Η επεξεργασία δεδομένων που σχηματίζουν γράφους είναι καθοριστικής σημασίας σε τομείς όπως μηχανές αναζήτησης που προσαρμόζονται στις απαιτήσεις των χρηστών (personalized information and recommendation systems) και συστήματα ανάλυσης και λήψης αποφάσεων. Τα δεδομένα γράφων έχουν συγκεκριμένη μορφή και γι'αυτό το λόγο έχουν αναπτυχθεί διάφορα συστήματα για την αποδοτική τους αποθήκευση και επεξεργασία σε Cloud υποδομές. Μερικά από τα πιο δημοφιλή είναι το Google Pregel[11], Google Big Query[12], Apache Giraph, [13] και το Graphlab[14].

Σε αυτή τη διπλωματική εργασία, επικεντρωνόμαστε κυρίως στα Διασυνδεδεμένα Δεδομένα (Linked Data), δηλαλή κομμάτια δεδομένων που δημιοσιεύονται σε διάφορες ιστοσελίδες με συνδέσεις μεταξύ τους. Στα πλαίσια της εργασίας αυτής θα μελετήσουμε και θα εγκαταστήσουμε ένα κατανεμημένο σύστημα για την επεξεργασία γράφων, θα εισάγουμε τα δεδομένα στο σύστημα, θα εκτελέσουμε ερωτήματα διαφορετικών ειδών πάνω σε σύνολα δεδομένων διαφορετικού μεγέθους. Θα αξιολογήσουμε πειραματικά το σύστημα μετρώντας το χρόνο απόκρισής τους και πώς αυτός μεταβάλλεται όταν ο αριθμός των υπολογιστικών πόρων αυξάνεται, όταν κάνουμε ένα ερώτημα διαφορετικού είδους, ή όταν τα δεδομένα που εμπλέκονται σε ένα ερώτημα του ίδιου είδους είναι διαφορετικά. Στη συνέχεια, θα πραγματοποιήσουμε τα παραπάνω σε ένα κεντρικό σύστημα, ώστε να δούμε αν η συμπεριφορά είναι ή ίδια, ή αν διαφέρει. Η χρήση ενός κεντρικού συστήματος μας επιτρέπει επιπλέον να επιβεβαιώσουμε την εγκυρότητα των αποτελεσμάτων των δικών μας κατανεμημένων αλγορίθμων.

1.2 Διάρθρωση του κειμένου

Το υπόλοιπο κείμενο της διπλωματικής εργασίας είναι οργανωμένο ως εξής:

Κεφάλαιο 2 Το κεφάλαιο αυτό περιλαμβάνει το απαραίτητο θεωρητικό υπόβραθο για την εργασία μας. Παρουσιάζει το σύστημα κατανεμημένης επεξεργασίας Hadoop, την αρχιτεκτονική του και το μοντέλο MapReduce. Εξηγεί γιατί το μοντέλο αυτό είναι ακατάλληλο για δεδομένα σε μορφή γράφου και μας εισάγει στο σύστημα κατανεμημένης επεξεργασία γράφων Giraph και ένα πιο κατάλληλο για το σκοπό αυτό προγραμματιστικό μοντέλο, το Bulk Synchronous Model (BSP).

Κεφάλαιο 3 Το κεφάλαιο αυτό παρέχει μια επισκόπηση των Linked Data, τους λόγους για τους οποίους είναι σημαντικά και ποιοι είναι οι διαθέσιμοι τρόποι για την περιγραφή και την επερώτησή τους. Επιπλέον περιέχει πληροφορία σχετικά με τα διάφορα ερωτήματα που επιλέξαμε να υλοποιήσουμε. Περιγράφονται τα διαφορετικά χαρακτηριστικά του καθενός και τον τρόπο με τον οποίο αυτά υλοποιήθηκαν στο Giraph, χρησιμοποιώντας το Bulk Synchronous Model.

Κεφάλαιο 4 Το κεφάλαιο αυτό περιγράφει το στήσιμο του περιβάλλοντος, τις εκδόσεις του Hadoop, Giraph και Openvirtuoso που εγκαταστήσαμε καθώς και το πώς παραμετροποιήσαμε τη συστοιχία των υπολογιστών που χρησιμοποιήσαμε στα πειράματά μας. Έπειτα, παρουσιάζονται τα αποτελέσματα των πειραμάτων μας. Εκτελέσαμε τα ίδια ερωτήματα και στο Giraph και στο

Openvirtuoso. Αρχικά, μελετήσαμε την επίδραση διαφόρων παραμέτρων του Hadoop κατά την αντιγραφή της εισόδου από το τοπικό στο κατανεμημένο σύστημα αρχείων, κατά τη φόρτωση του γράφου και κατά τον υπολογισμό των ερωτημάτων. Στη συνέχεια, εξετάσαμε την επίδραση του είδους του ερωτήματος, της χρήσης διαφορετικών συνόλων δεδομένων ως είσοδο, καθώς της χρήση διαφορετικών στιγμιοτύπων του ίδιου τύπου ερωτήματος στο χρόνο υπολογισμού του αποτελέσματος, καθώς και το πώς κλιμακώνεται το σύστημά μας καθώς ο αριθμός των διαθέσιμων πόρων αυξάνεται. Τέλος, συγκρίνουμε το χρόνο που απαιτείται για τη φόρτωση του γράφου και την εκτέλεση των ερωτημάτων στα δύο συστήματα και επιβεβαιώσαμε την εγκυρότητα και την πληρότητα των αποτελεσμάτων μας.

Κεφάλαιο 5 Στο κεφάλαιο αυτό γίνεται μια σύνοψη των προηγούμενων κεφαλαίων και παρουσιάζονται τα συμπεράσματα στα οποία καταλήξαμε. Επιπλέον, παρουσιάζονται μελλοντικές επεκτάσεις που θα μπορούσαν να γίνουν ώστε να βελτιώσουμε το σύστημά μας από άποψη απόδοσης και λειτουργικότητας.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

Ο τεράστιος όγκος πληροφορίας που είναι διαθέσιμος στο Ίντερνετ απαιτεί την ανάπτυξη κλιμακώσιμων συστημάτων, ικανών να επεξεργαστούν τον τεράστιο όγκο δεδομένων. Για το σκοπό αυτό, πολλά νέα συστήματα κάνουν την εμφάνισή τους, είτε από άποψη λογισμικού (software) είτε από άποψη υλικού υπολογιστών (hardware). Σημαντική πρόοδος έχει γίνει στη δημιουργία παράλληλων αλγορίθμων και στην ανάπτυξη προγραμματιστικών πλαισίων (frameworks), ενώ ταυτόχρονα συστοιχίες υπερυπολογιστών (High Performance Computing Clusters, HPC Clusters), μονάδες επεξεργασίας γραφικών για γενικού σκοπού επεξεργασία (GPGPUs) ή αρχιτεκτονικές πολυνηματικής διαμοιραζόμενης μνήμης (multihtreaded shared memory architectures), επιδιώκουν να συνεισφέρουν με το δικό τους τρόπο στην αποδοτικότερη επεξεργασία των δεδομένων. Παρόλα αυτά, τα κατανεμημένα συστήματα συνεχίζουν να αποτελούν μια καλύτερη λύση για την επεξεργασία των Big Data, καθώς επιτυγχάνουν να αντιμετωπίσουν έξυπνα και αποδοτικά τη διαθέσιμη πληροφορία, χωρίς να αυξάνουν την πολυπλοκότητα των προγραμμάτων ή να απαιτούν εξειδικευμένο hardware για τη χρήση τους.

Στα πλαίσια αυτής της διπλωματικής, πρόκειται να χρησιμοποιήσουμε το Apache Giraph [13], ένα από τα πιο δημοφιλή κατανεμημένα προγραμματιστικά πλαίσια (frameworks) για επεξεργασία γράφων. Για να δικαιολογήσουμε την επιλογή μας και να εξοικειώσουμε το χρήστη με τις κυριότερες έννοιες σχετικά με τα κατανεμημένα συστήματα, στο κεφάλαιο αυτό θα παρουσιάσουμε το απαραίτητο θεωρητικό υπόβαθρο, περιγράφοντας τους βασικούς μηχανισμούς και προγραμματιστικά μοντέλα που χρησιμοποιούνται σήμερα.

Συγκεκριμένα, στην ενότητα 2.1, θα παρουσιάσουμε το Hadoop και τις βασικές έννοιες γύρω από αυτό. Θα περιγράψουμε τα πλεονεκτήματά του σε σχέση με το παράλληλο προγραμματισμό, θα παρουσιάσουμε το MapReduce, το προγραμματιστικό μοντέλο που υλοποιεί το Hadoop, και θα δώσουμε μία βασική περιγραφή της αρχιτεκτονικής του. Η ενότητα 2.2, εξηγεί γιατί η προσέγγιση με το MapReduce είναι μη αποδοτική για δεδομένα γράφων, τη σχετική έρευνα πάνω σε αυτόν τον τομέα και το λόγο που το Giraph είναι το σύστημα της προτίμησής μας. Τέλος, η ενότητα 2.3, καλύπτει τις βασικές έννοιες γύρω από το Giraph, το προγραμματιστικό μοντέλο που χρησιμοποιεί (Bulk Synchronous Model - BSP) και την αντίστοιχη αρχιτεκτονική.

2.1 Hadoop

Το Hadoop [15] είναι ένα προγραμματιστικό πλαίσιο (framework) ανοιχτού κώδικα, υλοποιημένο σε Java, το οποία υποστηρίζει την επεξεργασία μεγάλων συνόλων δεδομένων σε κατανεμημένο περιβάλλον. Εμπνεύστηκε από τις δημοσιεύσεις της Google σχετικά με το MapReduce και το δικό της σύστημα αρχείων (Google File System, GFS).

Παρέχει ένα κατανεμημένο σύστημα αρχείων (Hadoop Distributed Filesystem, HDFS), το οποίο αποθηκεύει τα δεδομένα στους υπολογιστικούς κόμβους, προσφέροντας πολύ υψηλό ρυθμό μεταφοράς

δεδομένων συνολικά στο cluster. Επιπλέον το Hadoop υλοποιεί ένα παράλληλο υπολογιστικό μοντέλο, που ονομάζεται MapReduce και το οποίο χωρίζει την εφαρμογή σε πολλά μικρά κομμάτια εργασίας, το καθένα από τα οποία μπορεί να εκτελεστεί ξανά και ξανά σε κάθε κόμβο του cluster.

Παρόλο που τα ίδια προβλήματα μπορούν να λυθούν με διαφορετικό τρόπο, όπως με παραλληλοποίηση της εφαρμογής, το Hadoop φαίνεται να αποτελεί μία πολύ καλύτερη λύση για διάφορους λόγους:

Πρώτον, κατά την προσπάθεια παραλληλοποίησης μια εφαρμογής, οι προγραμματιστές πρέπει να αναλάβουν το μοίρασμα των δεδομένων, κάτι που δεν είναι πάντα εύκολο ή προφανές. Κάποιο ακατάλληλο μοίρασμα, μπορεί να οδηγήσει σε τμήματα πολύ διαφορετικού μεγέθους και παρόλο που οι διεργασίες που έχουν ολοκληρώσει το δικό τους κομμάτι προς επεξεργασία νωρίτερα, μπορεί να είναι ικανές να αναλάβουν επιπλέον εργασία, η όλη εκτέλεση εξαρτάται από το μεγαλύτερο κομμάτι. Μια καλύτερη προσέγγιση, θα ήταν να χωρίσουμε τα δεδομένα εισόδου σε τμήματα ίσου μεγέθους και να αναθέσουμε το κάθε τμήμα σε μια διεργασία.

Δεύτερον, ο συνδυασμός των αποτελεσμάτων από ανεξάρτητες διεργασίες μπορεί να απαιτεί περαιτέρω επεξεργασία.

Τρίτον, η υπολογιστική ικανότητα μιας μεμονωμένης υπολογιστικής μονάδας είναι περιορισμένη, με αποτέλεσμα να είναι αδύνατο να έχουμε μεγαλύτερη ταχύτητα για συγκεκριμένο σύνολο δεδομένων και αριθμό επεξεργαστών. Αντίθετα, η χρήση πολλαπών υπολογιστικών μονάδων, μπορεί να οδηγήσει σε πολύ καλύτερη απόδοση με τη σωστή ρύθμιση διαφόρων παραμέτρων.

Τέλος, ένας μεγάλος αριθμός από σύνολα δεδομένων, έχει τόσο μεγάλο μέγεθος, που είναι αδύνατον να επεξεργαστούν σε μια μεμονωμένη υπολογιστική μονάδα, καθιστώντας αναγκαία την χρήση πολλών υπολογιστικών μονάδων για την επεξεργασία τους.

Επομένως, παρόλο που είναι εφικτό να παραλληλοποιήσουμε την επεξεργασία, στην πράξη είναι πολύπλοκο. Χρησιμοποιώντας ένα προγραμματιστικό πλαίσιο σαν το Hadoop, το οποίο μπορεί να αναλάβει τέτοιου είδους ζητήματα είναι πολύ μεγάλη βοήθεια. Οι προγραμματιστές μπορούν να δουλέψουν στο κύριο κομμάτι της επεξεργασίας της πληροφορίας, ενώ το μοίρασμα των δεδομένων, η κατανομή τους και η παραλληλοποίηση των διαφόρων εργασιών σε ένα κατανεμημένο περιβάλλον, καθώς και η κλιμακωσιμότητα και η ανοχή σε σφάλματα υλικού βρίσκονται υπό την ευθύνη του MapReduce [16].

2.1.1 MapReduce

Το MapReduce [17] είναι ένα από τα βασικότερα στοχεία του Apache Hadoop framework. Είναι ένα προγραμματιστικό μοντέλο για την επεξεργασία και την παραγωγή μεγάλων συνόλων δεδομένων με ένα παράλληλο, κατανεμημένο αλγόριθμο σε ένα cluster. Λειτουργεί με το να χωρίζει την επεξεργασία σε δύο βασικές φάσεις: τη φάση του *map* και τη φάση του *reduce*. Οι αντίστοιχες map και reduce συναρτήσεις έχουν την ακόλουθη γενική μορφή:

$$map : (k1, v1) -> list(k2, v2)$$

$$reduce : (k2, list(v2)) -> list(k3, v3)$$

Η map συνάρτηση δέχεται ως είσοδο ένα ζεύγος κλειδιού-τιμής και παράγει ως έξοδο μια λίστα από ενδιάμεσα κλειδιά-τιμές. Η reduce συνάρτηση δέχεται ένα κλειδί και μια λίστα τιμών και παρέχει μια λίστα από ζεύγη κλειδιού-τιμής ως έξοδο.

Οι τύποι των προαναφερθέντων ζευγών κλειδιού-τιμής, μπορούν να επιλεχθούν από τον προγραμματιστή και μπορεί να είναι διαφορετικοί μεταξύ τους. Σε κάθε περίπτωση όμως, ή είσοδος της συνάρτησης reduce πρέπει να έχεις τους ίδιους τύπους με την έξοδο της map συνάρτησης.

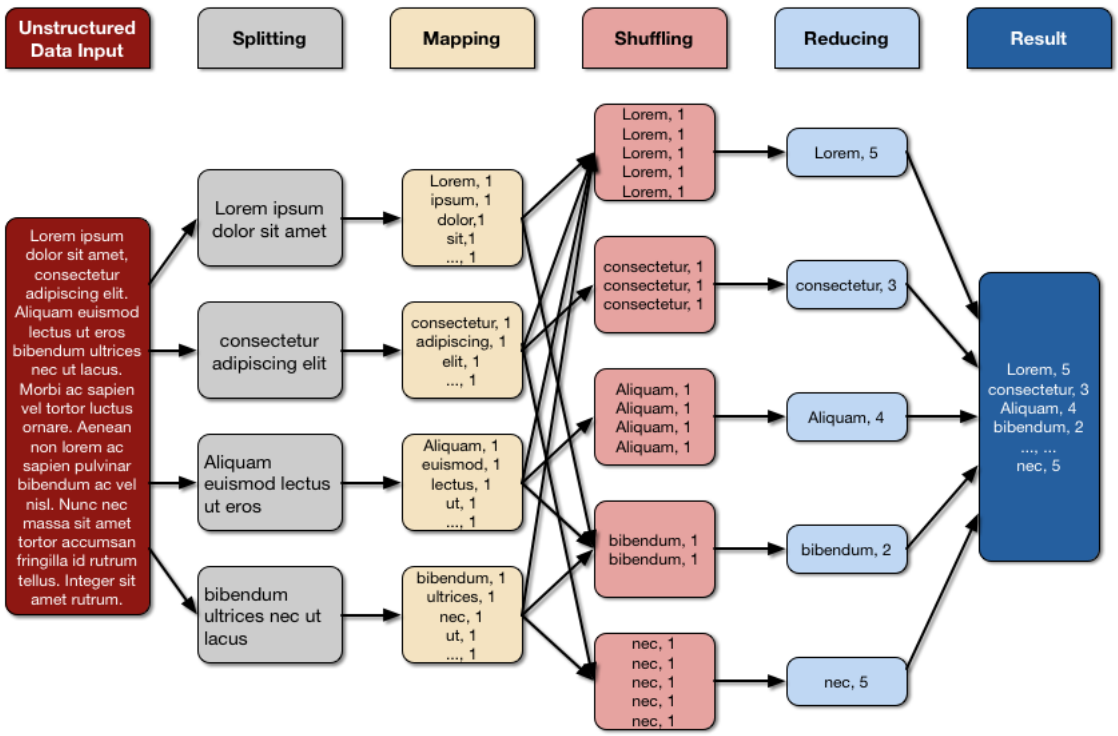
Για να κάνουμε πιο κατανοητό το πως γίνεται η επεξεργασία με τη βοήθεια του MapReduce, μπορούμε να κοιτάξουμε το παράδειγμα *Καταμέτρηση Λέξεων*, το οποίο μετράει τις εμφανίσεις κάθε λέξης σε ένα σύνολο από έγγραφα.

```
map(String input_key, String input_value):
    //input_key: document name
    //input_value: document contents
    for each word w in input_value:
        emit (w, 1);

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int sum = 0;
    for each v in intermediate_values:
        sum += v;
    emit (word, sum);
```

Στο παράδειγμα αυτό, το κάθε έγγραφο χωρίζεται σε λέξεις και η κάθε λέξη μετράται από τη map συνάρτηση, χρησιμοποιώντας τη λέξη ως το αποτέλεσμα-κλειδί. Το framework αναλαμβάνει να συγκεντρώσει μαζί όλα τα ζεύγη με το ίδιο κλειδί και να τα τροφοδοτήσει στη συνάρτηση reduce, η οποία με τη σειρά της αθροίζει όλες τις τιμές που παίρνει ως είσοδο, για να βρει τις συνολικές εμφανίσεις αυτής της λέξης.

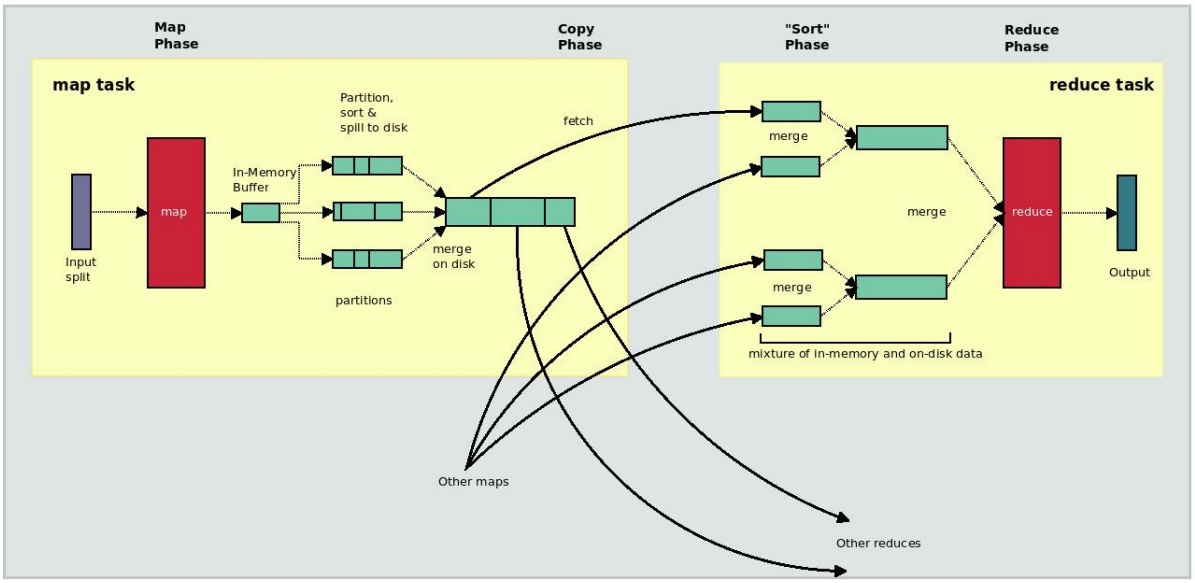
Η ροή των δεδομένων απεικονίζεται στο σχήμα 2.1:



Σχήμα 2.1: Ροή δεδομένων για εφαρμογή καταμέτρησης λέξεων με χρήση του προγραμματιστικού μοντέλου MapReduce [1]

Για την εκτέλεση μιας MapReduce εργασίας, ο προγραμματιστής απλώς χρειάζεται να ορίσει τις συναρτήσεις map και reduce, και το MapReduce είναι υπεύθυνο για όλα τα υπόλοιπα. Στην πράξη όμως, είναι σημαντικό να έχει κανείς μια στοιχειώδη κατανόηση του τρόπου λειτουργίας του συστήματος, ειδικά αν θέλει να βελτιστοποιήσει το MapReduce πρόγραμμά του.

Όλη η διαδικασία απεικονίζεται στο σχήμα 2.2:



Σχήμα 2.2: Τρόπος λειτουργίας του προγραμματιστικού μοντέλου MapReduce

Όπως βλέπουμε στην εικόνα, ο Mapper περιλαμβάνει τα ακόλουθα στάδια:

Στάδιο Επεξεργασίας Πρώτα, η αρχική είσοδος χωρίζεται σε ισομεγέθη τμήματα. Το Hadoop δημιουργεί μια map εργασία (map task) για το κάθε τμήμα, το οποίο τρέχει την ορισμένη από το χρήστη συνάρτηση για την κάθε εγγραφή του τμήματος.

Στάδιο Spill Η έξοδος του map γράφεται σε έναν κυκλικό απομονωτή μνήμης (circular memory buffer), που σχετίζεται με το κάθε map task. Όταν το μέγεθος του απομονωτή, φτάσει ένα κατώφλι (η προκαθορισμένη επιλογή είναι ένα νήμα που τρέχει στο υπόβαθρο αρχίζει να τοποθετεί(spill) τα περιεχόμενα στο δίσκο, ενώ παράλληλα το map συνεχίζει να γράφει δεδομένα στον απομονωτή μέχρι να γεμίσει, οπότε και σταματάει, έως ότου ολοκληρωθεί η μεταφορά στο δίσκο.

Στάδιο Κατάτμησης Ανάλογα με το αν έχουμε ένα ή πολλά reduce tasks έχουμε κατάτμηση (partitioning). Συγκεκριμένα, πριν το γράψιμο στο δίσκο ένα νήμα που τρέχει στο υπόβαθρο χωρίζει τα δεδομένα σε τμήματα (partitions) που αντιστοιχούν στους reducers στους οποίους θα σταλθούν τελικά. Μπορούν να υπάρχουν πολλά κλειδιά (και οι αντίστοιχες τιμές) σε κάθε partition, αλλά οι εγγραφές για ένα οποιοδήποτε κλειδί βρίσκονται όλες στο ίδιο. Ο χρήστης μπορεί να καθορίσει τον τρόπο με τον οποίο θα γίνει η κατάτμηση, χρησιμοποιώντας τη δική του συνάρτηση γι'αυτό το σκοπό, αν και συνήθως ο προεπιλεγμένος τρόπος κατάτμησης, μία συνάρτηση κατακερματισμού (hash function) λειτουργεί πολύ καλά.

Στάδιο Ταξινόμησης Μέσα σε κάθε partition, το νήμα που βρίσκεται στο υπόβαθρο κάνει μία ταξινόμηση ανά κλειδί στη μνήμη. Το ταξινομημένο αποτέλεσμα παρέχεται στη συνδυαστική συνάρτηση (combiner function) αν υπάρχει, έτσι ώστε λιγότερα δεδομένα να γράφονται στο δίσκο και να μεταφέρονται στο reducer.

Στάδιο Συγχώνευσης(Merging) Πριν ολοκληρωθεί η map λειτουργία, τα spill αρχεία συγχωνεύονται σε ένα μοναδικό ταξινομημένο αρχείο [16].

Ο Reducer έχει τρία στάδια:

Στάδιο Αντιγραφής Τα αρχεία εξόδου των map tasks για κάθε partition, μεταφέρονται (μέσω HTTP πρωτοκόλου) στο μέρος που βρίσκονται τα reduce tasks (είτε είναι ένα ή περισσότερα). Τα reduce tasks, μπορεί να χρειάζονται την έξοδο από πολλά map tasks, τα οποία μπορεί να μην ολοκληρωθούν ταυτόχρονα. Έτσι τα reduce tasks αρχίζουν την αντιγραφή μόλις τελειώσει το καθένα.

Στάδιο Ταξινόμησης Όταν έχουν αντιγραφτεί όλες οι έξοδοι από τα map, το reduce task τις συγχωνεύει διατηρώντας την ταξινομημένη σειρά.

Στάδιο Μείωσης (Reduce phase) Σε αυτό το στάδιο, καλείται η reduce συνάρτηση για κάθε κλειδί στην ταξινομημένη έξοδο. Η έξοδος αυτού του σταδίου γράφεται κατευθείαν στο σύστημα αρχείων, συνήθως το HDFS.

Επομένως, είναι φανερό τώρα, πώς τα προγράμματα τα οποία είναι γραμμένα με αυτόν τον τρόπο είναι εγγενώς παραλληλοποιήσιμα από την πλατφόρμα MapReduce. Το MapReduce frameowrk δεν απαιτεί κάποια ειδική συστοιχία υπολογιστών (cluster) για παραλληλοποίηση, αλλά μπορεί να αποτελείται από απλά εμπορικά μηχανήματα.

2.1.2 HDFS

Το Hadoop χρησιμοποιείται κυρίως για την επεξεργασία μεγάλων συνόλων δεδομένων, τα οποία συνήθως ξεπερνούν τη χωρητικότα μιας μεμονωμένης φυσικής μηχανής. Επομένως, είναι αναγκαίο να χρησιμοποιήσουμε ένα κατανεμημένο σύστημα αρχείων, το οποίο να διαχειρίζεται την αποθήκευση όχι σε ένα μεμονωμένο υπολογιστή αλλά σε ένα ολόκληρο δίκτυο υπολογιστών.

Το Hadoop γι'αυτο το σκοπό χρησιμοποιεί ένα κατανεμημένο σύστημα αρχείων, που ονομάζεται *Hadoop Distributed Filesystem* ή HDFS και μπορεί να τρέχει σε συμπλέγματα εμπορικών υπολογιστών [16].

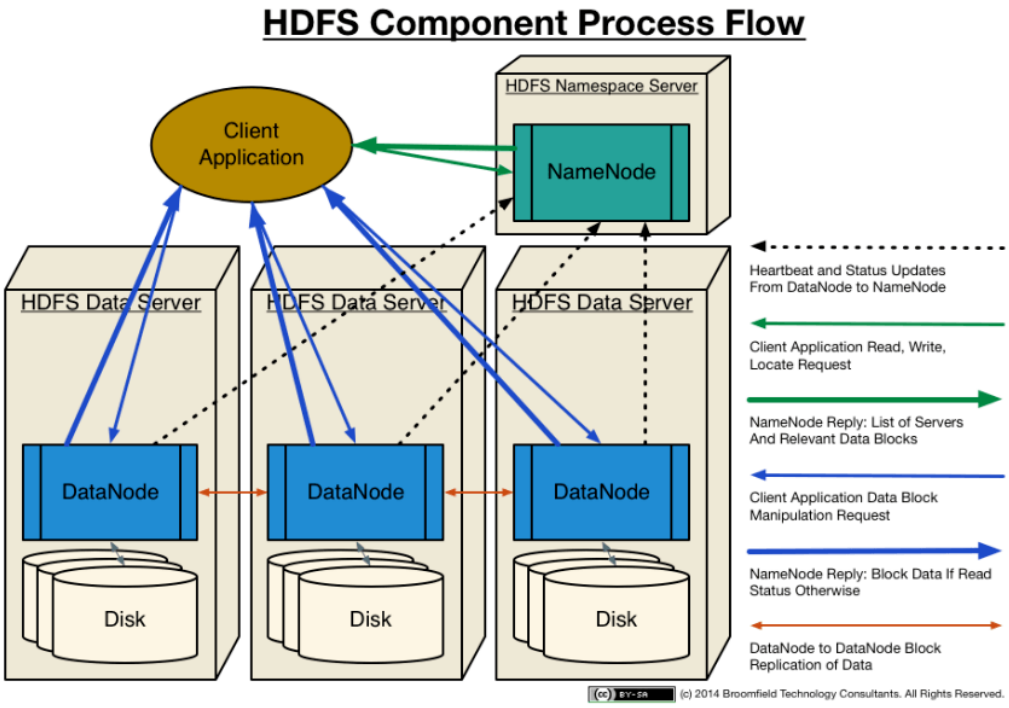
Ένα HDFS cluster έχει δύο τύπους κόμβων που λειτουργούν σε αρχιτεκτονική Master-Slave (Αφέντη-Εργάτη): τον Master *Namenode* και ένα σύνολο από Slaves, τα *Datanodes*, όπως βλέπουμε στην εικόνα 2.3.

Το *Namenode* είναι υπεύθυνο για τη διαχείριση του συστήματος αρχείων. Διατηρεί ένα δέντρο του συστήματος αρχείων και μεταδεδομένα για όλα τα αρχεία και τους καταλόγους στο δέντρο. Αυτή η πληροφορία αποθηκεύεται μόνιμα στον τοπικό δίσκο. Επίσης το Namenode γνωρίζει πού βρίσκονται όλοι τα datanodes στους οποίους είναι αποθηκευμένα όλα τα τμήματα(blocks) από ένα συγκεκριμένο αρχείο. Όμως δεν αποθηκεύει μόνιμα τις τοποθεσίες των τμημάτων αυτών, γιατί αυτή η πληροφορία ανακατασκευάζεται από τους datanodes όταν το σύστημα ξεκινάει.

Από την άλλη μεριά, τα *Datanodes* είναι οι εργάτες του συστήματος αρχείων. Αποθηκεύουν και ανακτούν blocks, όποτε τους ζητηθεί και δίνουν αναφορά στο Namenode ανά τακτά χρονικά διαστήματα με μια λίστα από τα blocks που αποθηκεύουν.

Το HDFS μπορεί να είναι πολύτιμο αλλά δεν είναι κατάλληλο για κάθε είδους εφαρμογή. Αποτελεί μια από τις καλύτερες επιλογές, όταν έχουμε να διαχειριστούμε πολύ μεγάλα αρχεία της τάξης των εκατοντάδων giagabytes, terabytes ή και ακόμα μεγαλύτερα. Όμως είναι ακατάλληλο στην περίπτωση πολλών μικρών αρχείων, καθώς το Namenode είναι υποχρεωμένο να διατηρήσει τα μεταδεδομένα του συστήματος αρχείων στη μνήμη του, θέτοντας έτσι ένα όριο στον αριθμό των αρχείο που μπορούν να υπάρχουν.

Επιπλέον, χρησιμοποιείται το μοντέλο WORM(Write-once, read-many), το οποίο σημαίνει ότι είναι κατάλληλο μόνο σε περιπτώσεις που ένα σύνολο δεδομένων αντιγράφεται από την πηγή και στη



Σχήμα 2.3: HDFS Architecture [1]

συνέχεια πολλές επιμέρους αναλύσεις γίνονται στο ίδιο σύνολο δεδομένων με την πάροδο του χρόνου και όχι σε περιπτώσεις που απαιτείται άμεση πρόσβαση στα δεδομένα.

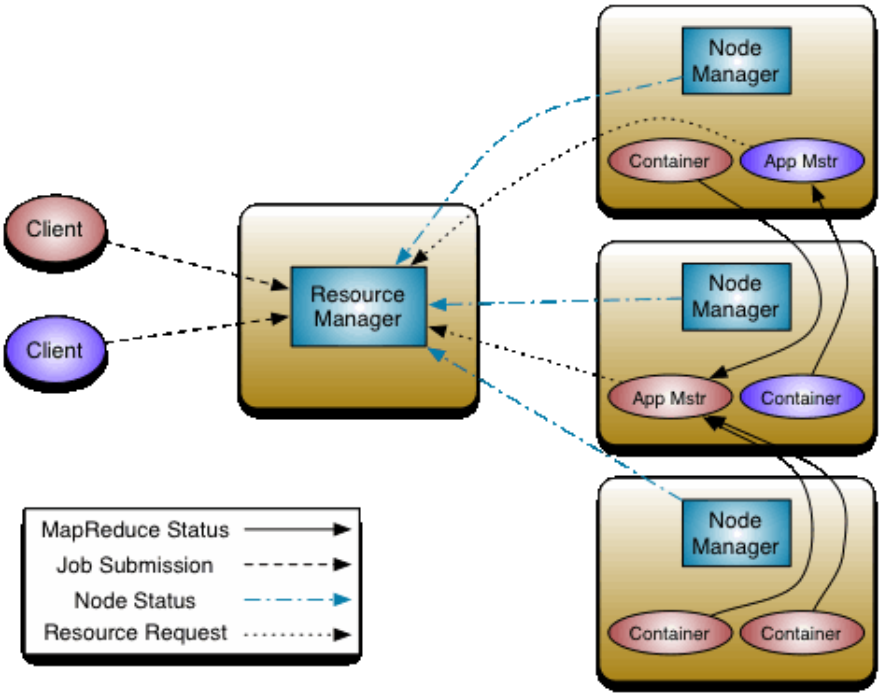
Τέλος, τα αρχεία στο HDFS γράφονται από μία μόνο πηγή, μόνο στο τέλος του αρχείου Έτσι, δεν μπορούν να γράφουν πολλοί ταυτόχρονα ή να κάνουν αλλαγές σε οποιδήποτε σημείο του αρχείου, παρά μόνο να κάνουν προσθήκες στο τέλος.

2.1.3 Αρχιτεκτονική του Apache Hadoop - Apache Hadoop YARN

Το Apache Hadoop YARN [2] είναι ένα σύστημα που χρησιμοποιείται για τον προγραμματισμό των εργασιών και τη διαχείριση των πόρων του cluster, η αρχιτεκτονική του οποίου απεικονίζεται στο σχήμα 2.4.

Η θεμελιώδης ιδέα του YARN είναι ο χωρισμός των λειτουργιών της διαχείρισης των πόρων και του προγραμματισμού/παρακολούθησης των εργασιών σε ξεχωριστούς δαίμονες (daemons). Έτσι έχουμε ένα κεντρικό διαχειριστή πόρων, τον ResourceManager (RM) και έναν υπεύθυνο για τις εφαρμογές, τον ApplicationMaster (AM). Στην έκδοση του Hadoop που χρησιμοποιήσαμε, το ρόλο του ResourceManager και του ApplicationMaster παίζουν ο JobTracker και TaskTracker αντίστοιχα.

JobTracker Ο JobTracker [18] είναι η διεπαφή μεταξύ μιας εφαρμογής πελάτη και της πλατφόρμας Hadoop. Είναι η υπηρεσία του Hadoop που οδηγεί τις διάφορες MapReduce εργασίες σε συγκεκριμένους κόμβους, ιδανικά σε αυτούς που έχουν τα δεδομένα ή τουλάχιστον βρίσκονται κοντά σε αυτά. Με την υποβολή μιας εργασίας στο Hadoop cluster ο Jobtracker είναι υπεύθυνος για να επικοινωνήσει με το Namenode, ώστε να καθορίσει πού βρίσκονται οι διάφορες ομάδες δεδομένων του αρχικού αρχείου εισόδου, να βρει ποιοι είναι οι TaskTracker κόμβοι με διαθέσιμες υποδοχές (slots) κοντά στα δεδομένα, και να αναθέσει την εργασία στους επιλεγμένους



Σχήμα 2.4: Αρχιτεκτονική του Apache Hadoop YARN [2]

κόμβους. Μετά την υποβολή της εργασίας στους TaskTrackers, αυτοί παρακολουθούνται, ώστε να γίνουν οι απαραίτητες ενέργειες σε περίπτωση αποτυχίας.

TaskTracker Σε κάθε Hadoop cluster μαζί με τον DataNode, ξεκινάει και μια διεργασία, που ονομάζεται TaskTracker, σε κάθε κόμβο του cluster τόπου αποθηκεύονται δεδομένα. Ο κάθε TaskTracker ρυθμίζεται με ένα σύνολο διαθέσιμων υποδοχών (slots), ο οποίος υποδηλώνει τον αριθμό των εργασιών που μπορούν να εκτελεστούν παράλληλα. Όταν ο JobTracker προσπαθεί να βρει πού θα υποβάλλει μια εργασία, επιδιώκει πρώτα να βρει μια διαθέσιμη υποδοχή στον ίδιο server όπου βρίσκεται ο DataNode που περιέχει τα δεδομένα.

2.2 Επεξεργασίων γράφων με το MapReduce

Σε αυτή τη διπλωματική εργασία επικεντρωνόμαστε στην ανάλυση των Διαδυνδεδεμένων Δεδομένων (Linked Data). Τα Linked Data, έχουν το χαρακτηριστικό ότι είναι τεράστια σε μέγεθος και μπορούν να σχηματίσουν τεράστιους γράφους. Επομένως, για την επεξεργασία τους, πρέπει να λάβουμε υπόψη μας και τους δύο αυτούς παράγοντες. Όπως έχουμε ήδη αναφέρει, η επεξεργασία των μεγάλων δεδομένων, μπορεί να γίνει εύκολα και αποδοτικά με την βοήθεια της πλατφόρμας MapReduce, η οποία παρέχει ένα απλό και ισχυρό μοντέλο, το οποίο επιτρέπει στους πργραμματιστές να κατασκευάσουν παράλληλους, κλιμακώσιμους αλγόριθμους, ικανούς να τρέξουν σε συπλέγματα εμπορικών υπολογιστών. Παρόλα αυτά, η εγγενής μορφή αυτών των δεδομένων καθιστά το μοντέλο MapReduce ακατάλληλο.

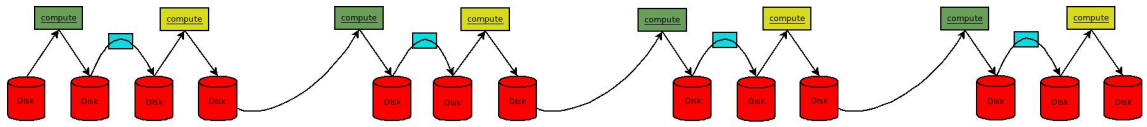
Γενικά, οι αλγόριθμοι για την επεξεργασία γράφων είναι επαναληπτικοί και χρειάζονται να διασχίσουν το γράφο με κάποιο τρόπο. Το MapReduce Μοντέλο όμως, δεν παρέχει άμεση υποστήριξη για την ανάλυση εργασιών πάνω σε επαναληπτικά δεδομένα. Αντί γι'αυτό, οι χρήστες καλούνται να σχεδιάσουν επαναληπτικά προγράμματα, με το να να συνδέουν αλυσιδωτά πολλαπλές MapReduce εργασίες και να καθορίζουν την εκτέλεσή τους χρησιμοποιώντας ένα πρόγραμμα-οδηγό. Αυτή η προσέγγιση όμως δεν είναι κατάλληλη για επεξεργασία γράφων, καθώς απαιτεί το πέρασμα ολόκληρης

της κατάσταης του γράφου από το ένα στάδιο στο επόμενο, κάτι το οποίο εκτός από την ανάγκη για συντονισμό των βημάτων μιας αλυσιδωτής Mapreduce εργασίας, απαιτεί επιπλέον επικοινωνία καθώς και το αντίστοιχο κόστος σειριοποίησης.

Παρά τους παραπάνω περιορισμούς, πολλές βιβλιοθήκες για την επεξεργασία γράφων έχουν δημιουργηθεί βασιζόμενες στο MapReduce εξαιτίας της ικανότητάς του να τρέξουν αξιόπιστα σε περιβάλλοντα παραγωγής. Επαναληπτικές πλατφόρμες στη μορφή του MapReduce προγραμματιστικού μοντέλου έχουν ερευνηθεί στο Twister[19] και στο Haloop[20]. Παρόλα αυτά, αυτές οι προσεγγίσεις παραμένουν μη αποδοτικές για επεξεργασία γράφων, γιατί η αποδοτικότητα των υπολογισμών γράφων εξαρτάται σε μεγάλο βαθμό από τα δίκτυο, αφού οι δομές γράφων στέλνονται μεταξύ των επεξεργαστών μέσω δικτύου μετά από κάθε επανάληψη.

Ενώ μεγάλο πλήθος από τα δεδομένα παραμένουν αμετάβλητα από επανάληψη σε επανάληψη, τα δεδομένα πρέπει να φορτώνονται και να επεξεργάζονται ξανά σε κάθε επανάληψη, οδηγώντας σε σπατάλη των μονάδων Εισόδου/Εξόδου (I/O), των υπολογιστικών πόρων και του bandwidth του δικτύου.

Στο σχήμα 2.5 μπορούμε να δούμε το παραπάνω χαρακτηριστικό του MapReduce:



Σχήμα 2.5: Αλυσιδωτή σύνδεση εργασιών MapReduce

Για την επίλυση του προβλήματος που παρουσιάζεται στο Hadoop κατά την επεξεργασία γράφων λόγω της αρχιτεκτονικής του συγκεκριμένου συστήματος, διάφορα κατανεμημένα συστήματα για επεξεργασία γράφων έχουν εισαχθεί. Συγκεκριμένα, το 2010, η Google πρωτοπόρησε σε αυτήν την περιοχή με την εισαγωγή του Pregel [11], ως μια κλιμακώσιμη πλατφόρμα για την υλοποίηση αλγορίθμων γράφων. Το Pregel ακολουθεί μια προσέγγιση, βασισμένη γύρω από την κορυφή (vertex-centric approach), την οποία εμπνεύστηκε από το προγραμματιστικό μοντέλο Bulk Synchronous Model(BSP) [3], όπου τα προγράμματα είναι υλοποιημένα ως μια αλληλουχία επαναλήψεων. Το Pregel στοχεύει στη μαζική επεξεργασία (batch oriented processing), πραγματοποιεί όλους τους υπλογισμούς στη μνήμη, τρέχει στη δική του υποδομή και ακολουθεί αρχιτεκτονική Master-Slave.

Δυστυχώς, ο πηγαίος κώδικας του Pregel δε δημοσιοποιήθηκε. Το Apache Giraph σχεδιάστηκε για να φέρει την επεξεργασία γράφων μεγάλης-κλίμακας στην κοινότητα ανοιχτού λογισμικού, βασιζόμενο στο μοντέλο του Pregel, ενώ παράλληλα παρέχει τη δυνατότητα να τρέξει στην υπάρχουσα υποδομή του Hadoop.

Εκτός από το Giraph, ασύγχρονα μοντέλα υπολογισμού γράφων έχουν προταθεί σε συστήματα όπως το Signal Collect [21], το GraphLab [14] και το Grace[22]. Ο ασύγχρονος υπολογισμός γράφων έχει δειχτεί ότι συγκλίνει πιο γρήγορα για κάποιες εφαρμογές, όμως ταυτόχρονα προσθέτει αξιοσημείωτη πολυπλοκότητα στο σύστημα και στον προγραμματιστή. Δεν υπάρχει επαναληπτικότητα, αφού ο κάθε αλγόριθμος χρειάζεται ειδικό χειρισμό, με αποτέλεσμα να είναι πολύ δύσκολος ο εντοπισμός των λαθών, καθώς και το αν αυτά βρίσκονται στην υποδομή του συστήματος ή στον κώδικα της εφαρμογής. Ακόμη, η ασύγχρονη φύση των αλγορίθμων μπορεί να οδηγήσει σε τεράστιες ουρές μηνυμάτων για κάποιες κορυφές, οδηγώντας σε προβλήματα έλλειψης μνήμης.

2.3 Giraph

Για το σκοπό αυτής της διπλωματικής εργασίας πρόκειται να χρησιμοποιήσουμε το Giraph [13]. Αρχικά θα περιγράψουμε το BSP υπολογιστικό μοντέλο, το οποίο υλοποιεί το Giraph, έπειτα θα παρου-

σιάσομε τον τρόπο λειτουργίας του και το λόγο που το προτιμήσαμε ανάμεσα στα υπόλοιπα διαθέσιμα frameworks.

2.3.1 Το προγραμματιστικό μοντέλο BSP

Είναι εμφανές ότι στις μέρες μας, με τη ραγδαία αύξηση των δεδομένων και τους υπολογισμούς που αυτή συνεπάγεται, οι σειριακοί αλγόριθμοι δεν είναι βιώσιμη λύση. Γι'αυτό τα τελευταία χρόνια έχει προταθεί ένας μεγάλος αριθμός από παράλληλα μοντέλα, όπως SIMD (Single-Instruction Multipe-Data) παραλληλισμό, σύγχρονη αποστολή μηνύματος (synchronous message passing), λογικός προγραμματισμό, μείωση των γράφων (graph reduction), και διαφόρων ειδών εικονικής διαμοιραζόμενης μνήμης, βασισμένη σε κρυφή μνήμη (cache-based virtual shared memory). Παρόλο που καθεμία από αυτές τις προσεγγίσεις έχει τα δικά της πλεονεκτήματα, οι περισσότερες από αυτές στερούνται καθολικής εφαρμογής, ή καθιστούν πολύ δύσκολο να επιτευχθεί μεταφερσιμότητα και απόδοση. Αυτές που βασίζονται στο πέρασμα μηνυμάτων είναι ανεπαρκή εξαιτίας της πολυπλοκότητας της δημιουργίας των σωστών ζευγών επικοινωνίας (αποστολή και λήψη) σε μεγάλο και πολύπλοκο λογισμικό, οδηγώντας πολλές φορές το σύστημα σε αδιέξοδο.

Το **Bulk Synchronous Parallel** (BSP) από την άλλη, είναι πολύ διαφορετικό από τις προαναφερθείσες προσεγγίσεις. Προσφέρει έναν τρόπο για το σχεδιασμό καθολικών, κλιμακώσιμων, πλήρως μεταφέρσιμων προγραμμάτων, τα οποία μπορούν να προσφέρουν υψηλή απόδοση με προβλέψιμο τρόπο, σε κάθε γενικού σκοπού παράλληλη αρχιτεκτονική. Επιπλέον επιτρέπει την ορθρότητα των παράλληλων προγραμμάτων να καθορίζεται με τρόπο, εξίσου απλό με αυτόν των σειριακών προγραμμάτων. Το κλειδί για την καθολική του εφαρμογή ανάμεσα σε όλο το φάσμα της παράλληλης αρχιτεκτονικής, είναι το γεγονός ότι αποσυνδέει δύο θεμελιώδεις πτυχές του παράλληλου υπολογισμού: την επικοινωνία και το συγχρονισμό.

Το BSP προγραμματιστικό μοντέλο

Ένα BSP σύστημα αποτελείται από:

1. μηχανές ικανές για την επεξεργασία και/ή συναλλαγές τοπικής μνήμης (local memory transactions) (π.χ. επεξεργαστές)
2. ένα δίκτυο το οποίο δρομολογεί μηνύματα μεταξύ ζευγών των παραπάνω μηχανών και
3. μια εγκατάσταση υλικού που να επιτρέπει το συγχρονισμό όλων ή ενός υποσυνόλου/μέρους των παραπάνω μηχανών.

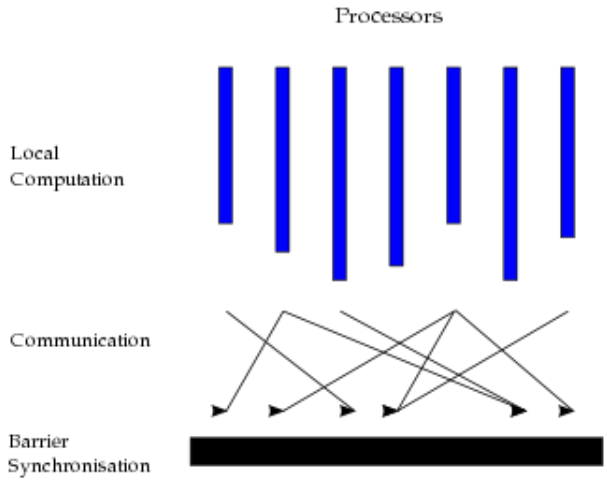
Το παραπάνω εύκολα μεταφράζεται ως ένα σύνολο επεξεργαστών, οι οποίοι μπορούν να ακολουθήσουν διαφορετικά νήματα υπολογισμού, με τον κάθε επεξεργαστή να είναι εξοπλισμένο με γρήγορη τοπική μνήμη και να διασυνδέεται με ένα δίκτυο επικοινωνίας. Ένας BSP αλγόριθμος βασίζεται σε μεγάλο βαθμό στο τρίτο χαρακτηριστικό. Ο υπολογισμός γίνεται σε μια σειρά από καθολικά βήματα που ονομάζονται supersteps, με το κάθε superstep να χωρίζεται περαιτέρω σε τρία στάδια, αποτελούμενα από:

1. **Σύγχρονος υπολογισμός** ο κάθε επεξεργαστής μπορεί να εκτελέσει τοπικούς υπολογισμούς, π.χ. ή κάθε διεργασία μπορεί να χρησιμοποιήσει τις τιμές που βρίσκονται στη γρήγορη τοπική μνήμη του επεξεργαστή. Οι υπολογισμοί γίνονται ασύγχρονα και μπορεί να επικαλύπτονται με την επικοινωνία.

2. **Επικοινωνία** Οι διεργασίες ανταλάσσουν δεδομένα μεταξύ τους για τη διευκόλυνση της δυνατότητας απομακρυσμένης αποθήκευσης δεδομένων. απομακρυσμένης αποθήκευσης δεδομένων.

3. **Εμπόδιο συγχρονισμού(synchronization barrier)** Όταν μία διεργασία φτάσει στο synchronization barrier, περιμένει μέχρι όλες οι υπόλοιπες διεργασίες να φτάσουν σ'αυτό.

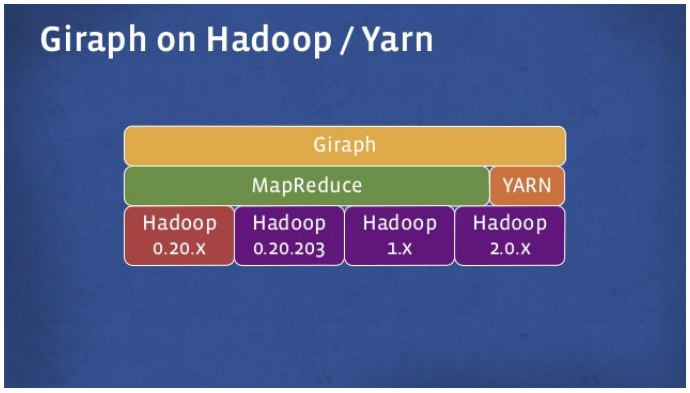
Η απεικόνιση ενός superstep βρίσκεται στο σχήμα 2.6.



Σχήμα 2.6: Superstep [3]

2.3.2 Τρόπος λειτουργίας του Giraph

Το **Apache Giraph** είναι ένα προγραμματιστικό πλαίσιο για επαναληπτική επεξεργασία γράφων. Αποτελεί μία χαλαρή υλοποίηση του Pregel της Google και χρησιμοποιεί την MapReduce υλοποίηση του Apache Hadoop ώστε να επεξεργάζεται γράφους, όπως βλέπουμε στο σχήμα 2.7.



Σχήμα 2.7: Χρήση του MapReduce από το Giraph [4]

Διαισθητικά, όταν σχεδιάζουμε έναν αλγόριθμο στο Giraph, πρέπει να ακολουθούμε τη λογική 'Σκέψου σαν κορυφή' ('Think Like A Vertex'). Ο αλγόριθμός μας εκτελείται από κάθε κορυφή του γράφου, η οποία γνωρίζει τη δική της κατάσταση, τους γείτονές της, μπορεί να στείλει μηνύματα σε

οποιαδήποτε άλλη κορυφή στο γράφο χρησιμοποιώντας το BSP προγραμματιστικό μοντέλο, μπορεί να δηλώσει πότε τελείωσε και να αλλάξει την τοπολογία του γράφου.

Η είσοδος στο Giraph είναι ένας γράφος που αποτελείται από κορυφές και ακμές, κάθε μία από τις οποίες έχει τη δική της τιμή. Επομένως η είσοδος δεν καθορίζει μόνο την τοπολογία του γράφου αλλά και τις αρχικές τιμές των κορυφών και των ακμών.

Στο Giraph, τα συστήματα επεξεργασίας γράφων εκφράζονται ως μια ακολουθία επαναλήψεων που ονομάζονται supersteps. Κατά της διάρκεια ενός superstep, το σύστημα ξεκινάει την προκαθορισμένη από το χρήστη συνάρτηση compute() για την κάθε κορυφή. Οι προκαθορισμένες συναρτήσεις από το χρήστη καθορίζουν τη συμπεριφορά μια μοναδικής κορυφής V και ενός μοναδικού superstep S.

Η μέθοδος compute:

- λαμβάνει και διαβάζει τα μηνύματα που στάλθηκαν στην κορυφή κατά το προηγούμενο superstep S-1.
- κάνει υπολογισμούς χρησιμοποιώντας τα μηνύματα, και τις τιμές των κορυφών και των εξερχόμενων ακμών, πράγμα που μπορεί να να οδηγήσει σε αλλαγές στις τιμές, και
- μπορεί να στείλει μηνύματα σε άλλες κορυφές, τα οποία λαμβάνονται από αυτές στο επόμενο superstep S+1.

Η μέθοδος compute δεν έχει άμεση πρόσβαση στις τιμές των άλλων κορυφών και των εξερχόμενων ακμών τους. Η επικοινωνία μεταξύ των κορυφών γίνεται στέλνοντας μηνύματα. Τα μηνύματα αυτά συνήθως στέλνονται κατά μήκος των εξερχόμενων ακμών, αλλά το πρόγραμμα μπορεί να στείλει ένα μήνυμα σε κάθε κορυφή με γνωστό αναγνωριστικό (VertexID). Κατά κανόνα, κάθε superstep αντιπροσωπεύει ατομικές μονάδες παράλληλου υπολογισμού.

Μεταξύ δύο συνεχόμενων supersteps παρεμβάλλεται πάντα ένα εμπόδιο συγχρονισμού (synchronization barrier): Με αυτό εννοούμε ότι:

1. τα μηνύματα που στάλθηκαν στο παρόν superstep θα φτάσουν στον προορισμό τους στο επόμενο superstep, και
2. οι κορυφές μπορούν να αρχίσουν τον υπολογισμό του επόμενου superstep, μόνο εφόσον κάθε ακμή έχει ολοκληρώσει τον υπολογισμό του τωρινού superstep.

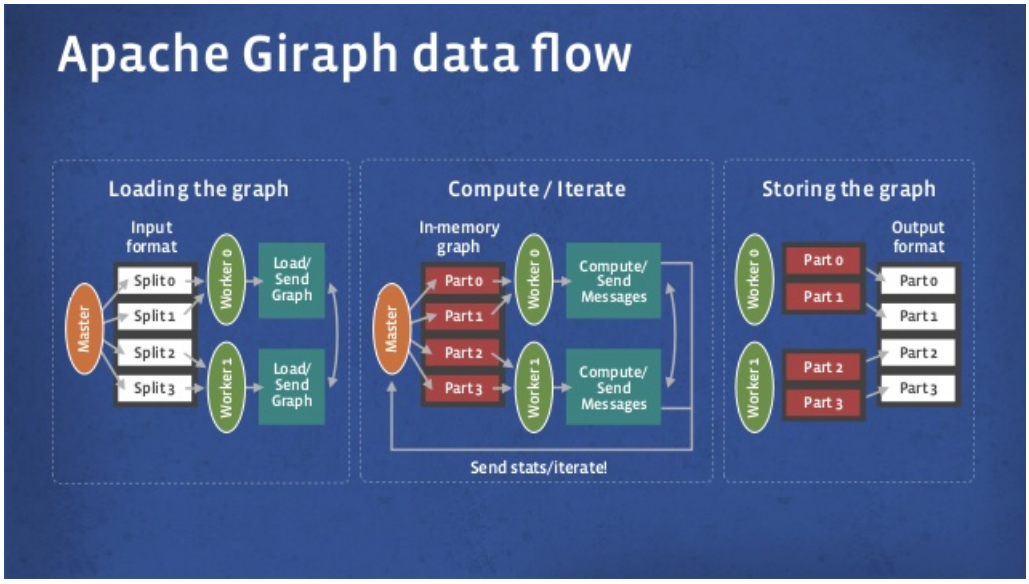
Ο γράφος μπορεί να αλλάξει κατά τη διάρκεια του υπολογισμού με την προσθήκη ή την αφαίρεση κορυφών ή ακμών. Οι τιμές διατηρούνται ανάμεσα στα σημεία συγχρονισμού. Αυτό σημαίνει, ότι η τιμή κάθε κορυφής ή ακμής κατά το ξενκίνημα ενός superstep είναι η ίδια με την αντίστοιχη τιμή στο τέλος του προηγούμενου superstep, όταν η τοπολογία του γράφου δεν έχει αλλαχθεί.

Το Giraph εφαρμόζει το μοντέλο Master-Slave όπου ο κόμβος που παίζει το ρόλο του Master αναθέτει τμήματα της εισόδου στους κόμβους που παίζουν το ρόλο των Slaves, συντονίζει το συγχρονισμό, ζητά σημεία ελέγχου, συναθροίζει τις τιμές των συναθροιστών (aggregators) και συλλέγει καταστάσεις υγείας (health statuses). Για το συγχρονισμό χρησιμοποιείται ο Apache Zookeeper [23]. Γενικά, τα προγράμματα Giraph τρέχουν ως εργασίες Hadoop χωρίς τη φάση reduce. Συγκεκριμένα, το Giraph αξιοποιεί το κομμάτι προγραμματισμού εργασιών του Hadoop με το να τρέχει Slaves ως ειδικού mappers, οι οποίοι επικοινωνούν μεταξύ τους για να παραδώσουν μηνύματα μεταξώ των κορυφών και να συγχρονιστούν μεταξύ των supersteps.

Κατά την εκτέλεση του προγράμματος, το σύνολο των κορυφών του γράφου, χωρίζεται σε επιμέρους τμήματα, που μοιράζονται στους εργάτες (workers). Ο προκαθορισμένος μηχανισμός χωρισμού (partioning) είναι μια συνάρτηση κατακερματισμού, όμως υπάρχει η δυνατότητα να χρησιμοποιηθεί

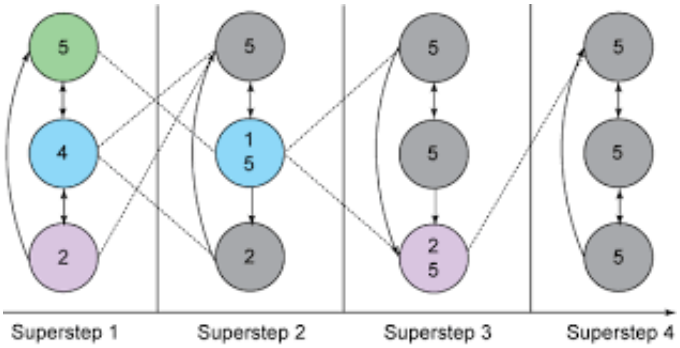
και προκαθορισμένη από το χρήστη συνάρτηση. Στο πρώτο superstep του προγράμματος που εκτελείται, όλες οι κορυφές είναι ενεργές. Η κάθε κορυφή μπορεί να απενεργοποιήσει τον εαυτό της με το να ψηφίσει να σταματήσει (vote to halt) και παραμένει ανερνεργή σε κάθε superstep που ακολουθεί, εκτός κι αν λάβει κάποιο μήνυμα. Αυτή η διαδικασία συνεχίζεται μέχρι όλες οι κορυφές να μην έχουν μηνύματα να στείλουν και να γίνουν ανενεργές, οπότε και σταματά η εκτέλεση του προγράμματος. Η κάθε κορυφή έχει ως έξοδο κάποιο τοπικό υπολογισμό, ο οποίος συνήθως είναι η τελική τιμή της ίδιας της κορυφής. Η κάθε μηχανή που κάνει κάποιο υπολογισμό, διατηρεί τις ακμές και τις κορυφές στη μνήμη και χρησιμοποιεί μεταφορές δικτύου μόνο για τα μηνύματα. Συνεπώς, το μοντέλο αυτό ταιριάζει απόλυτα σε κατανεμημένες υλοποιήσεις, επειδή δεν περιλαμβάνει κάποιο μηχανισμό που να εντοπίζει τη σειρά της εκτέλεσης κατά της διάρκεια ενός superstep, και όλη η επικοινωνία γίνεται από το superstep S στο superstep S+1.

Η ροή δεδομένν στο Apache Giraph απεικονίζεται στο σχήμα 2.8:



Σχήμα 2.8: Ροή Δεδομένων στο Apache Giraph [4]

Στο σχήμα 2.9, μπορούμε να δούμε πώς γίνεται ο υπολογισμός της μέγιστης τιμής στο Giraph. Οι διακεκομμένες γραμμές είναι τα μηνύματα και οι σκιασμένες κορυφές είναι ανενεργές. Αρχικά, όλες οι κορυφές είναι ενεργές και η καθεμία στέλνει την τιμής της στους γείτονές της. Στο επόμενο superstep, η κάθε κορυφή υπολογίζει τη μέγιστη τιμή με το να συγκρίνει τις τιμές από τα εισερχόμενα μηνύματα με τη δική της. Στην περίπτωση που η δική της τιμή είναι η μεγαλύτερη, σταματάει. Διαφορετικά, αντικαθιστά την τιμή της με τη μέγιστη τιμή που έχει βρεθεί μέχρι τώρα, και τη διαδίδει στους γείτονές της. Η ίδια διαδικασία συνεχίζεται μέχρι όλες οι κορυφές να γίνουν ανενεργές.



Σχήμα 2.9: Παράδειγμα Εκτέλεσης στο Giraph - Υπολογισμός Μέγιστης Τιμής [5]

2.3.3 Λόγος Προτίμησης του Giraph

Υπάρχουν πολλοί λόγοι που προτιμήσαμε το Giraph έναντι των άλλων συστημάτων για επξεργασία γράφων, Αρχικά, σε αντίθεση με το σύστημα Pregel, είναι ανοιχτού κώδικα και ταυτόχρονα είναι καλά προσαρμοσμένο για επεξεργασία γράφων. Οι μεμονωμένοι υπολογισμοί γίνονται στη μνήμη, διατηρεί την κατάστασή του, και στέλνονται μόνο οι ενδιάμεσες τιμές. Ο δίσκος χρησιμοποιείται μόνο κατά τη διάρκεια της εισόδου, της εξόδου και των σημείων ελέγχου (checkpoints), σε αντίθεση με κάποιο MapReduce πρόγραμμα, όπου απαιτείται πρόσβαση σε αυτόν πριν και μετά από κάθε map και reduce task. Επιπλέον, σε περίπτωση που η απαιτούμενη μνήμη ξεπεράσει τα όρια του συστήματος, μπορεί να δουλέψει και με τη χρήση του δίσκου, ώστε να μην αποτύχει ο υπολογισμός, έχοντας όμως χαμηλότερη απόδοση. Επίσης, είναι από τα λίγα συστήματα επεξεργασίας γράφων που υποστηρίζουν αλλαγή της τοπολογίας του γράφου (graph mutation).

Εκτός από τα προαναφερθέντα χαρακτηριστικά του Giraph, ένα πολύ μεγάλο πλεονέκτημα είναι ότι είναι πλήρως ενσωματωμένο με το Hadoop. Για την ακρίβεια, αξιοποιεί το 100% της υπάρχουσας υποδομής του Hadoop, με αποτέλεσμα να επωφελείται και από τα πλεονεκτήματα που προέρχονται από τη χρήση του, όπως η ανοχή σε σφάλματα υλικού και η κλιμακωσιμότητα. Ακόμη, η επικονωνία που είναι βασισμένη στα μηνύματα, οδηγεί στο να μη χρειαζόμαστε locks, ενώ το synchronization barrier καθιστά μη αναγκαία τη χρήση σημαφόρων. Τέλος, είναι πολύ πιο απλό από τα ασύγχρονα μοντέλα, ένας πολύ σημαντικός παράγοντας για τον εντοπισμό και την αντιμετώπιση σφαλμάτων σε μεγάλες συστοιχίες υπολογιστών.

Κεφάλαιο 3

Κατανεμημένη επεξεργασία ερωτημάτων σε Διασυνδεδεμένα Δεδομένα

Όπως αναφέραμε στο κεφάλαιο 1, ο κύριος σκοπός αυτής της διπλωματικής εργασίας είναι να αξιοποιήσουμε την ικανότητα των κατανεμημένων συστημάτων για γρήγορη επεξεργασία γράφων, να εισάγουμε μεθόδους για την επίλυση διαφόρων ειδών ερωτημάτων πάνω σε δεδομένα γράφων σε τέτοια συστήματα (π.χ. Apache Giraph) και να μελετήσουμε την απόδοση της κατανεμημένης επεξεργασίας τέτοιων ερωτημάτων.

Σε αυτό το κεφάλαιο, θα περιγράψουμε εν συντομία τί είναι τα Διασυνδεδεμένα Δεδομένα, τους τρόπους που ενδείκνυνται για τη δημοσίευσή τους, το μοντέλο που χρησιμοποιείται για την περιγραφή τους, τους υπάρχοντες τρόπους σειριοποίησης και τη γλώσσα ερωτημάτων SPARQL. Στη συνέχεια, θα αναφέρουμε ποια σύνολα δεδομένων χρησιμοποιήσαμε για τα ερωτήματά μας και θα επικεντρωθούμε στην εισαγωγή αποδοτικών μεθοδολογιών για την επίλυση αντιπροσωπευτικών τύπων ερωτημάτων πάνω σε τέτοια σύνολα δεδομένων, όπως συναντώνται στη βιβλιογραφία. Συγκεκριμένα, θα παρουσιάσουμε τους βασικούς παράγοντες που λάβαμε υπόψη μας, καθώς και μετρικές απόδοσης. Τέλος, θα παρουσιάσουμε το τρόπο με τον οποίο οι επιλεγμένοι τύποι ερωτημάτων λύθηκαν με κατανεμημένο τρόπο στο Apache Giraph και τη μεθοδολογία που προτείνεται για την επίλυσή τους.

3.1 Διασυνδεδεμένα Δεδομένα

3.1.1 Χρησιμότητα των Διασυνδεδεμένων Δεδομένων

Ζούμε σε ένα κόσμο που συνεχώς παράγει ένα τεράστιο όγκο δεδομένων. Ένα μεγάλο, αν όχι το μεγαλύτερο μέρος αυτών των δεδομένων, παράγεται μέσω του διαδικτύου. Τα δεδομένα αυτά δεν είναι χρήσιμα αν τα αντιμετωπίζουμε ως απομονωμένες νησίδες πληροφορίας, αλλά μπορεί να είναι πολύ σημαντικά όταν βρούμε τις σχέσεις που υπάρχουν μεταξύ τους. Αυτό μας επιτρέπει να εξάγουμε χρήσιμη πληροφορία και να βγάλουμε σημαντικά συμπεράσματα, στα οποία δε θα μπορούσαμε να είχαμε καταλήξει διαφορετικά.

Τα Διασυνδεδεμένα Δεδομένα υπάρχουν γι'αυτόν ακριβώς το σκοπό: το να χρησιμοποιούμε το διαδίκτυο για να συνδέσουμε σχετικά δεδομένα που δε συνδέονταν πριν. Αυτό μπορεί να επιτευχθεί με το να δημοσιεύουμε τα δομημένα δεδομένα χρησιμοποιώντας τεχνολογίες όπως το Hypertext Transfer Protocol (HTTP), το Resource Description Framework (RDF) και μοναδικά αναγνωριστικά πόρων (uniform resource identifiers, URIs), έτσι ώστε τα δεδομένα να μπορούν να διασυνδέονται και να γίνονται πιο χρήσιμα μέσω σημασιολογικών ερωτημάτων [24].

Σε αυτό το σημείο, προκύπτει το εξής προφανές ερώτημα: Γιατί χρειαζόμαστε τα Διασυνδεδεμένα δεδομένα και δεν προσπαθούμε να χρησιμοποιήσουμε ή να συνδυάσουμε έναν ή περισσότερους από τους υπάρχοντες μηχανισμούς για τη δημοσίευση και επαναχρησιμοποίηση των δεδομένων στο διαδίκτυο? Αρχικά, είναι σημαντικό τα δεδομένα αυτά να είναι ως ένα βαθμό δομημένα. Όσο πιο δομημένη

είναι η μορφή τους, τόσο πιο εύκολη γίνεται η δημιουργία εργαλείων για την επεξεργασία τους. Σήμερα, δομημένα δεδομένα γίνονται διαθέσιμα στο διαδίκτυο με διάφορους τρόπους και κατά συνέπεια, έχουν αναπτυχθεί διάφορες προσεγγίσεις για την ανακάλυψη, την ανάκτηση και την επεξεργασία των διαθέσιμων δεδομένων στο διαδίκτυο. Οι πιο δημοφιλείς, είναι μέσω της χρήσης HTML ή διαφόρων Web εφαρμογών. Όμως, στην πρώτη περίπτωση υπάρχει περιορισμός στις σχέσεις και στις οντότητες που μπορούν να περιγραφούν, ενώ στη δεύτερη απαιτείται τεράστια προσπάθεια από τον προγραμματιστή για να ενοποιήσει τα δεδομένα από ένα τεράστιο αριθμό αγνώστων πηγών δεδομένων και επιπλέον δεν είναι δυνατό να υπάρξει η κατάλληλη διασύνδεση των δεδομένων ώστε να μπορούν να εντοπιστούν. Τα Διασυνδεδεμένα Δεδομένα, από την άλλη πλευρά, παρέχουν μηχανισμούς ώστε να ξεπεραστούν όλα τα παραπάνω προβλήματα.

3.1.2 Οι αρχές των Διασυνδεδεμένων Δεδομένων

Ο όρος Διασυνδεδεμένα Δεδομένα(Linked Data) αναφέρεται σε ένα σύνολο βέλτιστων πρακτικών για τη δημοσίευση και τη διασύνδεση δομημένων δεδομένων στο διαδίκτυο. Αυτές οι τεχνικές εισήχθησαν από τον Tim Berners-Lee, διευθυνή της Κοινοπραξίας του Παγκόσμιου Ιστού (World Wide Web Consortium - W3C), στην αρχιτεκτονική του σημείωση 'Linked Data'[25] το 2016 και μπορούν να παραφραστούν ως εξής:

1. Χρησιμοποιήστε URIs ως ονόματα για τα πράγματα.
2. Χρησιμοποήστε HTTP URIs έτσι ώστε τα ονόματα αυτά να μπορούν να ερμηνευτούν.
3. Να παρέχετε χρήσιμη πληροφορία για το τί προσδιορίζει το κάθε όνομα, χρησιμοποιώντας πρότυπα όπως το RDF, SPARQL και άλλα.
4. Να αναφέρεστε σε άλλα πράγματα χρησιμοποιώντας τα ονόματα, που είναι βασισμένα στο HTTP URI τους, ώστε περισσότερα πράγματα να μπορούν να ανακαλυφθούν.

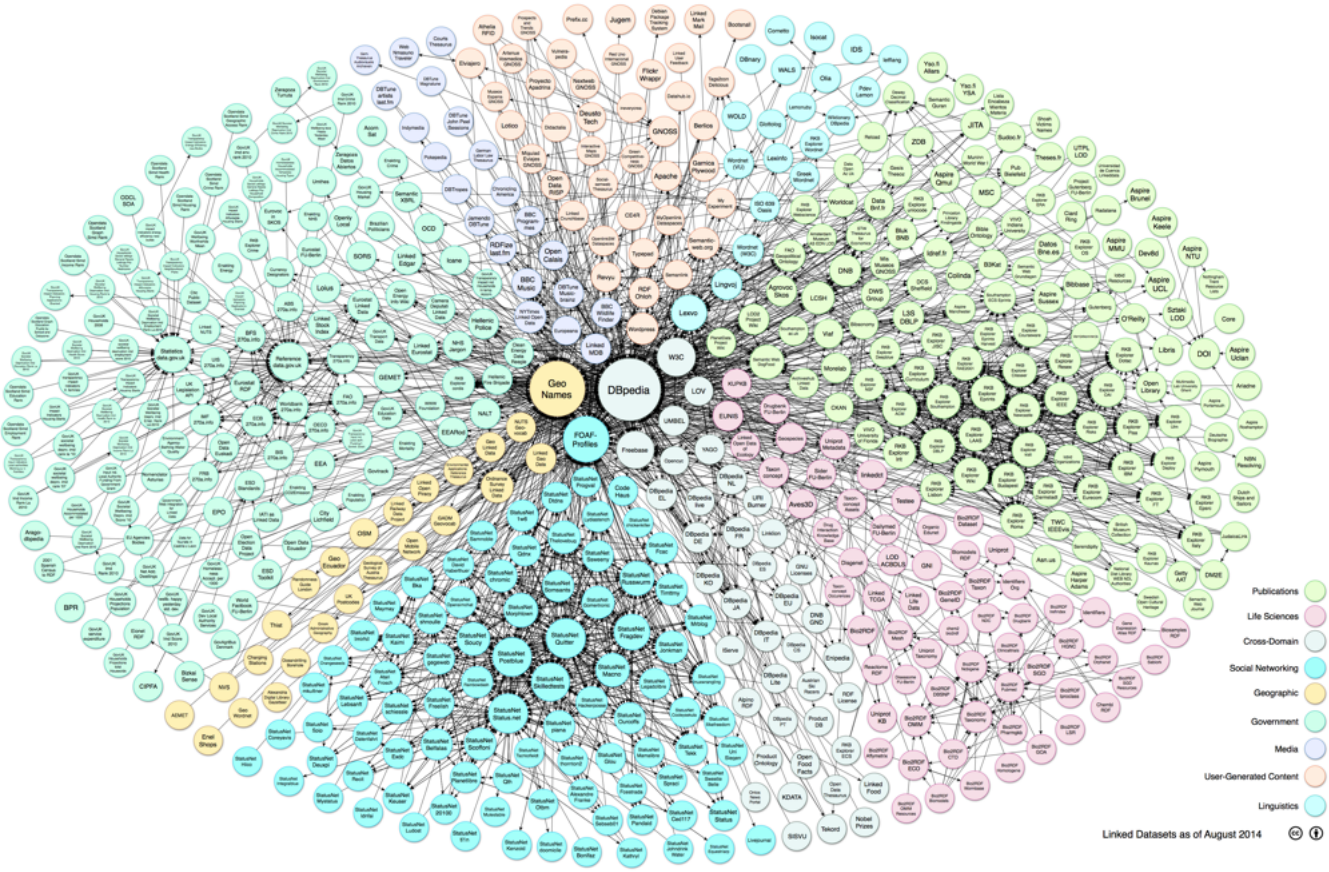
Μια ειδική κατηγορία Διασυνδεδεμένων Δεδομένων είναι τα Ανοικτά Διασυνδεδεμένα Δεδομένα (Linked Open Data), το περιεχόμενο των οποίων είναι ανοικτό. Τέτοιου είδους βάσεις γνώσεων αποτελούν η DBpedia και η Freebase.

Η εικόνα 3.1 απεικονίζει την κλίμακα των Διασυνδεδμένων Δεδομένων που προέρχονται από την κοινότητα Linking Open Data και ταξινομεί τα σύνολα δεδομένων ανά τοπικά περδία τονίζοντας την ποικιλία των συνόλων δεδομένων που είναι παρόντα στον παγκόσμιο ιστό (Web of Data). Το γράφημα που φαίνεται σε αυτήν την εικόνα είναι διαθέσιμο στο `http://lod-cloud.net`. Βελτιωμένες εκδόσεις δημοσιεύονται στο website ανά τακτά χρονικά διαστήματα.

3.1.3 RDF

Όπως αναφέρθηκε, το να διασυνδέεις δεδομένα που είναι κατανεμημένα στο διαδίκτυο απαιτεί έναν πρότυπο μηχανισμό που έχει ως στόχο να καθορίσει την ύπαρξη και το νόημα των συνδέσεων μεταξύ των αντικειμένων που περιγράφονται σε αυτά τα δεδομένα. Αυτός ο μηχανισμός παρέχεται από το Resource Description Mechanism (RDF). Ο μηχανισμός αυτός διακρίνεται για την απλότητά του, την ικανότητα να εκφράσει πολύπλοκα σχήματα και σχέσεις καθώς και για την καταλληλότητα να μοντελοποιήσει όλα τα εξωτερικά συστήματα δεδομένων για αδόμητα, ημι-δομημένα και δομημένα δεδομένα.

Στο RDF, η περιγραφή ενός πόρου επιτυγχάνεται μέσα από ένα αριθμό τριπλετών. Τα τρία κομμάτια της κάθε τριπλέτας ονομάζονται υποκείμενο, κατηγορούμενο και αντικείμενο. Μία τριπλέτα αντικατροπτίζει τη βασική δομή μιας απλής πρότασης, όπως για παράδειγμα η παρακάτω: 'Η Βασιλική έχει



Σχήμα 3.1: LOD Cloud 2014 [6]

ψευδώνυμο Βάσω'. όπου 'Η Βασιλική' είναι το υποκείμενο, 'έχει ψευδώνυμο είναι το κατηγορούμενο' και 'Βάσω' είναι το αντικείμενο.

Ένας τρόπος να σκεφτεί κανείς ένα σύνολο από RDF τριπλέτες είναι ως έναν RDF γράφο. Τα URIs, που εμφανίζονται ως υποκείμενο και αντικείμενο, είναι οι κορυφές του γράφου, και η κάθε τριπλέτα είναι μία κατευθυνόμενη ακμή που συνδέει το υποκείμενο με το αντικείμενο. Καθώς τα URIs των Διασυνδεδεμένων Δεδομένων είναι καθολικά μοναδικά και μπορούν να αναζητηθούν σε ένα σύνολο από RDF τριπλέτες, μπορούμε να φανταστούμε όλα τα Διασυνδεδεμένα Δεδομένα ως ένα καθολικό γράφο, όπως προτείνεται από τον Tim Berners-Lee. Οι εφαρμογές για Διασυνδεδεμένα Δεδομένα κάνουν ενέργειες πάνω σε αυτό το γιγαντιαίο γράφο και ανακτούν κομμάτια του με το να αναζητούν URIs κατ'απαίτηση.

RDF - Τρόποι Σειριοποίησης

Το RDF είναι ένα μοντέλο για την περιγραφή των διασυνδεδεμένων δεδομένων στη μορφή τριπλετών: υποκείμενο, κατηγορούμενο, αντικείμενο. Για να δημοσιεύσουμε ένα RDF γράφο στο διαδίκτυο, πρέπει πρώτα να τον σειριοποιήσουμε χρησιμοποιώντας κάποιο RDF συντακτικό. Αυτό απλά σημαίνει ότι παίρνουμε τις τριπλέτες που αποτελούν τον RDF γράφο και χρησιμοποιούμε συγκεκριμένο συντακτικό για να τις γράψουμε σε ένα αρχείο. Δύο είδη σειριοποίησης έχουν προτυποποιηθεί από το W3C, το RDF/XML και RDFa. Όμως για την κάλυψη συγκεκριμένων αναγκών έχουν δημιουργηθεί πολλές άλλες μορφές σειριοποίησης.

Στα δικά μας πειράματα εμείς χρησιμοποιήσαμε σύνολα δεδομένων σε N-Triples μορφή. Είναι μία απλή μορφή κειμένου, όπου όλα τα URIs καθορίζονται πλήρως σε κάθε τριπλέτα. Συνεπώς τα αρ-

χεία που περιέχουν N-Triples μπορεί να είναι πολύ μεγαλύτερα από αυτά όπου χρησιμοποιούνται άλλες μορφές σειριοποίησης, καθώς σε αυτές γίνεται χρήση προθεμάτων και άλλων ειδών συντομογραφίας. Όμως ο πλεονασμός αποτελεί ταυτόχρονα και το κυριότερο πλεονέκτημα αυτής της μορφής N-Triples, σε σχέση με τις άλλες μορφές, καθώς επιτρέπει να γίνει η προσπέλαση αυτών των αρχείων ανά μία γραμμή τη φορά, καθιστώντας τα ιδανικά για τη φόρτωση πολλών αρχείων δεδομένων τα οποία δε θα χωρέσουν στην κύρια μνήμη. Επίσης λόγω αυτού του πλεονασμού, τα αρχεία αυτά επιδέχονται συμπίεση, μειώνοντας έτσι την κίνηση στο δίκτυο όταν ανταλλάσσονται αρχεία.

Ένα αντιπροσωπευτικό παράδειγμα φαίνεται παρακάτω:

```
1 <http://biglynx.co.uk/people/dave-smith> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
2 <http://biglynx.co.uk/people/dave-smith> <http://xmlns.com/foaf/0.1/name> 'Dave Smith' .
```

3.1.4 SPARQL

Καθώς το μέγεθος των Διασυνδεδεμένων Δεδομένων αυξάνεται σημαντικά χρόνο με το χρόνο, είναι απαραίτητη η χρήση μιας γλώσσας επερώτησης ώστε να ανακτήσουμε την επιθυμητή πληροφορία. Η SPARQL(SPARQL Protocol and RDF Query Language), είναι η βασική γλώσσα για ερωτήματα πάνω σε Διασυνδεδεμένα Δεδομένα, ικανή να ανακτήσει και να διαχειριστεί δεδομένα σε μορφή RDF. Μοιάζει πολύ με την SQL, αλλά εφαρμόζεται σε ένα γράφο RDF.

3.1.5 DBpedia

Μία κλασική περίπτωση ενός μεγάλου συνόλου Διασυνδεδεμένων Δεδομένων είναι η DBpedia[26], σκοπός της οποίας είναι να εξάγει δομημένο περιεχόμενο από την πληροφορία που δημιουργείται ως τμήμα της Wikipedia, και κάνει αυτό το περιεχόμενο διαθέσιμο στον Παγκόσμιο Ιστό (World Wide Web). Είναι μόνο ένα από τα πιο δημοφιλή κομμάτα της αποκεντρωμένης προσπάθειας για Linked Data, καθώς περιλαμβάνει όχι μόνο δεδομένα από τη Wikipedia, αλλά ενσωματώνει και συνδέσμους σε άλλα σύνολα δεδομένων στο διαδίκτυο, όπως τα Geonames.

Εμείς στην εργασία αυτή χρησιμοποιήσαμε RDF dumps σε N-Triple μορφή σειριοποίησης, που είναι διαθέσιμα για λήψη στο site της DBpedia.

Η βάση γνώσεων της DBpedia αναπτύσσεται συνεχώς, καθώς μεταβάλλεται κάθε φορά που αλλάζει το περιεχόμενο της Wikipedia, καλύπτει ένα ευρύ φάσμα από διαφορετικά πεδία και συνδέει οντότητες μεταξύ αυτών των πεδίων. Έτσι, ένα πλούσιο σώμα από διάφορες πηγές γνώσης μπορεί να αποκτηθεί μέσω της συνεργασίας ενός τεράστιου αριθμού χρηστών, των χρηστών της Wikipedia, οι οποίοι δεν έχουν καν γνώση ότι συνεισφέρουν σε μια δομημένη βάση γνώσεων. Επομένως, η DBpedia είναι κατάλληλη και για τους σκοπούς αυτής της διπλωματικής εργασίας, καθώς μας επιτρέπει να χρησιμοποιήσουμε το Giraph για να κάνουμε ερωτήματα πάνω σε δεδομένα γράφων με πλούσιο περιεχόμενο, που περιλαμβάνουν διάφορα πεδία και διαφόρων ειδών σχέσεις μεταξύ των οντοτήτων. Επιπλέον, το γεγονός ότι το μέγεθός της αυξάνεται συνεχώς, μας επιτρέπει να χρησιμοποιήσουμε σύνολα δεδομένων διαφορετικού μεγέθους, ώστε να μελετήσουμε την κλιμακωσιμότητα του συστήματός μας.

3.2 Επιλογή των ερωτημάτων

Για να αξιολογήσουμε την απόδοση του συστήματός μας για κατανεμημένη επεξεργασία σε σύγκριση με κεντρικές βάσεις για RDF δεδομένα, επιλέξαμε να χρησιμοποιήσουμε ένα σύνολο αντιπροσωπευ-

τικών ερωτημάτων με διαφορετικά χαρακτηριστικά σύμφωνα με τα κριτήρια που ακολουθούν. Ο στόχος είναι να υλοποιήσουμε δημοφιλείς τύπους ερωτημάτων με κατανεμημένο τρόπο και να ορίσουμε το προφίλ των ερωτημάτων που μπορούν να επωφεληθούν περισσότερο από την κατανεμημένη επεξεργασία.

1. **Μέγεθος Εισόδου**. Αυτό μετράται ως η αναλογία των στιγμιοτύπων των κλάσεων που περιλαμβάνονται στο ερώτημα ως προς το συνολικό αριθμό των στιγμιοτύπων των κλάσεων για ένα δεδομένο σύνολο δεδομένων. Εδώ αναφερόμαστε όχι μόνο στα στιγμιότυπα των κλάσεων που εμπεριέχονται ρητά στο ερώτημα αλλά και αυτά που συνεπάγονται από τη βάση γνώσεων.

2. **Επιλεκτικότητα**. Αυτό μετράται ως η εκτιμώμενη αναλογία των στιγμιοτύπων που περιλαμβάνονται στο ερώτημα και ικανοποιούν τα κριτήρια του ερωτήματος. Προφανώς, το αν η επιλεκτικότητα είναι υψηλή ή χαμηλή εξαρτάται από το σύνολο δεδομένων που χρησιμοποιείται.

3. **Πολυπλοκότητα**. Χρησιμοποιούμε τον αριθμό των κλάσεων και των ιδιοτήτων οι οποίες περιλαμβάνονται στο ερώτημα ως ένδειξη της πολυπλοκότητας. Όμως ο πραγματικός βαθμός πολυπλοκότητας στην πράξη μπορεί να διαφέρει ανάλογα με τα συστήματα ή τα σχήματα που χρησιμοποιούνται.

Επομένως, έχουμε επιλέξει ενδεικτικά ερωτήματα τα οποία καλύπτουν ένα φάσμα από ιδιότητες με βάση τα παραπάνω κριτήρια. Ταυτόχρονα έχουμε δώσει έμφαση σε ερωτήματα με μεγάλο μέγεθος εισόδου και υψηλή επιλεκτικότητα. Κατά το σχεδιασμό των ερωτημάτων έχουμε λάβει υπόψη μας και κάποιους άλλους παράγοντες, όπως για παράδειγμα τον τρόπο με τον οποίο οι κλάσεις και οι ιδιότητες συνδέονται μαζί στο ερώτημα.

3.3 Μετρικές απόδοσης και επιλεγμένα ερωτήματα

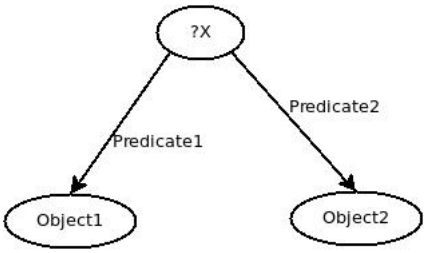
Θεωρούμε ένα σύνολο από μετρικές απόδοσης:

1. Χρόνο φόρτωσης
2. Μέγεθος αποθετηρίου (repository)
3. Χρόνος Απόκρισης Ερωτήματος
4. Πληρότητα και Ορθρότητα του Ερωτήματος

Από τις παραπάνω μετρικές, οι πρώτες τρεις είναι πρότυπες για κάθε database benchmark. Την τελευταία την εισάγαμε εμείς για τους σκοπούς των δικών μας πειραμάτων.

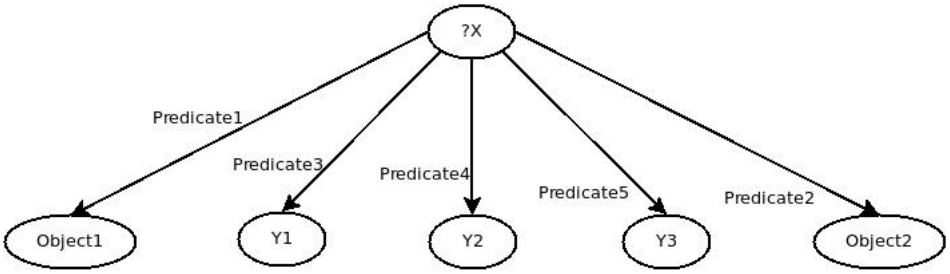
Επιλέξαμε να υλοποιήσουμε διαφορετικά ερωτήματα με βάση τα προαναφερθένα κριτήρια. Εδώ θα παρουσιάσουμε τα διαφορετικά είδη ερωτημάτων, δίνοντας περισσότερη έμφαση στην πολυπλοκότητά τους και τον τρόπο με τον οποίο οι κλάσεις και οι ιδιότητες συνδέονται μεταξύ τους. Τα ερωτήματα αυτά μπορούν να περιγραφτούν καλύτερα μέσω κατάλληλων σχημάτων, όπου το κάθε ερώτημα απεικονίζεται ως ένας γράφος. Οι κορυφές του γράφου αντιπροσωπεύουν τα υποκείμενα ή τα αντικείμενα που εμπλέκονται στο ερώτημα, ενώ οι ακμές του γράφου αντιπροσωπεύουν τα κατηγορούμενα που περιλαμβάνονται. Οι άλλοι δύο παράγοντες, το μέγεθος εισόδου και η επιλεκτικότητα, εξαρτώνται από τα σύνολα δεδομένων που χρησιμοποιήθηκαν και μπορεί να ποικίλουν ακόμα και για ερωτήματα με την ίδια γραφική αναπαράσταση. Για το σχεδιασμό των ερωτημάτων μας, καθώς και για τους διάφορους παράγοντες και τις μετρικές απόδοσης που λάβαμε υπόψη μας, χρησιμοποιήσαμε το Lehigh University Benchmark(LUBM) [27].

Οι απεικονίσεις των ερωτημάτων που υλοποιήσαμε φαίνονται στα σχήματα 3.2 - 3.8. Στα σχήματα αυτά, όπου χρησιμοποιουμε σύμβολα όπως Y, Z, Y1, κλπ, υποθέτουμε ότι μπορεί να έχουμε οποιαδήποτε κορυφή, η οποία συνδέεται με τους γείτονές της με τον τρόπο που φαίνεται στο σχήμα. Από την άλλη, σύμβολα της μορφής 'Object1', 'Predicate1', θεωρείται ότι είναι η ίδια η τιμή του αναγνωριστικού της κορυφής (VertexID) και της ακμής αντίστοιχα.



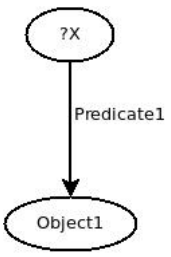
Σχήμα 3.2: Ερώτημα 1

Το **Ερώτημα 1** (Σχήμα 3.2) είναι πολύ απλό, ζητώντας ένα οποιοδήποτε υποκείμενο το οποίο έχει ως αντικείμενα τα 'Object1' και 'Object2', με 'Predicate1' και 'Predicate2' να είναι οι τιμές των εξερχόμενων ακμών που συνδέουν το υποκείμενο με τα προαναφερθέντα αντικείμενα αντίστοιχα.



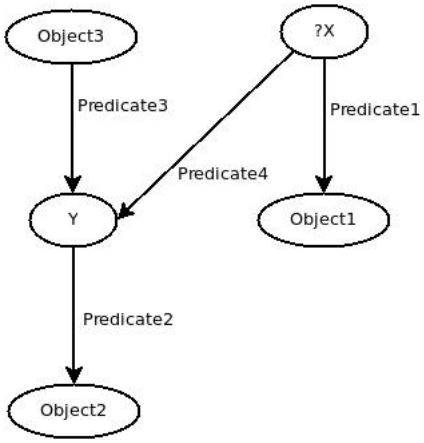
Σχήμα 3.3: Ερώτημα 2

Το **ερώτημα 2** (Σχήμα 3.3) περιλαμβάνει πολλές ιδιότητες για το ίδιο υποκείμενο. Μπορεί να θεωρηθεί ως επέκταση του Ερωτήματος 1, αφού έχει την ίδια μορφή, με τη διαφορά ότι το το επιθυμητό υποκείμενο-απάντηση, θα πρέπει να έχει τρεις επιπλέον εξερχόμενες ακμές με συγκεκριμένες τιμές, που να είναι συνδεμένες σε οποιοδήποτε αντικείμενο. Είναι προφανές ότι η πολυπλοκότητα αυξάνεται σε σχέση με το πρώτο ερώτημα, καθώς ο αριθμός των κλάσεων και των ιδιοτήτων που εμπεριέχονται είναι αρκετά μεγαλύτερος. Μια επιπλέον παρατήρηση είναι ότι ενώ στο διάγραμμα βλέπουμε ο αριθμός των κόμβων να έχει αυξηθεί μόνο κατά τρία, στην πράξη ο αριθμός αυτός είναι αρκετά μεγαλύτερος καθώς κάθε μία από αυτές τις τρεις κορυφές (Y1, Y2, Y3), στην ουσία αντιπροσωπεύει ένα σύνολο κορυφών, καθεμία από τις οποίες συνδέεται με το υποκείμενο με ακμή συγκεκριμένης τιμής.



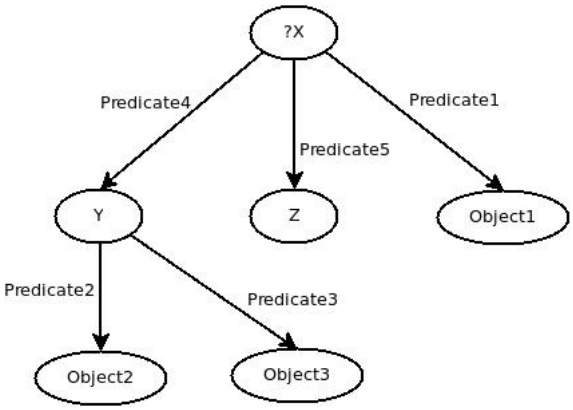
Σχήμα 3.4: Ερώτημα 3

Το **Ερώτημα 3** (Σχήμα 3.4) είναι το ερώτημα με τη μικρότερη πολυπλοκότητα. Το ερώτημα αυτό ζητάει οποιοδήποτε υποκείμενο συνδέεται με το αντικείμενο 'Object1', με την ακμή την οποία τα συνδέει να έχει την τιμή 'Predicate1'.



Σχήμα 3.5: Ερώτημα 4

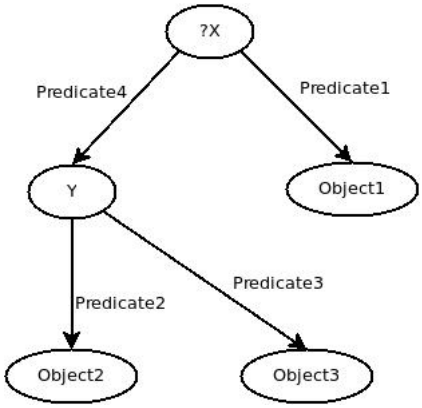
Το **ερώτημα 4** (Σχήμα 3.5) είναι ένα ακόμα ερώτημα με αυξημένο αριθμό κλάσεων και ιδιοτήτων, επομένως η πολυπλοκότητα και πιθανότατα η επιλεκτικότητα να είναι πολύ υψηλές. Συγκεκριμένα, τα υποκείμενα που αποτελούν μέρος της απάντησης πρέπει αν έχουν τουλάχιστον δύο εξερχόμενες ακμές. Μία με την τιμή 'Predicate1' και άλλη μια με την τιμή 'Predicate4', που να δείχουν στα αντικείμενα 'Object1' και Y αντίστοιχα. Το Y εδώ δεν αναφέρεται σε ένα συγκεκριμένο αντικείμενο, αλλά σε κάθε κορυφή, η οποία έχει μια εξερχόμενη ακμή με τιμή 'Predicate2' που να τη συνδέει με το 'Object2' και μία εισερχόμενη ακμή με τιμή 'Predicate3' από μια συγκεκριμένη κορυφή 'Object3'. Επαναλαμβάνουμε εδώ ότι το σύμβολο Y, αναφέρεται σε οποιαδήποτε κορυφή με τα επιθυμητά χαρακτηριστικά, έτσι ο αριθμός των κορυφών που περιλαμβάνονται στο ερώτημα μπορεί να ποικίλει ανάλογα με το σύνολο των δεδομένων που χρησιμοποιήσαμε.



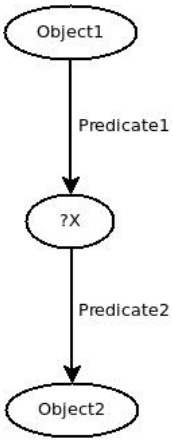
Σχήμα 3.6: Ερώτημα 5

Το **Ερώτημα 5** (Σχήμα 3.6) είναι ακόμα πιο πολύπλοκο από το ερώτημα 4, περιλαμβάνοντας μία ακόμα ιδιότητα και κλάση. Το υποκείμενο-απάντηση θα πρέπει να είναι συνδεμένο με τρία αντικείμενα, ένα συγκεκριμένο: το 'Object1' με ακμή τιμής 'Predicate1', και άλλα δύο Y και Z, τα οποία μπορεί να είναι οποιαδήποτε αντικείμενα έχουν κάποια συγκεκριμένα χαρακτηριστικά. Συγκεκριμένα, η κορυφή Y, θα πρεει να είναι συνδεμένη με δύο άλλες κορυφές: 'Object2' και 'Object3' με ακμές που έχουν τιμές: 'Predicate2' and 'Predicate3' αντίστοιχα. Η κορυφή Z μπορεί να είναι οποιοδήποτε αντικείμενο, με την προϋπόθεση ότι είναι συνδεμένο με το υποκείμενο-απάντηση με ακμή τιμής 'Predicate5'.

Το **Ερώτημα 6** (Σχήμα 3.7) είναι σχεδόν το ίδιο με το 5, αλλά λίγο λιγότερο πολύπλοκο, αφού περιλαμβάνει μία λιγότερη ιδιότητα και κλάση.



Σχήμα 3.7: Ερώτημα 6



Σχήμα 3.8: Ερώτημα 7

Το **Ερώτημα 7** (Σχήμα 3.8) είναι ένα από τα απλούστερα να μελετήσει κανείς. Ζητάει για μια κορυφή Y, η οποία έχει μια εισερχόμενη ακμή με τιμή 'Predicate1' από μια κορυφή 'Object1', και μια εξερχόμενη ακμή με τιμή 'Predicate2' προς το 'Object2'.

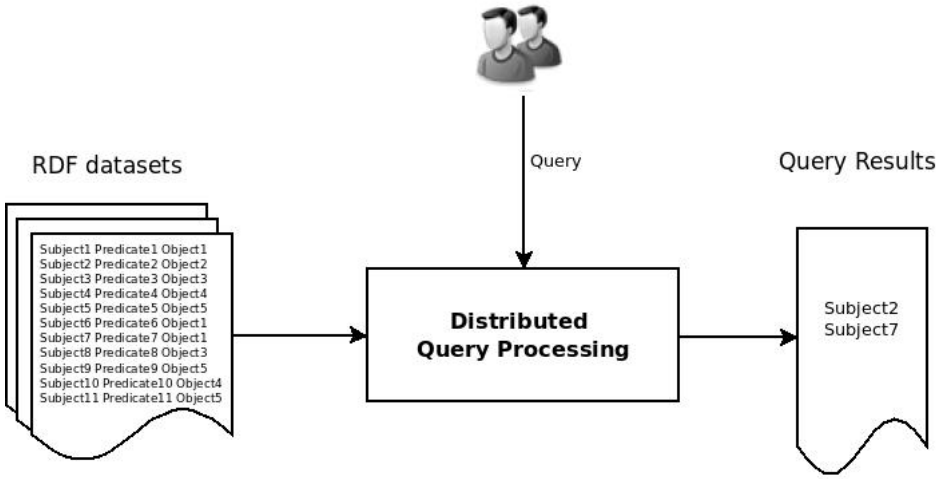
Μία διευκρίνηση είναι ότι κανένα από τα προαναφερθέντα ερωτήματα δεν υποθέτει πληροφορία για την ιεραρχία μεταξύ των κλάσεων (class hierarchy), ούτε απαιτεί κάποια λογική συνεπαγωγή (inference). Επιπλέον, είναι πορφανές ότι οι ετικέτες που χρησιμοποιήθηκαν στα σχήματα που περιγράφουν τα ερωτήματα, είναι απλά παραδείγματα ώστε να είναι πιο ξεκάθαρα και όχι πραγματικές. Στα πειράματά μας, αυτές οι ετικέτες (π.χ. 'Object1', 'Predicate1') παίρνουν πραγματικές τιμές, διαφορετικές κάθε φορά, ώστε να καταλήξουμε να έχουμε ένα σύνολο από 15 διαφορετικά ερωτήματα, διαφορετικής πολυπλοκότητας, μεγέθους εισόδου και επιλεκτικότητας. Εκτελέσαμε τα παραπάνω ερωτήματα σε ένα καταναμεμημένο σύστημα, το Giraph, αλλά και σε ένα κεντρικό, το Openvirtuoso, έτσι ώστε να μελετήσουμε πώς ποικίλει ο χρόνος εκτέλεσης με ερωτήματα διαφορετικών χαρακτηριστικών.

3.4 Εκτέλεση ερωτημάτων στο Apache Giraph

3.4.1 Το σύστημά μας

Όπως αναφέραμε νωρίτερα ο στόχος είναι να μπορέσει το σύστημά μας να απαντήσει τα προαναφερθέντα ερωτήματα με ένα κατανεμημένο τρόπο, αξιοποιώντας τη δύναμη των πολλαπλών εμπορικών υπολογιστικών κόμβων.

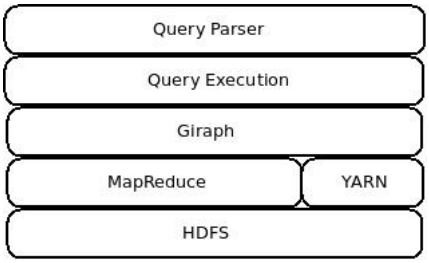
Η είσοδος είναι τα RDF σύνολα δεδομένων, ενώ η έξοδος θα είναι η απάντηση στα ερωτήματά μας, όπως φαίνεται στο σχήμα 3.9.



Σχήμα 3.9: Κατανεμημένο Σύστημα Επεξεργασίας Ερωτημάτων

Για την επεξεργασία των παραπάνω τύπων sparql ερωτημάτων, επιλέξαμε να χρησιμοποιήσουμε το Giraph, το οποίο μας επιτρέπει να βρούμε την απάντηση με έναν απλό τρόπο, αλλά ταυτόχρονα ικανό να αξιοποιήσει τη δύναμη ενός κατανεμημένου συστήματος. Μπορούμε να επικεντρωθούμε στον αλγόριθμο που σχεδιάζουμε χωρίς την ανάγκη να ανησυχούμε για τη γραφική αναπαράσταση του γράφου στη μνήμη, τον τρόπο με τον οποίο ο αλγόριθμός μας εκτελείται παράλληλα στο κατανεμημένο σύστημα και το πώς επιτυγχάνεται η ανοχή στα σφάλματα υλικού. Ο αλγόριθμός μας δεν έχει καμία γνώση του τρόπου με τον οποίο τα δεδομένα μοιράζονται στις διάφορες επεξεργαστικές μονάδες και τον τρόπο με τον οποίο ο κώδικας εκτελείται ταυτόχρονα. Έτσι, δεν υπάρχει καμία ανάγκη για κλειδώματα (locks) ή συγχρονισμό από την πλευρά μας.

Στο σχήμα 3.10 απεικονίζεται σε μορφή επιπέδων τα διάφορα μέρη του συστήματός μας.



Σχήμα 3.10: Πολυεπίπεδη Αναπαράσταση του Συστήματος Επεξεργασίας Ερωτημάτων

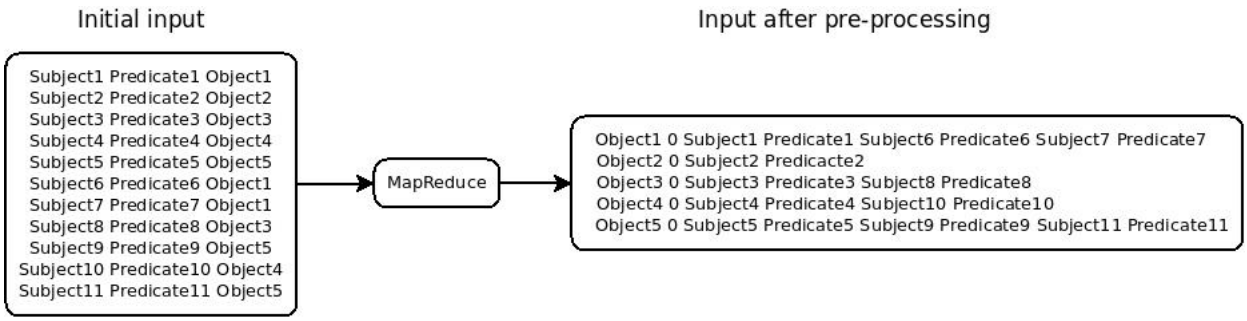
Η υλοποίησή μας αποτελείται από δύο τμήματα στην κορυφή της εικόνας. Συγκεκριμένα:

- **Query Parser**, ο οποίος δέχεται από το χρήστη την πληροφορία για το ποιο ερώτημα θέλει να τρέξει, τις απαραίτητες παραμέτρους ανάλογα με το ερώτημα, καθώς και πόσες φορές θέλει να το εκτελέσει και τον αριθμό των εργατών που επιθυμεί να χρησιμοποιήσει. Επομένως, με αυτόν τον τρόπο, μπορούμε να εκτελέσουμε τον ίδιο τύπο ερωτήματος χρησιμοποιώντας διαφορετικές τιμές για τα υποκείμενα, κατηγορούμενα και τα αντικείμενα που εμπλέκονται στο ερώτημα.
- **Query Execution** Το κομμάτι αυτό περιλαμβάνει τον κώδικά που τρέχει αφού ο χρήστης έχει εισάγει τις προτιμήσεις του και η εργασία έχει υποβληθεί. Σε αυτό το σημείο, τρέχει ο αναγκαίος κώδικας για τη φόρτωση των δεδομένων, την εκτέλεση των υπολογισμών και το γράψιμο της εξόδου. Ο κώδικας αυτός τρέχει σε όλους τους κόμβους-εργάτες.

3.4.2 Προεπεξεργασία

Όπως αναφέραμε παραπάνω, η είσοδος στο σύστημά μας είναι τα RDF σύνολα δεδομένων. Δυστυχώς, το Giraph δεν μπορεί να τα μεταφράσει αυτόματα σε γράφο, καθώς η πληροφορία για μία κορυφή ή ακμή μπορεί να βρίσκεται σε πολλές διαφορετικές γραμμές. Για το λόγο αυτό, κάναμε μια προεπεξεργασία των δεδομένων. Συγκεκριμένα, υλοποιήσαμε μια εργασία MapReduce, η οποία εκτελείται στα RDF σύνολα δεδομένων και τα ομαδοποιεί ανά αντικείμενο. Η ομαδοποίηση θα μπορούσε να είχε γίνει και ανά υποκείμενο ή κατηγορούμενο, όμως ο τρόπος που επιλέξαμε είναι πιο αποδοτικός για τα συγκεκριμένα ερωτήματα όπου το ζητούμενο είναι πάντα το υποκείμενο. Το αποτέλεσμα της προεπεξεργασίας είναι ένα σύνολο από λίστες γειτνίασης, όπου το πρώτο στοιχείο είναι το αντικείμενο, ακολουθούμενο από την αρχική του τιμή, μηδέν στην περίπτωσή μας, και τα ζεύγη υποκειμένου-κατηγορούμενου. Η αρχική τιμή χρειάζεται ώστε να να μην οδηγηθούμε σε ακαθόριστο αποτέλεσμα κατά τη διάρκεια εκτέλεσης του αλγορίθμου. Θα μπορούσε να είναι διαφορετική από μηδέν, αρκεί να μην είναι μονάδα, ή κάποια άλλη από τις τιμές που χρησιμοποιούνται μέσω των μηνυμάτων κατά την διάρκεια των υπολογισμών.

Η διαδικασία αυτή απεικονίζεται στο σχήμα 3.11.



Σχήμα 3.11: Προεπεξεργασία Δεδομένων

3.4.3 Υλοποίηση των Ερωτημάτων

Για την υλοποίηση των ερωτημάτων στο Giraph τρία πράγματα απαιτούνται από την πλευρά μας:

1. Κατάλληλη Μορφή Δεδομένων Εισόδου(Input Format)
2. Κατάλληλη Μορφή Δεδομένων Εξόδου (Output Format)
3. Αλγόριθμος Κορυφής - Συνάρτηση *compute*

Input Format Οι γράφοι μπορούν να αποθηκευτούν στο δίσκο με πολλές διαφορετικές μορφές. Το Giraph χρειάζεται να έχει ένα τρόπο να διαβάσει αυτά τα δεδομένα από το αποθηκευτικό σύστημα και να τα μεταφράσει σε γράφο. Ουσιαστικά, χρειάζεται ένα τρόπο να καθορίσει το πώς θα διαβάσει τα δεδομένα και θα τα μετατρέψει σε *Κορυφές* και *Ακμές*. Αυτό μπορεί να γίνει μέσω του Input Format.

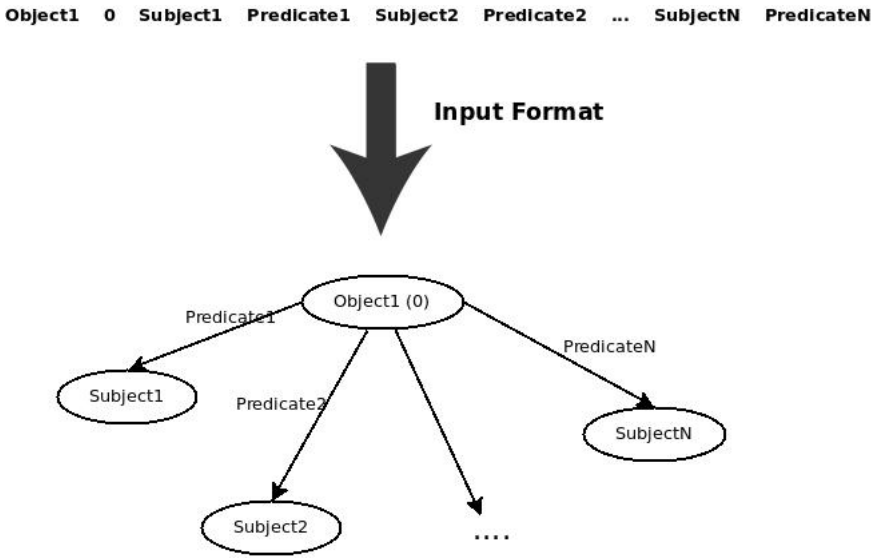
Το Giraph μας παρέχει ένα API γι'αυτόν τον σκοπό, αλλά όπως έχουμε ήδη αναφέρει δεν μπορεί να χρησιμοποιηθεί απευθείας για να διαβάσει τα RDF σύνολα δεδομένων σε N-Triple ή κάποια άλλη μορφή σειριοποίησης. Γι'αυτό και έχουμε το στάδιο της προεπεξεργασίας όπου τα RDF datasets έχουν μεταφραστεί σε λίστες γειτνίασης ομαδοποιημένες ανά αντικείμενο. Επομένως, μπορούμε πλέον να υλοποιήσουμε κατάλληλο Input Format, το οποίο θα μετατρέψει αυτήν την πληροφορία σε γράφο.

Συγκεκριμένα, μετά την προεπεξεργασία τα δεδομένα μας αποτελούνται από πολλές γραμμές της μορφής:

Object1 0 Subject1 Predicate1 Subject2 Predicate2 ... SubjectN PredicateN

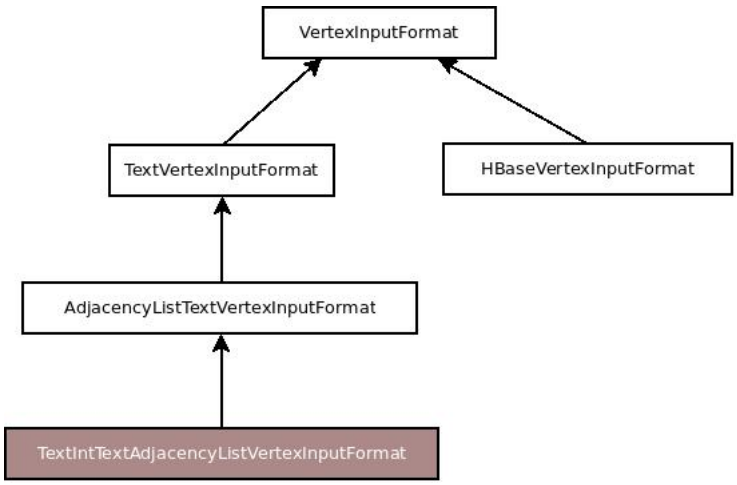
Εδώ, το Object1 υποδηλώνει μια κορυφή με Αναγνωριστικό: Object1 και αρχική τιμή μηδέν. Καθένα από τα ζεύγη υποκειμένου-κατηγορούμενου που ακολουθούν είναι οι κορυφές-γείτονες της αρχικής κορυφής, έχουντας ως αναγνωριστικά: Subject1, Subject2, ..., SubnectN αντίστοιχα. Παρόμοια, οι εξερχόμενες ακμές έχουν τιμές: Predicate1, Predicate2, ..., PredicateN.

Η αντίστοιχη γραφική αναπαράσταση απεικονίζεται στο σχήμα 3.12.



Σχήμα 3.12: Γραφική Αναπαράσταση

Το Giraph API για γραφικές αναπαραστάσεις βασισμένες στις κορυφές ονομάζεται *Vertex Input Format*. Το σχήμα 3.13 απεικονίζει ένα μικρό δείγμα από διάφορες κλάσεις input formats στον κώδικα του Giraph.



Σχήμα 3.13: Μορφή Δεδομένων Εισόδου

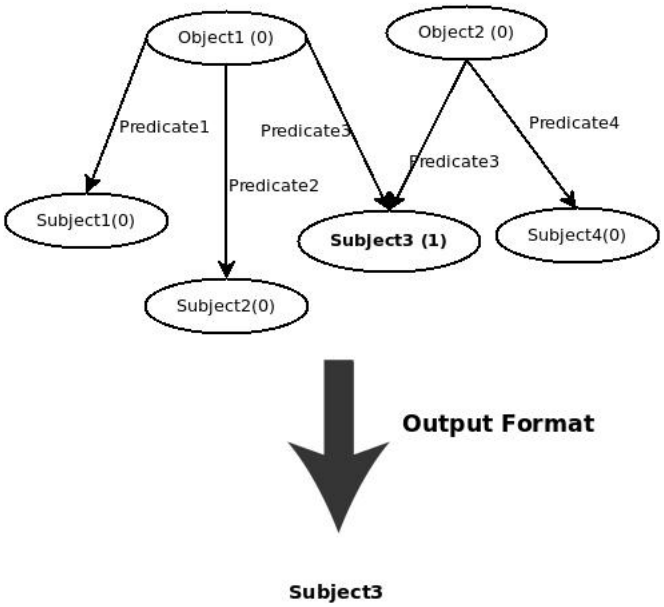
Τα λευκά κουτιά αναπαριστούν υπάρχουσες αφηρημένες κλάσεις, ενώ το χρωματισμένο: *TextIntTextAdjacencyListVertexInputFormat* αποτελεί τη δική μας υλοποίηση, όπου επεκτείνουμε την αφηρημένη κλάση *AdjacencyListTextVertexInputFormat*.

Output Format

Το Output Format κάνει την ακριβώς αντίθετη διαδικασία, κατά την οποία ο γράφος μεταφράζεται σε κάποια μορφή με την οποία μπορεί να αποθηκευτεί στο δίσκο.

Συγκεκριμένα, το αποτέλεσμα των ερωτημάτων βρίσκεται στις κορυφές και στις ακμές του γράφου. Στην περίπτωσή μας όλες οι κορυφές με τη τιμή '1', ικανοποιούν τα κριτήρια του ερωτήματος, και επομένως ως έξοδο θα θέλαμε να έχουμε μόνο τα αναγνωριστικά των κορυφών (VertexIDs) που έχουν αυτήν την τιμή αυτή.

Αυτή η μετατροπή του γράφου στην τελική μας έξοδο μπορεί να φανεί στο σχήμα 3.14



Σχήμα 3.14: Μορφή Δεδομένων Εξόδου

Παρόμοια με το Input Format, γράψαμε το δικό μας Output Format με το να επεκτείνουμε την αφηρημένη κλάση *TextVertexOutputFormat*.

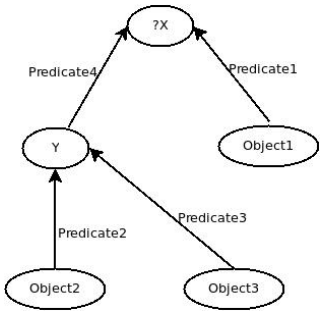
Vertex Algorithm - Compute Method Σε αυτό το σημείο ένα ή περισσότερα partitions του γράφου έχουν ανατεθεί σε κάθε κόμβο-εργάτη, και τώρα ο ίδιος είναι υπεύθυνος να καλέσει την compute συνάρτηση για όλες τις κορυφές που βρίσκονται σε αυτόν τον κόμβο-εργάτη. Αυτή η συνάρτηση υπολογισμού εκτελεί όλους τους υπολογισμούς που απαιτούνται για να βρεθεί η απάντηση και εκτελείται από κάθε κόμβο-εργάτη. Για να υλοποιήσουμε την compute μέθοδο, η οποία θα εκτελεστεί από την κάθε κορυφή, ακολουθήσαμε την ακόλουθη λογική.

Θεωρούμε τα SPARQL ερωτήματα ως δέντρα, με ρίζα το υποκείμενο ή το αντικείμενο το οποίο ψάχνουμε, μερικές ενδιάμεσες κορυφές/ακμές και φύλλα. Η επίλυση κάθε ερωτήματος εκτελείται από μία ακολουθία supersteps. Ως γενικό κανόνα, για να επιλύσουμε το ερώτημα στο Giraph, ξεκινάμε με τα φύλλα. Στο πρώτο superstep, όλες οι κορυφές ελέγχουν αν είναι τα επιθυμητά 'φύλλα' στο ερώτημα και αν οι εξερχόμενες ακμές τους ταιριάζουν με αυτές του ερωτήματος. Αν ισχύει αυτό, στέλνουν κατάλληλα μηνύματα στις κορυφές-γείτονες και σταματάνε. Από το δεύτερο superstep και μέχρι το πρόβλημα να λυθεί, οι κορυφές που έχουν λάβει μηνύματα από τις κορυφές-φύλλα, γίνονται ενεργές, ελέγχουν αν ικανοποιούν κάποια συγκεκριμένα χαρακτηριστικά και αναλόγως τα αποτελέσματα μπορεί να στείλουν κατάλληλα μηνύματα στους γείτονές τους. Η ίδια διαδικασία ακολουθείται μέχρι να επιλυθεί το ερώτημα. Στο τελικό superstep, οι κορυφές που ικανοποιούν τα κριτήρια του ερωτήματος έχουν την τιμή '1'. Δεν ανταλάσσεται κανένα άλλο μήνυμα και όλες οι κορυφές έχουν γίνει

ανενεργές. Έτσι, όταν ο υπολογισμός έχει ολοκληρωθεί, η έξοδος γράφεται και το πρόγραμμά μας τερματίζεται.

Για να κάνουμε κατανοητή αυτήν την προσέγγιση στον αναγνώστη, θα δούμε την παραπάνω διαδικασία βήμα βήμα. Ας πάρουμε για παράδειγμα το Ερώτημα 6, που απεικονίζεται στο σχήμα 3.7.

Αρχικά, με τη βοήθεια του Input Format, φορτώνουμε όλα τα δεδομένα ως ένα γράφο. Τώρα, με τη Compute μέθοδο, ψάχνουμε ουσιαστικά μέσα στο γράφο για έναν υπογράφο που έχει τη μορφή που βλέπουμε στο σχήμα 3.7, με τη διαφορά ότι οι ακμές έχουν ακριβώς την αντίθετη κατεύθυνση, όπως φαίνεται στο σχήμα 3.15.



Σχήμα 3.15: Ερώτημα 6 στο Giraph

Επομένως, ας πούμε, ότι το μέρος του γράφου που είναι φορτωμένο στο Giraph είναι αυτό που φαίνεται στο σχήμα 3.16.

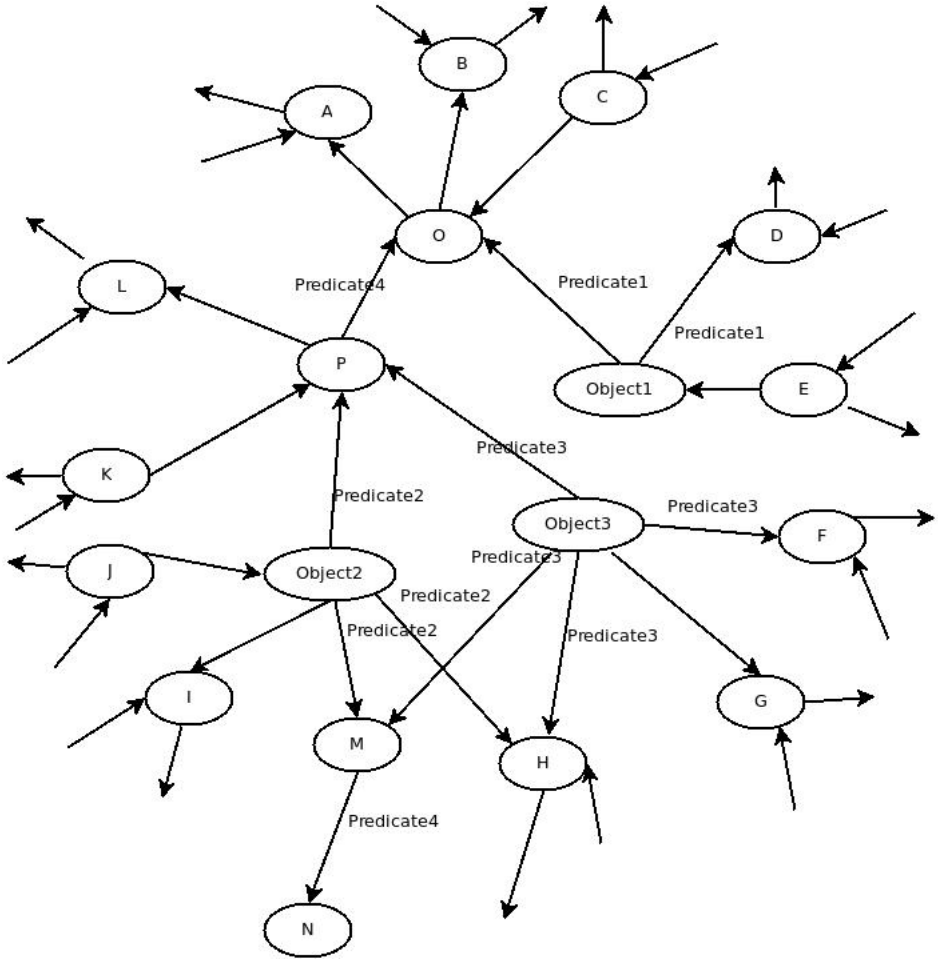
Η εκτέλεση του ερωτήματος σε αυτόν τον τύπο ερωτήματος ολοκληρώνεται σε τρία supersteps.

Superstep 0 Στο αρχικό superstep, η κάθε κορυφή ελέγει το δικό της VertexID. Αν δεν είναι ένα από τα Object1, Object2, Object3, η κορυφή δεν κάνει τίποτα και απλά σταματάει. Αν το VertexID της είναι το Object1, τότε ελέγχει αν το Predicate1 είναι η τιμή για μία από τις εξερχόμενες ακμές της. Στην περίπτωση που αυτό ισχύει στέλνει την τιμή ‘4’ στην κορυφή με την οποία είναι συνδεμένη με την ακμή με τιμή‘Predicate1’, και σταματάει. Παρόμοια, η κορυφή με VertexID Object2 και εξερχόμενη ακμή Predicate2, στέλνει την τιμή ‘1’ στην αντίστοιχη κορυφή-γείτονα (πιθανό Y), και η κορυφή με VertexID Object3 και εξερχόμενη ακμή Predicate3 στέλνει την τιμή ‘2’ στην αντίστοιχη κορυφή-γείτονα (και πάλι πιθανό Y). Και οι δύο αυτές κορυφές σταματάνε αμέσως μετά που θα στείλουν τα μηνύματα. Τα παραπάνω μπορούμε να τα δούμε στο σχήμα 3.17.

Οι τιμές που στέλνονται μέσω των μηνυμάτων από τη μία κορυφή στην άλλη μπορεί να είναι διαφορετικές από αυτές που παρουσιάζονται στο συγκεκριμένο παράδειγμα. Στη συγκεκριμένη υλοποίηση χρησιμοποιούμε τα μηνύματα μόνο για να δηλώσουμε ότι μία κορυφή έλαβε μήνυμα από μια άλλη κορυφή που περιλαμβάνεται στο ερώτημα ή/και για να μετατρέψουμε μια ανενεργή κορυφή, ενεργή ξανά. Για παράδειγμα, εδώ το ‘4’ σημαίνει ότι ‘Έλαβα μήνυμα από την κορυφή με VertexID: Object1, με την οποία είμαι συνδεμένη μέσω μιας ακμής με τιμή ‘Predicate1’.

Superstep 1 Στο superstep 1, είναι ενεργές μόνο οι κορυφές που έλαβαν μήνυμα από το προηγούμενο superstep. Έτσι στο παράδειγμά μας, τώρα ενεργές είναι οι: O, D, P, M, H και F. Οι O και D έχουν λάβει μήνυμα με τιμή 4 από την κορυφή με VertexID: Object1. Οι P, M και H έχουν λάβει μήνυμα με τιμή 1 από την κορυφή με VertexID: Object2. Οι P, M, H και F έχουν λάβει μήνυμα με τιμή 2 από την κορυφή με VertexID: Object3.

Σε αυτό το superstep, κάθε μία από τις ενεργές κορυφές ελέγχει το μήνυμα ή τα μηνύματα που έλαβε και ενεργεί αναλόγως. Οι O και D έχουν είναι κορυφές που ικανοποιούν μία από τις απαραίτητες συνθήκες ώστε να αποτελέσουν απάντηση στο ερώτημά μας. Για να δηλωθεί αυτή η πληροφορία, αλλά και να να παραμείνει αμετάβλητη στα supersteps που θα ακολουθήσουν, αλλάζουν την τιμή



Σχήμα 3.16: Τμήμα του γράφου που έχει φορτωθεί στο Giraph

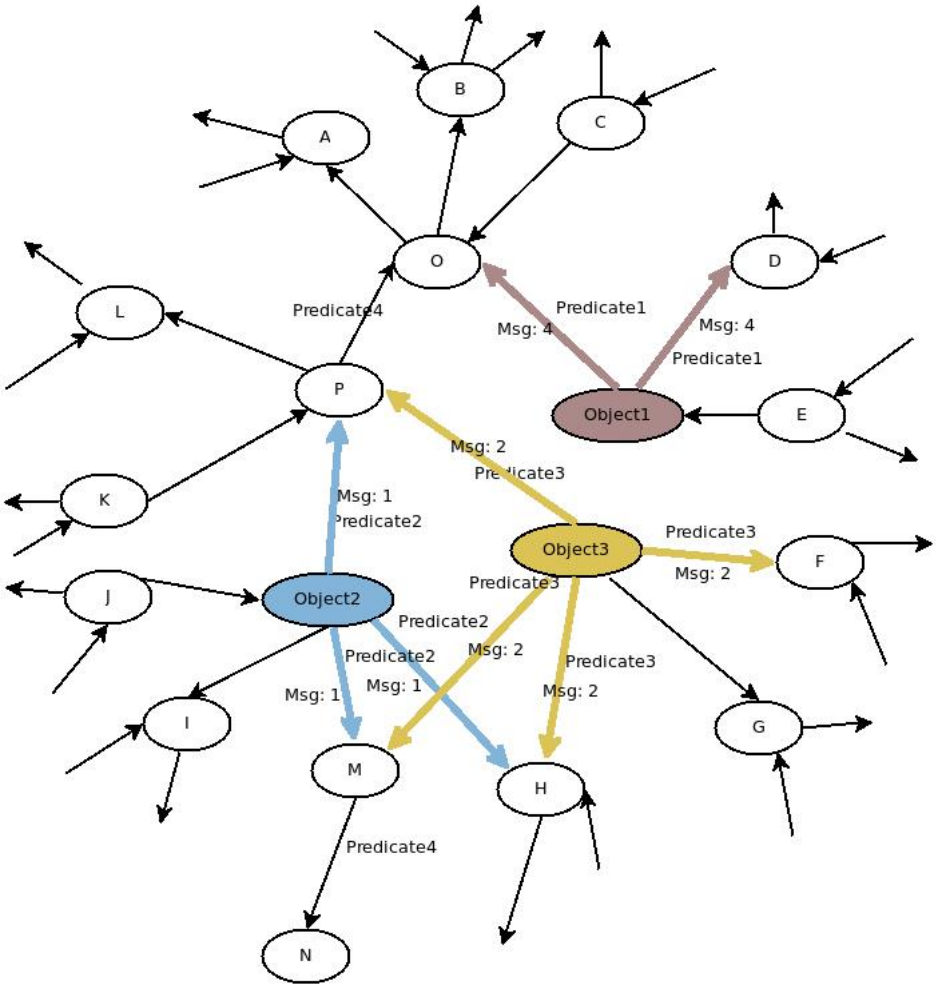
τους από 0 σε 5 και σταματάνε. Παρόμοια οι κορυφές που έλαβαν μήνυμα και από το Object2 και το Object3, (στην περίπτωσή μας οι P, M και H) ελέγχουν αν έχουν εξερχόμενη ακμή με τιμή: Predicate4 και αν αυτό ισχύει στέλνουν σε κορυφή μήνυμα με την τιμή '3', και σταματάνε. Εδώ, η κορυφή F, δεν κάνει τίποτα, γιατί έχει λάβει μήνυμα μόνο από την Object3. Αντίστοιχα η H, παρόλο που έλαβε και τα δύο απαιτούμενα μηνύματα από τις κορυφές Object2 και Object3, δεν έχει εξερχόμενη ακμή με την κατάλληλη τιμή και έστι σταματάει χωρίς να κάνει καμία άλλη ενέργεια.

Τα παραπάνω φαίνονται στο σχήμα 3.18

Superstep 2 Στο τελικό superstep ενεργές είναι μόνο οι κορυφές που έχουν λάβει μήνυμα από το προηγούμενο superstep. Στο παράδειγμά μας αυτές είναι οι O και N. Ελέγχουν αν έχουν λάβει μήνυμα με αυτήν που έχουμε δηλώσει ως επιθυμητή τιμή (3) και αν αυτό ισχύει, ελέγχουν τη δική τους τιμή. Αν αυτή είναι '5', που σημαίνει ότι έχουν λάβει μήνυμα από το Object1, τότε ικανοποιούν όλες τις απαραίτητες προϋποθέσεις, πράγμα που σημαίνει ότι αποτελούν απάντηση ή μέρος της απάντησης του ερωτήματός μας και θέτουν τη δική τους τιμή σε 1. Στο παράδειγμά μας, η κορυφή O αποτελεί απάντηση στο ερώτημα, ενώ η N, παρόλο που έχει λάβει μήνυμα με την επιθυμητή τιμή, δεν είναι συνδεμένη με Object1 και κατ'επέκταση δεν αποτελεί απάντηση του ερωτήματος.

Αντίστοιχα, η κορυφή D που έχει μαρκαριστεί ως πιθανή απάντηση στο superstep 0, τελικά δεν είναι, καθώς δεν είναι συνδεμένη με κάποια κορυφή που να έχει τα χαρακτηριστικά της Y, άρα δεν αποτελεί απάντηση και γίνεται ανενεργή.

Οι αλλαγές αυτές απεικονίζονται στο σχήμα 3.19



Σχήμα 3.17: Superstep 0

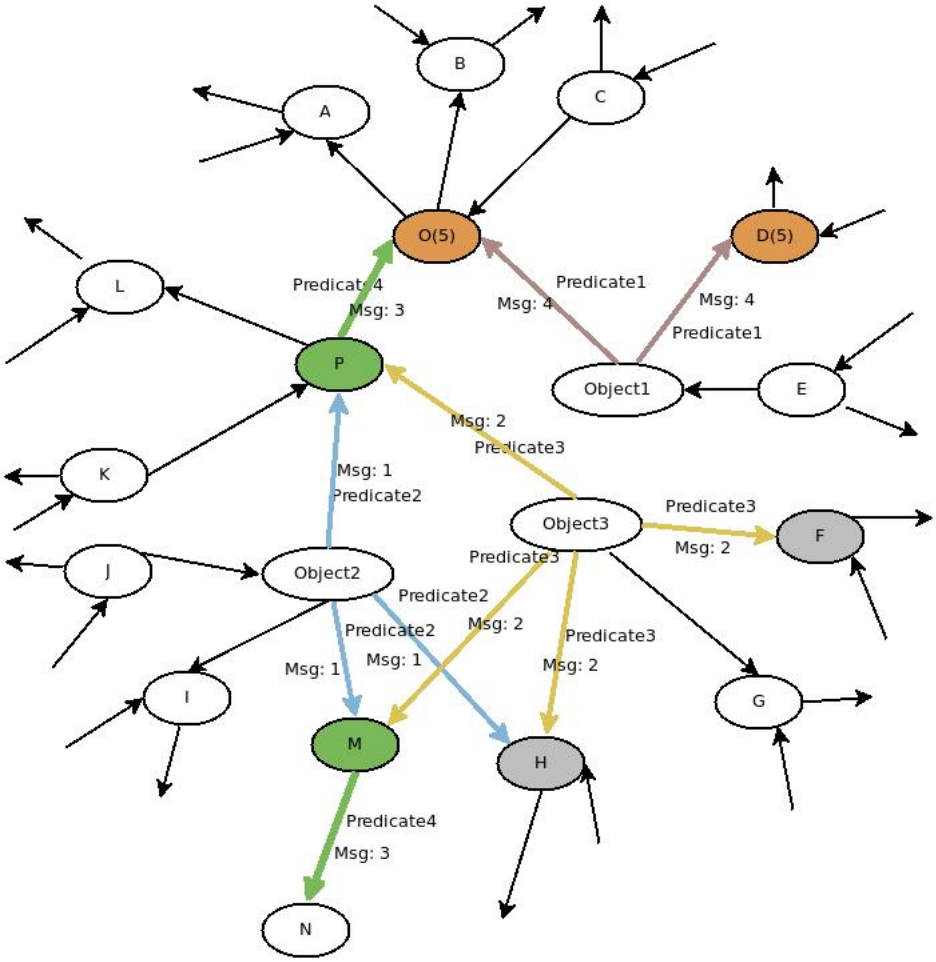
Μετά την ολοκλήρηση του 3ου superstep (superstep 2), όλες οι κορυφές είναι ανενεργές. Επομένως, η εργασία ολοκληρώνεται με το γράψιμο των αποτελεσμάτων στο HDFS. Στο συγκεκριμένο παράδειγμα μόνο η κορυφή O ικανοποιεί τα κριτήρια του ερωτήματος, οπότε είναι και το μοναδικό VertexID που τυπώνεται στο αρχείο εξόδου. Προφανώς, το παραπάνω είναι απλά ένα παράδειγμα. Στην πράξη πολλές κορυφές μπορεί να ικανοποιούν τα κριτήρια του ερωτήματος, και συνεπώς να έχουμε πολλές κορυφές ως έξοδο.

3.5 Χρόνοι στο Giraph

Για να μετρήσουμε το χρόνο που απαιτείται για την ολοκλήρωση κάποιου ερωτήματος στο Giraph, χρησιμοποιήσαμε τους χρόνους του Giraph (Giraph timers), που εμφανίζονται στο stdout, αμέσως μετά την ολοκλήρωση του ερωτήματος.

Ακολουθεί μια βασική περιγραφή τους:

- **Initialize** Μετράει το χρόνο που απαιτείται από την εργασία καθώς περίμενε για υπολογιστικούς πόρους (resources). Όταν οι πόροι μοιράζονται, είναι πολύ πιθανό η εργασία που υποβάλεις να μην μπορέσει να δεσμεύσει όλους τους κόμβους που χρειάζονται για να ξεκινήσει.
- **Setup** Είναι ο χρόνος που απαιτείται πριν να αρχίσει να διαβάζει την είσοδο, από τη στιγμή που όλοι οι κόμβοι έχουν διανεμηθεί και μετά.



Σχήμα 3.18: Superstep 1

- **Input Superstep** Είναι ο χρόνος που απαιτείται για την ανάγνωση της εισόδου (κορυφές και ακμές), τη φόρτωση δηλαδή του γράφου στη μνήμη των μεμονωμένων κόμβων-εργατών, την ανάθεση κορυφών σε τμήματα και στη συνέχεια τα τμήματα αυτά στους εργάτες, τη μεταφορά όλων των απαιτούμενων τμημάτων στο κάθε εργάτη καθώς και τήρηση κάποιων εσωτερικών δομών που θα χρησιμοποιούν κατά τον υπολογισμό.

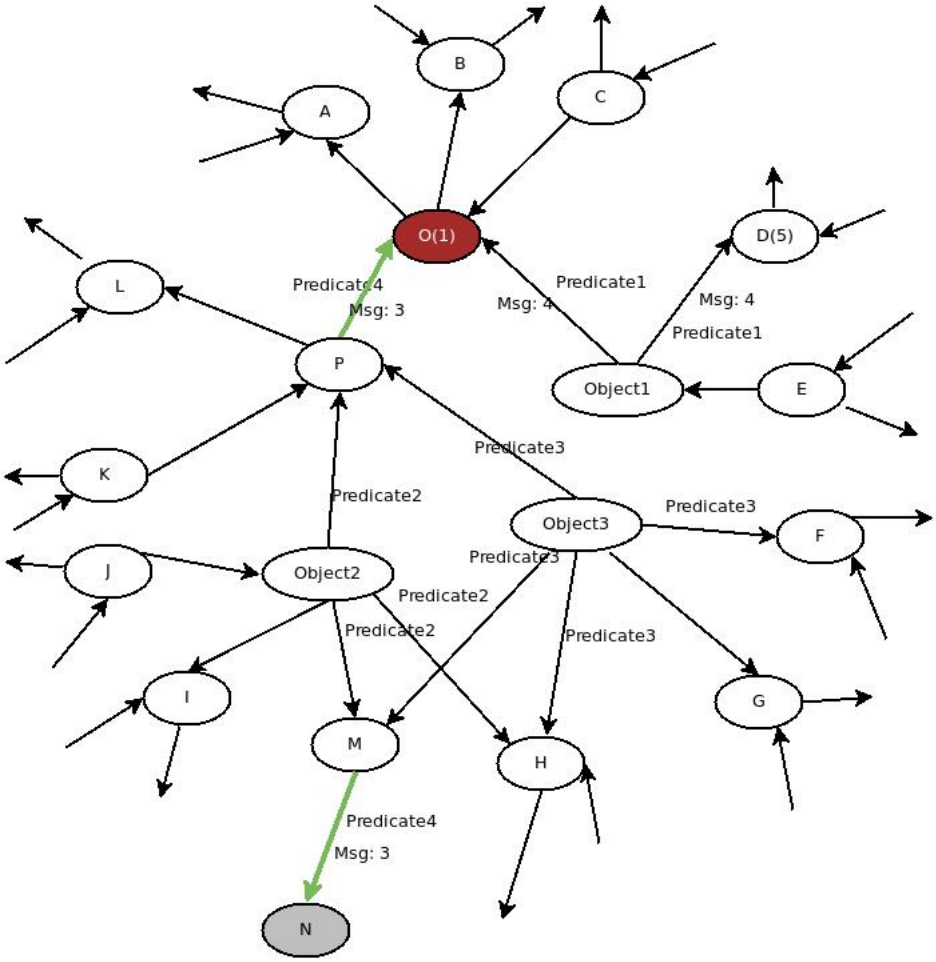
- **Superstep i** Εδώ τρέχει η προκαθορισμένη από το χρήστη συνάρτηση υπολογισμού.

- **Shutdown** Είναι ο χρόνος που απατείται για να σταματήσουμε και να επιβεβαιώσουμε ότι όλα έχουν γίνει μετά το γράψιμο της εξόδου, δηλαδή οι πόροι έχουν αποδεσμευτεί(όλες οι συνδέσεις δικτύου έχουν κλείσει, όλα τα νήματα έχουν ενωθεί,κλπ) και ο χρήστης έχει ενημερωθεί.

- **Total** Είναι ο πραγματικός χρόνος που χρειάζεται για να τρέξει η εφαρμογή από τη στιγμή που έχουν δεσμευτεί όλοι οι πόροι. Άρα δεν περιλαμβάνει τους χρόνους Initialize ή Shutdown.

3.6 Εκτέλεση στο Openvirtuoso

Για να αξιολογήσουμε την απόδοση του συστήματός μας σε σύγκριση με μια ευρείως διαδεδομένη κεντρική προσέγγιση για επερώτηση τέτοιων συνόλων δεδομένων, επιλέξαμε το Openvirtuoso [28]. Παρακάτω ακολουθούν τα ακριβή ερωτήματα που τρέξαμε, ορισμένα σε SPARQL.



Σχήμα 3.19: Superstep 2

```
Query1a
sparql select ?x where { ?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> . ?x
  <http://dbpedia.org/property/birthPlace>
  <http://dbpedia.org/resource/Queens> .};

Query1b
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> . ?x
  <http://dbpedia.org/property/birth>
  '1979'^^<http://www.w3.org/2001/XMLSchema#date> .};

Query1c
sparql select ?x where {?x <http://xmlns.com/foaf/0.1/name>
  'Sokrates'@de . ?x <http://purl.org/dc/elements/1.1/description>
  'griechischer Philosoph'@de .};

Query1d
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/City108524735> . ?x
  <http://dbpedia.org/property/populationAsOf>
```

```
  '2006'^^<http://www.w3.org/2001/XMLSchema#Integer> .};

Query1e
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/Emirate108557396> . ?x
  <http://dbpedia.org/property/mapCaption>
  <http://dbpedia.org/resource/Saudi_Arabia> .};

Query2a
sparql select ?x where {?x <http://xmlns.com/foaf/0.1/name> 'Verne,
  Jules'@de . ?x <http://dbpedia.org/property/deathPlace>
  <http://dbpedia.org/resource/Amiens> . ?x
  <http://xmlns.com/foaf/0.1/givenname> ?y1 . ?x
  <http://xmlns.com/foaf/0.1/surname> ?y2 . ?x
  <http://dbpedia.org/property/birthPlace> ?y3 .};

Query2b
sparql select ?x where {?x <http://dbpedia.org/property/influences>
  <http://dbpedia.org/resource/Plato> . ?x
  <http://dbpedia.org/property/influenced>
  <http://dbpedia.org/resource/Alexander_the_Great> . ?x
  <http://dbpedia.org/property/shortDescription> ?y1 . ?x
  <http://dbpedia.org/property/notableIdeas> ?y2 . ?x
  <http://dbpedia.org/property/mainInterests> ?y3 .};

Query2c
sparql select ?x where {?x <http://dbpedia.org/property/influences>
  <http://dbpedia.org/resource/Plato> . ?x
  <http://dbpedia.org/property/influenced> ?y1 . ?x
  <http://dbpedia.org/property/shortDescription> ?y2 . ?x
  <http://dbpedia.org/property/notableIdeas> ?y3 . ?x
  <http://dbpedia.org/property/mainInterests>
  <http://dbpedia.org/resource/Science> .};

Query3a
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/Book> .};

Query3b
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> .};

Query3c
sparql select ?x where {?x <http://xmlns.com/foaf/0.1/homepage>
  <http://www.educause.edu/> .};

Query4
sparql select ?x where {<http://dbpedia.org/resource/\%21\%21\%21>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/Group100031264> . ?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/Group100031264> . ?x
  <http://dbpedia.org/property/genre>
  <http://dbpedia.org/resource/Alternative_rock>
  . <http://dbpedia.org/class/yago/Group100031264>
```

```
  <http://www.w3.org/2000/01/rdf-schema#label>  'Group' .};

Query5
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> . ?x
  <http://xmlns.com/foaf/0.1/name> ?z . ?x
  <http://dbpedia.org/property/birthPlace> ?y . ?y
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/City108524735> . ?y
  <http://dbpedia.org/property/populationRef>
  <http://www.statistik.baden-wuerttemberg.de> .};

Query6
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> . ?x
  <http://dbpedia.org/property/birthPlace> ?y . ?y
  <http://dbpedia.org/property/populationRef>
  <http://www.statistik.baden-wuerttemberg.de> . ?y
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/City108524735> .};

Query7
sparql select ?x where
{<http://dbpedia.org/resource/Ferdinand_Rudolph_Hassler>
  <http://dbpedia.org/property/birthPlace> ?x . ?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/Capital108518505> .};
```

Κεφάλαιο 4

Πειράματα

4.1 Στήσιμο περιβάλλοντος

4.1.1 Σύνολα Δεδομένων

Για τα πειράματά μας χρησιμοποιήσαμε δύο σύνολα δεδομένων διαφορετικού μεγέθους από την DBpedia:

1. Σύνολο Δεδομένων 1: http://downloads.dbpedia.org/2.0/
2. Σύνολο Δεδομένων 2: http://downloads.dbpedia.org/3.0/en/

Αυτά τα σύνολα δεδομένων περιέχουν άρθρα, περιλήψεις, εικόνες, συνδέσμους σε εξωτερικές ιστοσελίδες, πληροφορίες που έχουν εξαχθεί από τα infoboxes της Wikipedia, συνδέσμους μεταξύ γεωγραφικών τόπων στην DBpedia και δεδομένα γι'αυτούς στη βάση δεδομένων Geonames και άλλα. Περισσότερη πληροφορία για το περιεχόμενο αυτών των συνόλων δεδομένων μπορεί να βρεθεί στο wiki της κοινότητας της DBpedia. Συγκεκριμένα, στους παρακάτω συνδέσμους για το Σύνολο Δεδομένων 1 και 2 αντίστοιχα:

- http://wiki.dbpedia.org/data-set-20
- http://wiki.dbpedia.org/data-set-30

Και τα δύο σύνολα δεδομένων περιέχουν δεδομένα σε N-Triple μορφή. Το συνολικό τους μέγεθος, πριν και μετά την προεπεξεργασία τους απεικονίζεται στον πίνακα 4.1:

Dataset Name	Αρχικό Μέγεθος	Μέγεθος μετά την προεπεξεργασία
Dataset 1	6.8GB	5.2GB
Dataset 2	18.0GB	13.0GB

Πίνακας 4.1: Μέγεθος των χρησιμοποιηθέντων συνόλων δεδομένων

4.1.2 Χαρακτηριστικά περιβάλλοντος

Όλα μας τα πειράματα έγιναν σε περιβάλλον cloud, χρησιμοποιώντας το Openstack ως πλατφόρμα λογισμικού. Δημιουργήσαμε 8 εικονικές μηχανές (Virtual Machines - VMs), που φιλοξενούνται στο Computer System Laboratory of NTUA. Οι εικονικές μηχανές έχουν εγκατεστημένο το λειτουργικό σύστημα Ubuntu 14.04.2 x64, 8GB RAM η καθεμία, 4VCPUs και 80.0GB αποθηκευτικό χώρο. Για τα πειράματά μας χρησιμοποιήσαμε:

- Hadoop έκδοση: 0.20.203.0
- Giraph έκδοση 1.2.0 and
- Openvirtuoso έκδοση 6.1.

Για να έχουμε ασφαλέστερα αποτελέσματα εκτελέσαμε το κάθε ερώτημα πέντε φορές και πήραμε το μέσο όρο ως τελικό αποτέλεσμα.

4.2 Αποτελέσματα Πειραμάτων

4.2.1 Επιλογή των παραμέτρων του Hadoop

Το Giraph αξιοποιεί την υπάρχουσα υποδομή του Hadoop και τα ερωτήματά μας υποβάλλονται σε αυτό ως Hadoop εργασίες, που έχουν μόνο Map μέρος. Μια εργασία Hadoop, μπορεί να έχει αρκετά διαφορετική απόδοση ανάλογα με τις επιλεχθείσες τιμές των διαφόρων παραμέτρων του Hadoop. Έτσι, θεωρήσαμε λογικό να υποθέσουμε ότι η εύρεση των σωστών τιμών για κάποιες παραμέτρους μπορεί να παίξει σημαντικό ρόλο στην απόδοση μιας εργασίας Giraph. Εδώ εξετάσαμε την επίδραση των ακόλουθων παραμέτρων:

1. dfs.replication
2. dfs.block.size

Η πρώτη καθορίζει το πόσες φορές αντιγράφεται η είσοδος από το τοπικό σύστημα αρχείων στο HDFS, κυρίως για να εξασφαλίσουμε υψηλή διαθεσιμότητα των δεδομένων, ενώ η δεύτερη καθορίζει το μέγεθος των blocks στο HDFS.

Ερευνήσαμε την επίδραση των παραμέτρων αυτών κατά τη διάρκεια των ακόλουθων σταδίων:

1. Αντιγραφή της εισόδου από το τοπικό σύστημα αρχείων στο HDFS
2. Φόρτωση του γράφου στο Giraph
3. Χρόνος υπολογισμού του ερωτήματος

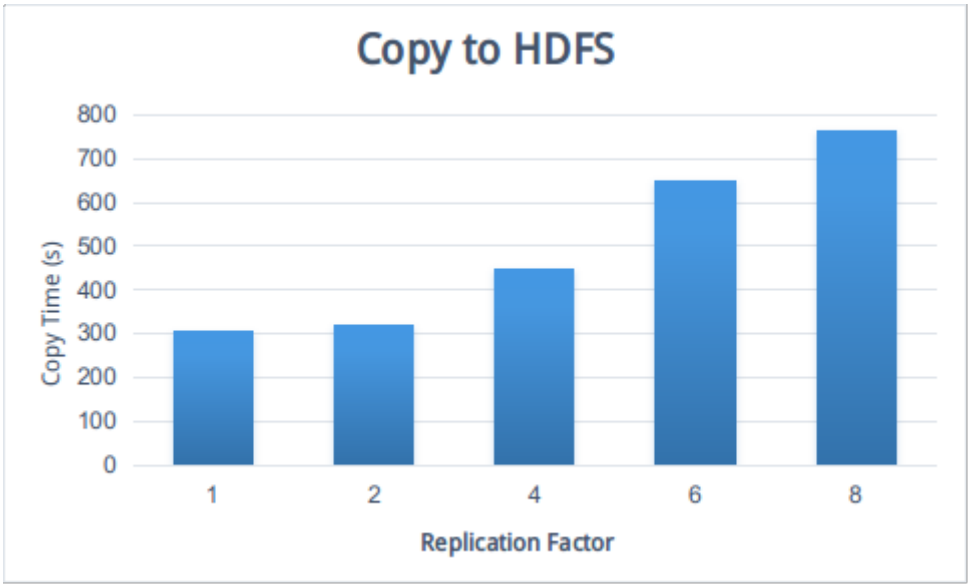
Αντιγραφή στο HDFS

Στο σχήμα 4.1 απεικονίζεται το πώς μεταβάλλεται ο χρόνος αντιγραφής όταν η παράμετρος: dfs.replication αυξάνεται.

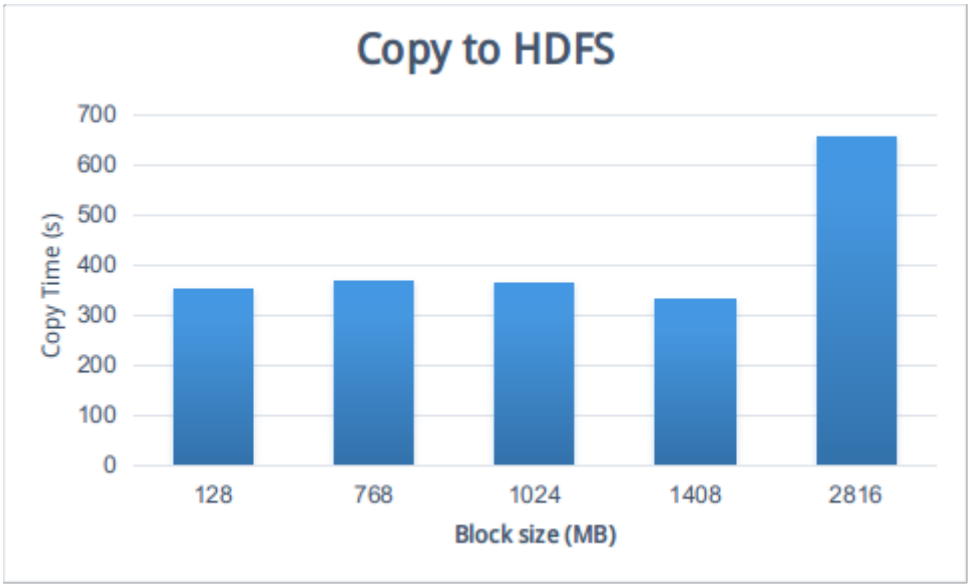
Όπως είναι αναμενόμενο, όσο αυξάνει ο αριθμός των αντιγράφων, απαιτείται περισσότερος χρόνος για την αντιγραφή τους από το τοπικό σύστημα αρχείων στο HDFS. Παρόλα αυτά, παρατηρήσαμε ότι με τιμή ίση με δύο, ο χρόνος αντιγραφής είναι συγκρίσιμος με αυτόν, όταν η τιμή είναι ένα, ενώ την ίδια στιγμή έχουμε υψηλή διαθεσιμότητα των δεδομένων, με αποτέλεσμα να είναι προτιμότερη. Μεγαλύτερες τιμές της παραμέτρου αυτής προσθέτουν αρκετά περισσότερο χρόνο στο στάδιο της αντιγραφής.

Στο σχήμα 4.2 απεικονίζεται το πώς μεταβάλλεται ο χρόνος αντιγραφής όταν η παράμετρος: dfs.block.size αυξάνεται.

Διαπιστώνουμε ότι για τιμές block από 128Mb σε 1408Mb, ο χρόνος αντιγραφής είναι συγκρίσιμος, με την τιμή 1408Mb(ή 1.375GB) να είναι η βέλτιση. Με μεγαλύτερες όμως τιμές διπλασσιάζεται ο χρόνος που απαιτείται για τη φάση της αντιγραφής.



Σχήμα 4.1: Αντιγραφή από το τοπικό σύστημα αρχείων στο HDFS για διαφορετικές τιμές της παραμέτρου dfs.replication



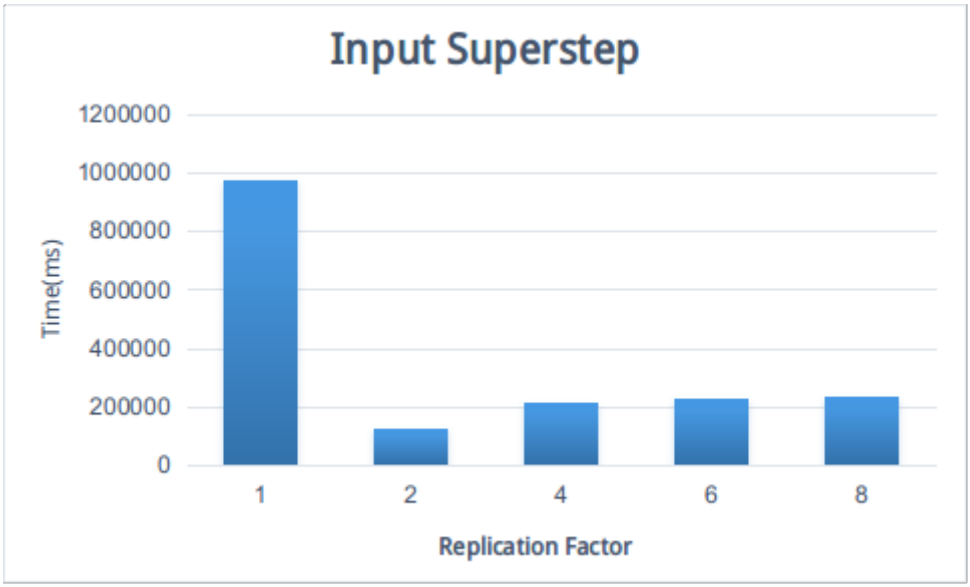
Σχήμα 4.2: Αντιγραφή από το τοπικό σύστημα αρχείων στο HDFS για διαφορετικές τιμές της παραμέτρου dfs.block.size

Φόρτωση γράφου στο Giraph

Στο σχήμα 4.3 απεικονίζεται το πώς μεταβάλλεται ο χρόνος φόρτωσης του γράφου, όταν η παράμετρος
dfs.replication αυξάνεται.

Παρατηρούμε ότι η φόρτωση του γράφου απαιτεί σημαντικά περισσότερο χρόνο όταν ο παράγοντας αντιγραφής είναι ένα, συγκριτικά με οποιαδήποτε άλλη τιμή. Αυτό είναι αναμενόμενο, γιατί τα δεδομένα βρίσκονται μόνο σε ένα κόμβο και όλοι οι εργάτες θέλουν να φορτώσουν μέρος του γράφου σύμφωνα με τα τμήματα που τους έχουν ανατεθεί.

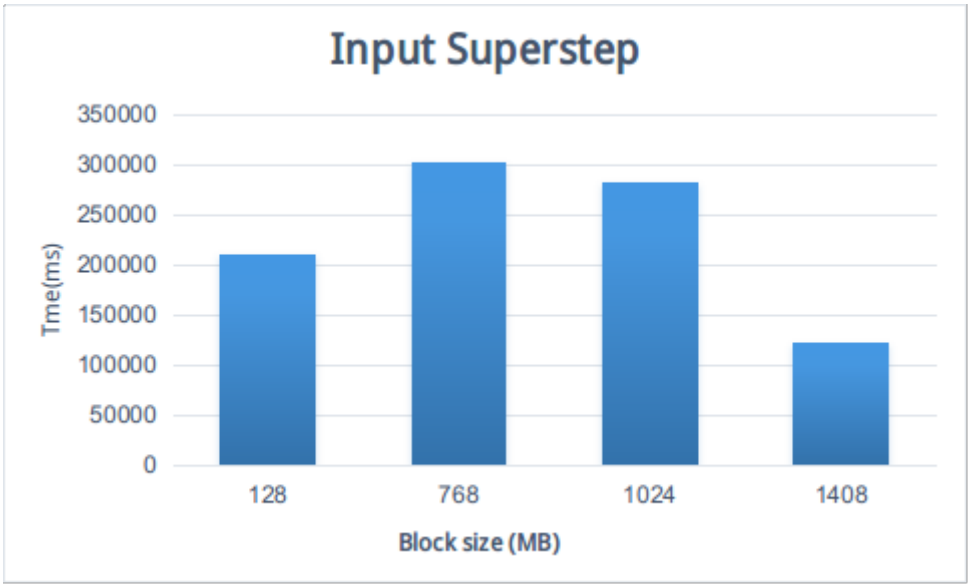
Όλες οι υπόλοιπες τιμές του παράγοντα αντιγραφής είναι πιο κοντά μεταξύ τους με την τιμή δύο να



Σχήμα 4.3: Input Superstep για διαφορετικές τιμές της παραμέτρου dfs.replication

συνεχίζει να είναι η καλύτερη επιλογή.

Στο σχήμα 4.4 απεικονίζεται το πώς μεταβάλλεται ο χρόνος για τη φόρτωση του γράφου, καθώς το μέγεθος του block αυξάνεται.



Σχήμα 4.4: Input Superstep για διαφορετικές τιμές της παραμέτρου dfs.block.size

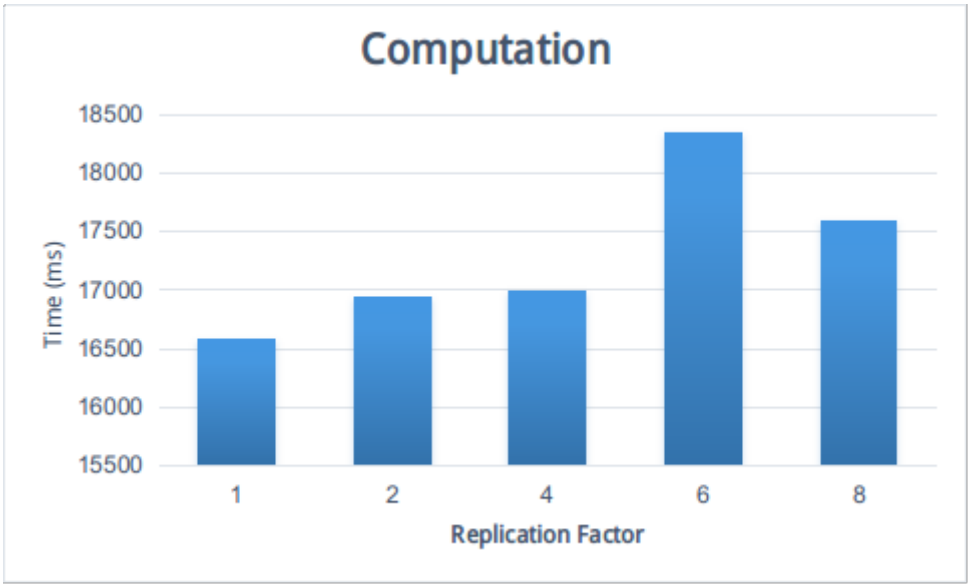
Με μέγεθος 2816MB όλες οι εκτελέσεις ήταν αποτυχημένες. Αυτό είναι αναμενόμενο, καθώς ο κάθε εργάτης πρέπει να φορτώσει τα κομμάτια της εισόδου που του ανατέθηκαν από τον αφέντη στη δική του τοπική μνήμη. Εκτός από τη μνήμη που πρέπει να δεσμευτεί για να αποθηκευτούν τα βασικά στοιχεία του γράφου: οι ακμές και οι κορυφές μαζί με τις τιμές τους, απαιτείται μνήμη και για τα εισερχόμενα/εξερχόμενα μηνύματα, κάνοντας έτσι τη συνολική μνήμη που απαιτείται να ξεπερνάει τα διαθέσιμα 8GB.

Σχετικά με τις υπόλοιπες τιμές, παρατηρούμε ότι δεν έχουμε σταθερή συμπεριφορά καθώς το μέγεθος του block αυξάνεται. Τα καλύτερα αποτελέσματα επιτυγχάνονται με τιμή: 1408MB. Φαίνεται να

είναι ιδανική τιμή, γιατί το μέγεθος είναι αρκετό μεγάλο, ώστε να έχουμε τα ελάχιστα μεταδεδομένα και την αντίστοιχή διαχείρισή τους, αλλά ταυτόχρονα και αρκετά μικρό, ώστε να χωράει στη μνήμη ενός μεμονωμένου κόμβου, με αποτέλεσμα να μη προκαλούνται αποτυχίες λόγω υπερφόρτωσης της μνήμης.

Χρόνος υπολογισμού ερωτήματος

Στο σχήμα 4.5 απεικονίζεται το πώς μεταβάλλεται ο χρόνος υπολογισμού του ερωτήματος, καθώς η παράμετρος dfs.replication αυξάνεται.



Σχήμα 4.5: Χρόνος Υπολογισμού για διαφορετικές τιμές της παραμέτρου dfs.replication.factor

Με μια πρώτη ματιά φαίνεται κάπως παράδοξο ότι ο υπολογισμός παίρνει το λιγότερο χρόνο, όταν ο παράγοντας αντιγραφής ισούται με ένα. Σε αυτήν την περίπτωση, ολόκληρη η είσοδος, με την αντιγραφή στο HDFS θα πάει σε ένα μόνο κόμβο. Έτσι, το μεγαλύτερο μέρος του γράφου εισόδου, υπάρχει σε ένα μόνο κόμβο και η επικοινωνία μεταξύ των εργατών είναι η ελάχιστη που απαιτείται.

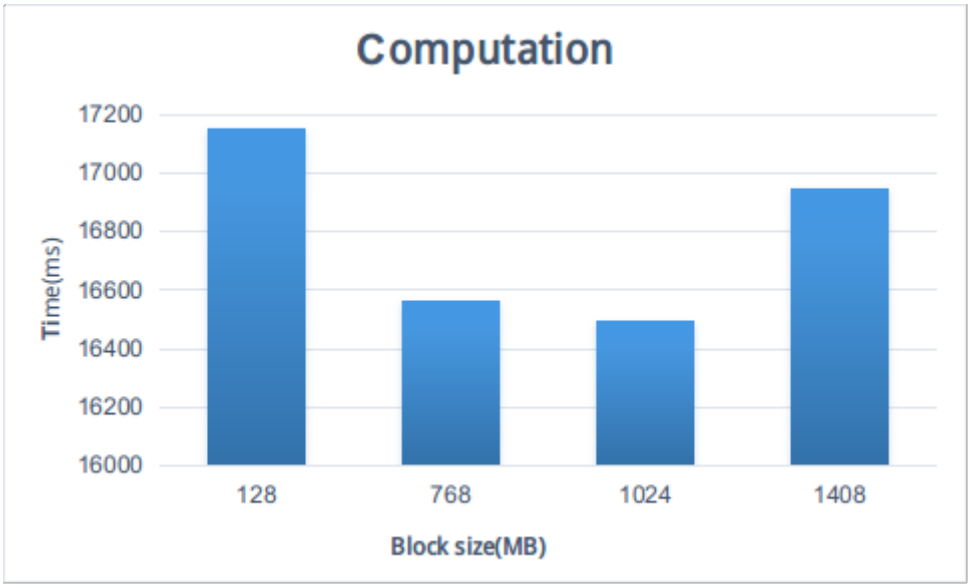
Στο σχήμα 4.6 απεικονίζεται το πώς μεταβάλλεται ο χρόνος υπολογισμού, όταν αυξάνεται το μέγεθος του block.

Παρατηρούμε ότι ο υπολογισμός εκτελείται σε μικρότερο χρονικό διάστημα όταν το μέγεθος του block αυξάνεται, πράγμα που ισχύει για μεγέθη block από 128MB έως 1024MB. Μεγαλύτερες τιμές από 1024MB οδηγούν σε μεγαλύτερο χρόνο υπολογισμού. Αυτό συμβαίνει, γιατί υπάρχουν πολύ λιγότερες επιλογές για την κατάτμηση του γράφου, οδηγώντας σε μη αποδοτική διαχείριση φορτίου (load-balancing), που σημαίνει ότι έχουμε εργάτες που απαιτούν αρκετά περισσότερο χρόνο να υπολογίσουν τα αποτελέσματά τους σε σχέση με άλλους.

Τα παραπάνω αποτελέσματα είναι για το το Σύνολο Δεδομένων 1. Παρόλα αυτά, εκτελέσαμε μερικά πειράματα χρησιμοποιώντας το μεγαλύτερο dataset, και δε φάνηκε να υπάρχει κάποια διαφοροποίηση στη συμπεριφορά.

4.2.2 Χρόνος - Διαφορετικός τύπος ερωτημάτων

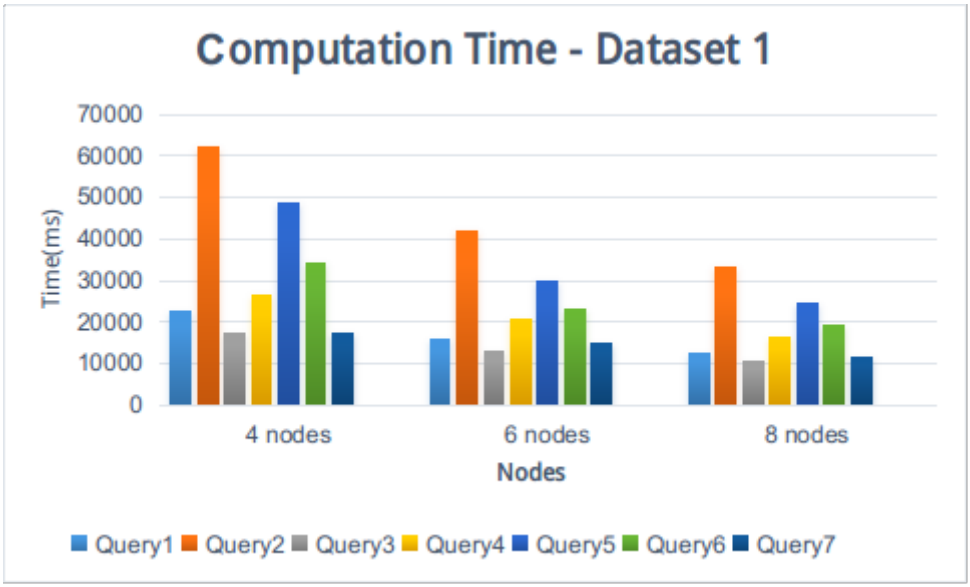
Σκοπός αυτού του τμήματος είναι να εντοπίσουμε τις διαφορές στο χρόνο υπολογισμού μεταξύ των ερωτημάτων διαφορετικού είδους, αν η ίδια σημπεριφορά παραμένει, καθώς ο αριθμός των κόμβων



Σχήμα 4.6: Χρόνος Υπολογισμού για διαφορετικές τιμές της παραμέτρου dfs.block.size

αυξάνεται και τους λόγους που εξηγούν αυτή τη συμπεριφορά.

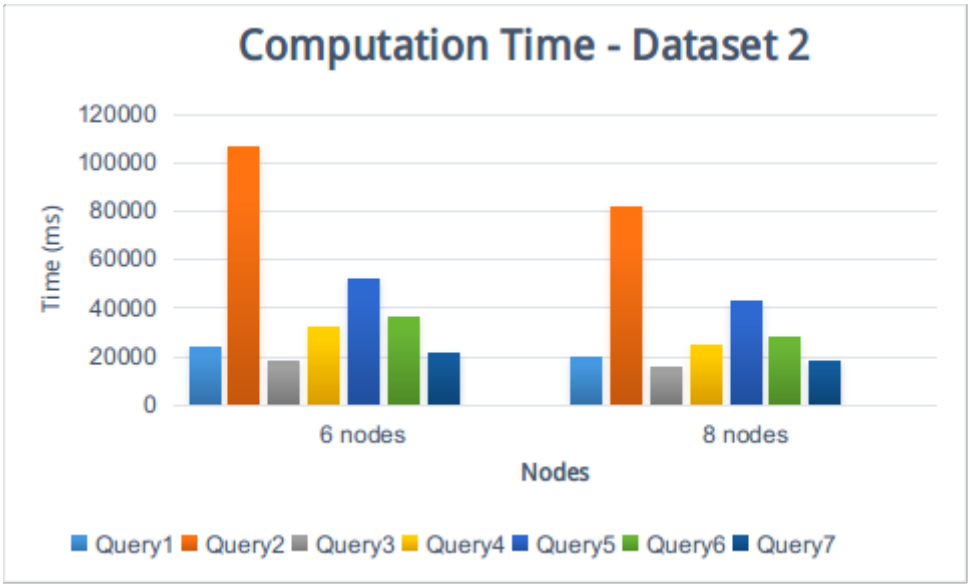
Τα αποτελέσματα αφορούν και τους εφτά διαφορετικούς τύπους ερωτημάτων και απεικονίζονται στα σχήματα 4.7 και 4.8.



Σχήμα 4.7: Χρόνος Υπολογισμού - Σύνολο Δεδομένων 1

Μπορούμε να δούμε ότι σε όλες τις περιπτώσεις η συμπεριφορά των ερωτημάτων από άποψη του χρόνου υπολογισμού είναι η ίδια, είτε έχουμε τέσσερις, είτε έξι είτε οχτώ κόμβους. Αυτό σημαίνει ότι αν το Ερώτημα 1 χρειάζεται λιγότερο χρόνο για να εκτελεστεί από το Ερώτημα 2 σε ένα cluster αποτελούμενο από 4 κόμβους, τότε θα χρειαστεί λιγότερο χρόνο και αν εκτελεστεί σε cluster έξι ή οχτώ κόμβων. Τα ίδια συμπεράσματα ισχύουν και για το μεγαλύτερο σύνολο δεδομένων.

Στον πίνακα 4.2 φαίνονται οι διαφορετικοί τύποι ερωτημάτων, ταξινομημένοι σε μια σειρά ξεκινώντας από αυτόν που απατεί το λιγότερο χρόνου υπολογισμού και καταλήγοντας σε αυτόν που απατεί τον περισσότερο.



Σχήμα 4.8: Χρόνος Υπολογισμού - Σύνολο Δεδομένων 1

Query3	Query7	Query1	Query4	Query6	Query5	Query2

Πίνακας 4.2: Τα ερωτήματα ταξινομημένα με βάση το χρόνο υπολογισμού

Εφόσον, αυτή η συμπεριφορά ισχύει σε κάθε περίπτωση, είναι ασφαλές να υποθέσουμε ότι δεν είναι τυχαία, αλλά βασίζεται στα εγγενή χαρακτηριστικά των ερωτημάτων αυτών. Τα χαρακτηριστικά που φαίνεται να επηρεάζουν περισσότερο το χρόνο υπολογισμού είναι:

1. Ο αριθμός των supersteps
2. Η πολυπλοκότητα του ερωτήματος, που καθορίζεται κυρίως από τον αριθμό των κορυφών και των ακμών που περιλαμβάνονται στο ερώτημα καθώς και τον τρόπο με τον οποίο αυτές συνδέονται μεταξύ τους.

Σχετικά με τον πρώτο παράγοντα, είναι λογικό να υποθέσουμε ότι όσο μικρότερος είναι ο αριθμός των απαιτούμενων supersteps, τόσο μικρότερος θα είναι και ο χρόνος υπολογισμού. Αυτό συμβαίνει γιατί κάθε superstep συνοδεύεται και από μια φάση συγχρονισμού, όπου ο κάθε εργάτης περιμένει όλους τους υπόλοιπους να ολοκληρώσουν, πράγμα που μπορεί να οδηγήσει σε πολύ μεγαλύτερο χρόνο υπολογισμού, κυρίως σε περιπτώσεις όπου δεν έχουμε ίση κατανομή φορτίου ανάμεσα στους κόμβους, ή αν ένας κόμβος είναι πιο αργός από τον άλλο. Στην περίπτωσή μας τα Ερωτήματα 1, 2, 3 και 7 χρειάζονται δύο supersteps για να ολοκληρωθούν, ενώ τα 4,5 και 6 χρειάζονται τρία. Βλέπουμε ότι γενικά τα ερωτήματα με δύο supersteps ολοκληρώνονται γρηγορότερα, με εξαίρεση το ερώτημα 2, το οποίο έχει το μεγαλύτερο χρόνο υπολογισμού από όλα, παρόλο που για την ολοκλήρωσή του απαιτούνται μόνο δύο supersteps. Σε αυτήν την περίπτωση, άλλοι παράγοντες, επηρεάζουν περισσότερο το αποτέλεσμα.

Η πολυπλοκότητα φαίνεται να παίζει επίσης πολύ σημαντικό ρόλο στον χρόνο υπολογισμού του ερωτήματος. Παίρνοντας ένα ένα τα ερωτήματα έχουμε:

Το ερώτημα 3, είναι το απλούστερο και αναμένεται να απαιτεί το λιγότερο χρόνο υπολογισμού. Συγκεκριμένα, απλά απαιτεί από μία ακριβώς κορυφή με συγκεκριμένο VertexID να στείλει μήνυμα ή μηνύματα σε όλες τις κορυφές που συνδέονται με αυτήν μέσω ακμής με συγκεκριμένη τιμή. Όλες οι κορυφές που έλαβαν μήνυμα αποτελούν μέρος της απάντησης.

Το ερώτημα 7, είναι λίγο πιο πολύπλοκο από το 3, περιλαμβάνοντας μια ακόμα κορυφή και ακμή. Ο υπολογισμός γίνεται με τον ίδιο τρόπο που γίνεται και στο ερώτημα 3, με τη διαφορά ότι στο τέλος έχουμε μία επιπλέον συνθήκη: όλες οι κορυφές που έλαβαν μήνυμα για να αποτελέσουν μέρος της απάντησης πρεπει επιπλέον να είναι συνδεμένες με με μια άλλη συγκεκριμένη κορυφή μέσω συγκεκριμένης ακμής.

Το ερώτημα 1, δεν είναι πιο πολύπλοκο από το 7, με την έννοια ότι περιλαμβάνει τον ίδιο αριθμό κορυφών και ακμών. Όμως, ο τρόπος με τον οποίο οι κορυφές συνδέοονται μεταξύ τους είναι διαφορετικός, εμπεριέχοντας περισσότερες ενεργές κορυφές στο αρχικό superstep και έναν αυξημένο αριθμό μηνυμάτων που στέλνεται μεταξώ των εργατών, οδηγώντας έτσι σε μεγαλύτερο χρόνο υπολογισμού.

Το ερώτημα 4, αυξάνει αισθητά την πολυπλοκότητα, καθώς περιέχει τέσσερις ιδιότητες (ακμές), τρείς κορυφές με συγκεκριμένη τιμή και ένα σύνολο από λορυφές Y, που συνδέονται με άλλες κορυφές με συγκεκριμένο τρόπο. Μοιάζει με συνδυασμό από το ερώτημα 7 και 1.

Το ερώτημα 6, έχει τον ίδιο αριθμό κορυφών και ακμών με το 4, έτσι η πολυπλοκότητα δεν είναι αισθητά αυξημένη. Διαφέρει στον τρόπο σύνδεσης των κορυφών μεταξύ τους, και μπορεί να θεωρηθεί ως συνδυασμός δύο ερωτημάτων τύπου 1, δικαιολογώντας έτσι τον λίγο περισσότερο χρόνο υπολογισμού που απαιτείται (σε σύγκριση με το 4).

Το ερώτημα 5, είναι πιο πολύπλοκο από όλα τα υπόλοιπα, περιλαμβάνοντας μια ακόμα ιδιότητα, καθώς και ένα σύνολο από κορυφές ‘Z’ με συγκεκριμένα χαρακτηριστικά. Συνεπώς απαιτεί ακόμα περισσότερο χρόνο υπολογισμού.

Τέλος, το ερώτημα 2, το οποίο απαιτεί την περισσότερη ώρα συγκριτικά με όλα τα υπόλοιπα. Απαιτεί μόνο δύο supersteps και ο τρόπος με τον οποίο συνδέονται μεταξύ τους οι κορυφές και οι ακμές είναι αρκετά απλός. Όμως, ο αριθμός των ιδιοτήτων που εμπεριέχονται είναι μεγάλος (πέντε), και κατά συνέπεια απαιτείτα η ανταλλαγή μεγάλου αριθμούν μηνυμάτων, τα οποία μάλιστα θα διαχειριστεί ένας εργάτης, μειώνοντας έτσι το επίπεδο παραλληλισμού και το χρόνο που απαιτείται για την ολοκλήρωσή του.

4.2.3 Χρόνος Υπολογισμού και Αριθμός κόμβων

Προφανώς, εφόσον είμαστε σε κατανεμημένο περιβάλλον θέλουμε να δούμε πώς μεταβάλλεται ο χρόνος υπολογισμού καθώς αυξάνεται ο αριθμός των κόμβων. Στα πειράματά μας, τρέξαμε όλους τους διαφορετικούς τύπους ερωτημάτων, με διαφορετικές επιλογές για υποκείμενα, κατηγορούμενα και αντικείμενα, σε διαφορετικό αριθμό κόμβων. Τρέξαμε το κάθε ερώτημα πέντε φορές, και πήραμε το μέσο όρο ως το τελικό αποτέλεσμα.

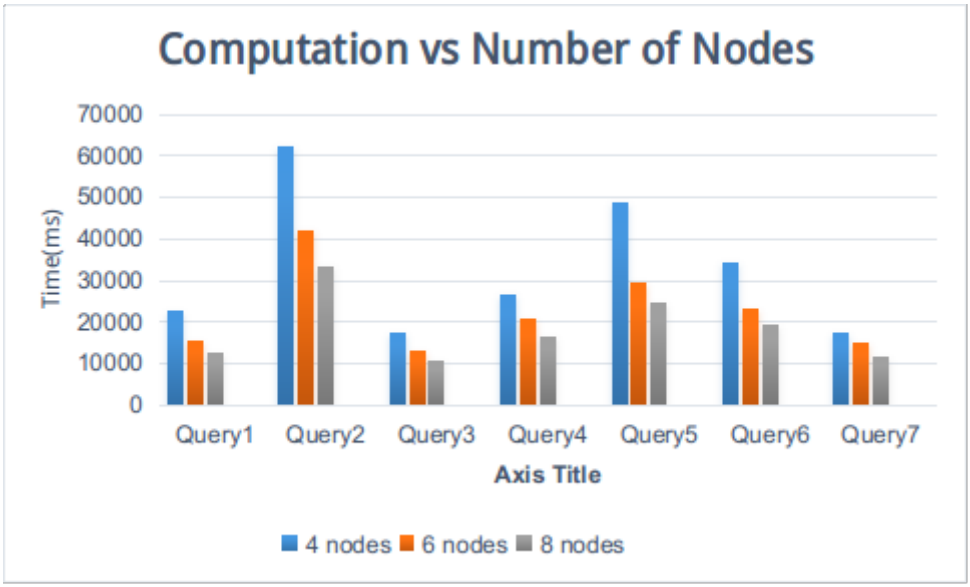
Τα αποτελέσματα και απεικονίζονται στο σχήμα 4.9.

Όπως είναι αναμενόμενο για όλα τα είδη ερωτημάτων, ο χρόνος υπολογισμού μειώνεται καθώς ο αριθμός των κόμβων αυξάνεται για όλους τους τύπους ερωτημάτων.

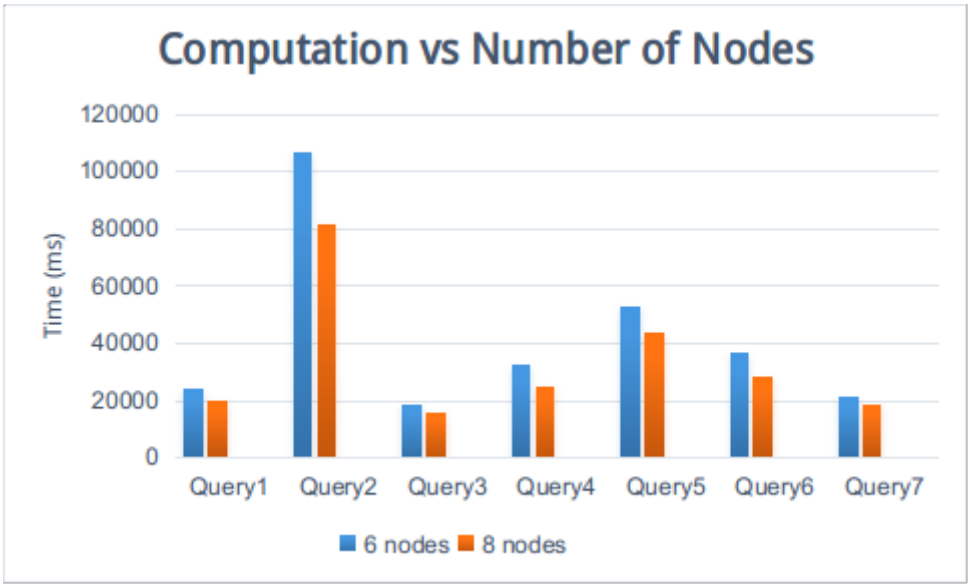
Τα αποτελέσματα για το μεγαλύτερο σύνολο δεδομένων φαίνονται στο σχήμα 4.10. Σε αυτήν την περίπτωση τα δεδομένα αποδείχτηκε ότι ήταν περισσότερα από όσα μπορούσε να διαχειριστεί η μνήμη μας.

Χρόνος Υπολογισμού και Διαφορετικά Στιγμιότυπα του ίδιου είδους Ερωτήματος

Για να δούμε πώς η επιλογή των υποκειμένων/κατηγορουμένων/αντικειμένων επηρεάζει το αποτέλεσμα, τα Ερωτήματα 1,2 και 3 εκτελέστηκαν χρησιμοποιώντας διαφορετικές τιμές για διαφορετικό



Σχήμα 4.9: Χρόνος υπολογισμού για διαφορετικό αριθμό κόμβων - Σύνολο Δεδομένων 1



Σχήμα 4.10: Χρόνος υπολογισμού για διαφορετικό αριθμό κόμβων - Σύνολο Δεδομένων 2

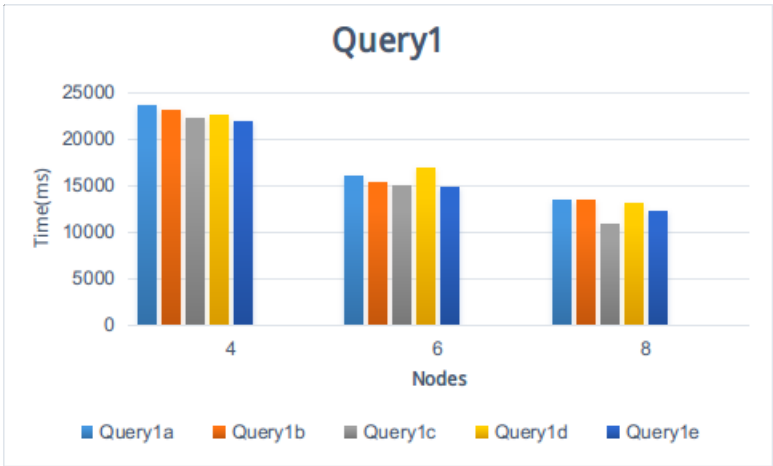
αριθμό κόμβων (4, 6 και 8 κόμβοι). Τα αποτελέσματα απεικονίζονται στα σχήματα 4.11, 4.12 και 4.13.

Όμοια, για το Σύνολο Δεδομένων 2, έχουμε τα σχήματα 4.14, 4.15 και 4.16 αντίστοιχα:

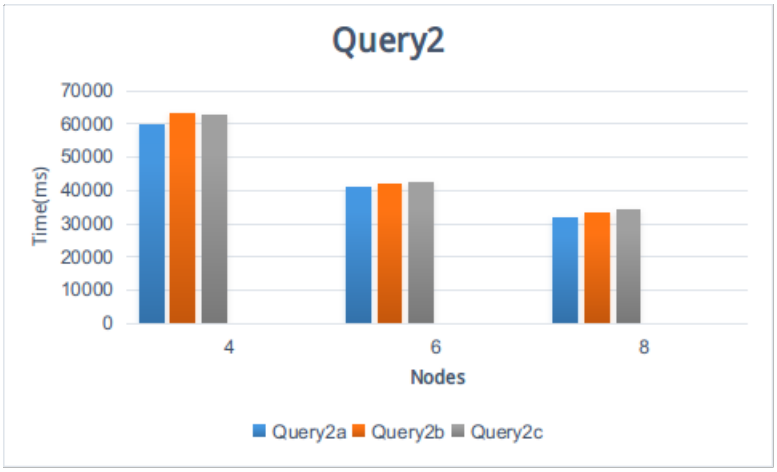
Από τα παραπάνω σχήματα μπορούμε να δούμε πώς η επιλογή των υποκειμένων, κατηγορουμένων και αντικειμένων μπορεί να οδηγήσει σε πολύ διαφορετικό χρόνο υπολογισμού. Σε όλες αυτές τις περιπτώσεις, η δομή και κατ'επέκταση γραφική αναπαράσταση του ερωτήματος παραμένει ίδια και αλλάζουν μόνο οι διάφορες τιμές των κορυφών ή/και ακμών. Στον ακόλουθο πίνακα 4.3 μπορούμε να δούμε τα διάφορα στιγμιότυπα από τον ίδιο τύπο ερωτήματος, ταξινομημένα κατά αύξοντα χρόνο υπολογισμού.

Αυτές οι διαφορές μπορούν να δικαιολογηθούν στις περισσότερες περιπτώσεις από τον αριθμό των κορυφών που εμπλέκονται στο ερώτημα.

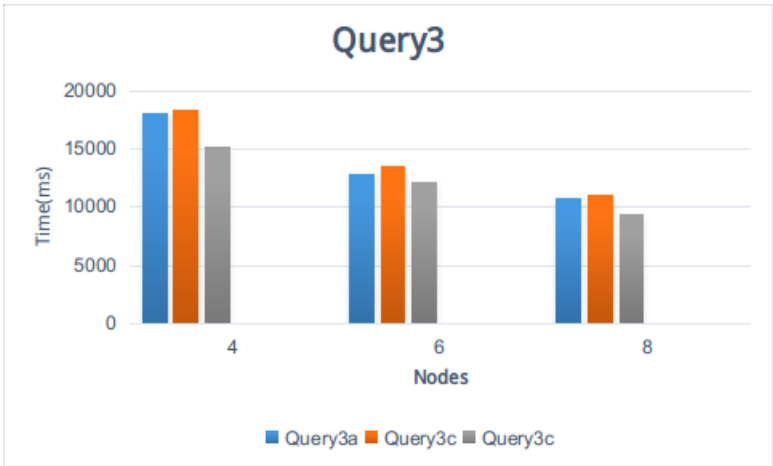
Όπως βλέπουμε στα σχήματα 4.17, 4.18 και 4.19, για το Ερώτημα 1 και 3, όσο μικρότερος είναι ο



Σχήμα 4.11: Ερώτημα 1 - Χρόνοι Υπολογισμού για το Σύνολο Δεδομένων 1

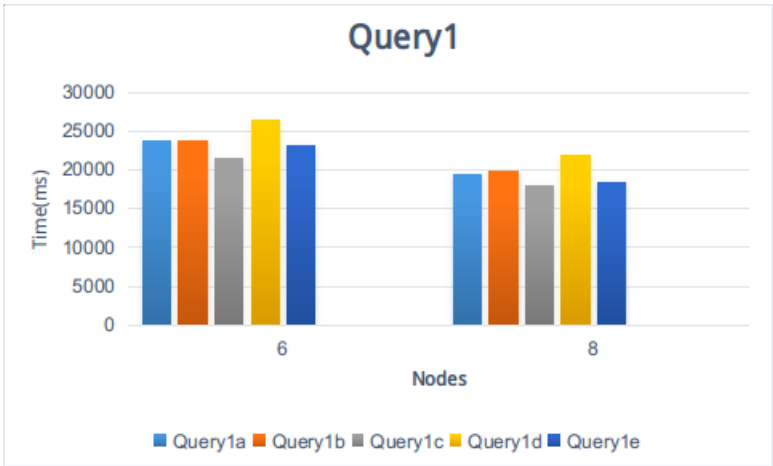


Σχήμα 4.12: Ερώτημα 2 - Χρόνοι Υπολογισμού για το Σύνολο Δεδομένων 1

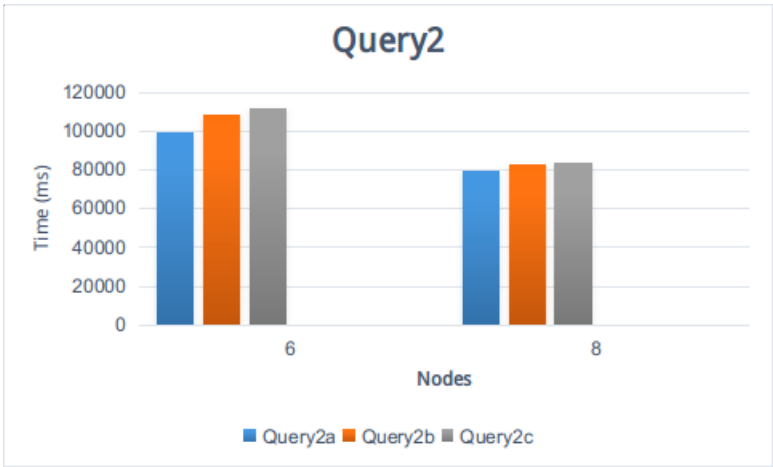


Σχήμα 4.13: Ερώτημα 3 - Χρόνοι Υπολογισμού για το Σύνολο Δεδομένων 1

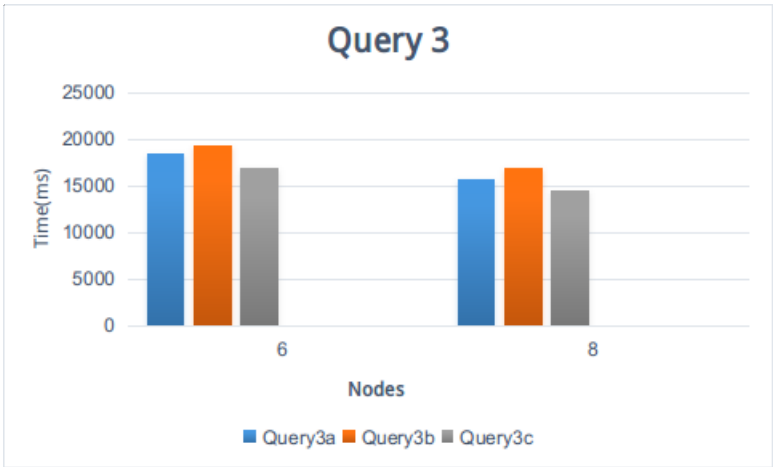
αριθμός των κορυφών, τόσο μικρότερος είναι και ο χρόνος υπολογισμού. Τα παραπάνω δεν ισχύουν για το ερώτημα 2, όπου το ερώτημα 2α χρειάζεται το λιγότερο χρόνο για να απαντηθεί, παρόλο που οι κορυφές που εμπλέκονται στο ερώτημα είναι περισσότερες και τα μηνύματα που ανταλλάχθηκαν είναι περισσότερα. Σε αυτήν την περίπτωση σημαντικότερο ρόλο πρέπει να παίζουν άλλοι παράγοντες



Σχήμα 4.14: Ερώτημα 1 - Χρόνοι Υπολογισμού για το Σύνολο Δεδομένων 2



Σχήμα 4.15: Ερώτημα 2 - Χρόνοι Υπολογισμού για το Σύνολο Δεδομένων 2



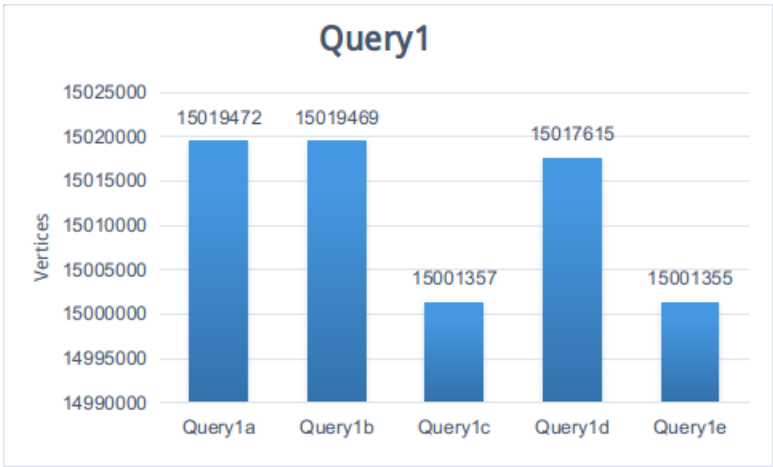
Σχήμα 4.16: Ερώτημα 3 - Χρόνοι Υπολογισμού για το Σύνολο Δεδομένων 2

όπως το πώς είναι χωρισμένος ο γράφος και πώς έχουν καταναμεμηθεί τα δεδομένα στους εργάτες.

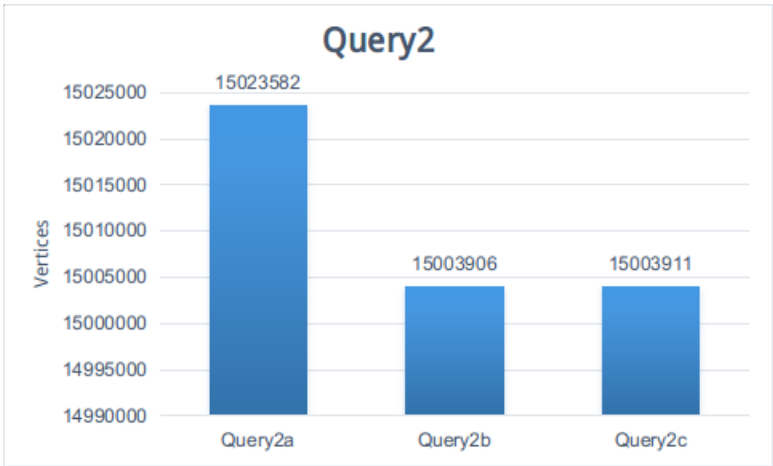
Ακριβώς η ίδια συμπεριφορά παρατηρείται και στο δεύτερο σύνολο δεδομένων που χρησιμοποιήσαμε.

Query1	Query1e	Query1c	Query1d	Query1b	Query1a
Query2	Query2a	Query2b	Query2c		
Query3	Query3c	Query3a	Query3b		

Πίνακας 4.3: Ερωτήματα ταξινομημένα ανά αύξοντα χρόνο υπολογισμού



Σχήμα 4.17: Ερώτημα 1 - Αριθμός Ακμών



Σχήμα 4.18: Ερώτημα 2 - Αριθμός Ακμών

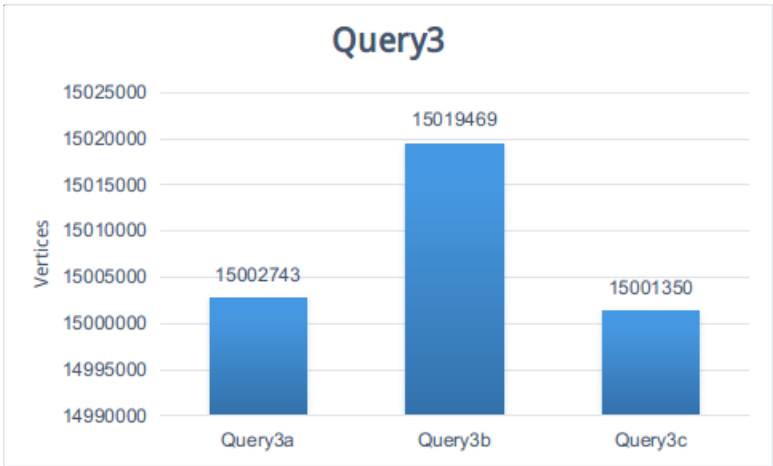
4.2.4 Openvirtuoso

Για να επιβεβαιώσουμε ότι η υλοποίηση των ερωτημάτων με κατανημένο τρόπο επιστρέφει όλα τα αποτελέσματα, αλλά και τα σωστά, τα συγκρίναμε με αυτά από το Openvirtuoso. Σε όλες τις περιπτώσεις τα αποτελέσματα ήταν τα ίδια.

Επόσης, θέλαμε να δούμε αν τα ερωτήματα συμπεριφέρονται με τον ίδιο τρόπο και στα δύο συστήματα και αν όχι, το λόγο που κρύβεται πίσω από αυτή τη διαφορετική συμπεριφορά.

4.2.5 Φόρτωση Γράφου

Ο χρόνος που απαιτείται από το Openvirtuoso για να φορτώσει όλα τα RDF δεδομένα, είναι αρκετά μεγαλύτερος. Η φόρτωση του γράφου στο Openvirtuoso έγινε 7 φορές και πήραμε ως αποτέλεσμα το μέσο όρο. Τον γράφο αποτελούν όλα τα RFD αρχεία που χρησιμοποιήσαμε και στο Giraph. Φαίνεται



Σχήμα 4.19: Ερώτημα 3 - Αριθμός Ακμών

ότι χρειάστηκαν περίπου 6 ώρες για τη φόρτωση ολόκληρου του γράφου. Τα μεμονωμένα αποτελέσματα για το σύνολο δεδομένων 1, φαίνεται στον πίνακα 4.4. Παρόλο που κ σύκριση δεν είναι δίκαιη, με την έννοια ότι στο Giraph χρησιμοποιήσαμε τέσσερις φορές περισσότερους υπολογιστικούς πόρους, οι 6 ώρες συνεχίζουν να παραμένουν μια αρκετά μεγάλη τιμή, συγκριτικά με τα δύο λεπτά που χρειάζεται το Giraph για το ίδιο σύνολο δεδομένων, όταν το ρυθμίσουμε με τις κατάλληλες παραμέτρους.

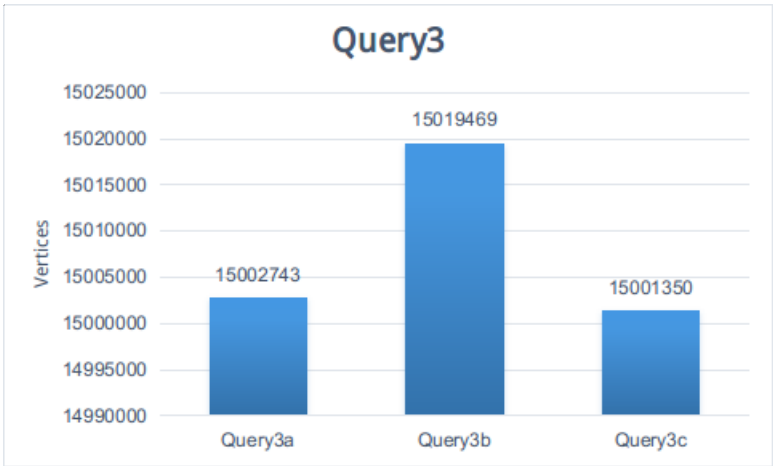
Χρόνος Φόρτωσης (ms)	
Openvirtuoso	**Giraph**
23233681	118553
25289059	105188
12021112	128640
25706906	118150
23536321	134792
18211069	119050
22010159	114458
21429758.14	**119833**

Πίνακας 4.4: Χρόνος Φόρτωσης - Openvirtuoso - Giraph

4.2.6 Χρόνος Υπολογισμού

Από την άλλη πλευρά, ο χρόνος απόκρισης του Openvirtuoso είναι πολύ καλύτερος από ότι στο Giraph, όπου μπορεί να είναι έως και 5 με 6 φορές πιο αργός, παρόλο που χρησιμοποιούμε και τους 8 κόμβους. Αυτό όμως είναι αναμενόμενο σε κάποιο βαθμό, δεδομένου ότι το Giraph είναι κατάλληλο για την επεξεργασία τεράστιων συνόλων δεδομένων, τα οποία μπορεί να βρίσκονται σε εκατοντάδες κόμβους. Υπό τέτοιες συνθήκες, υποθέτουμε ότι το Giraph θα έχει υψηλότερες επιδόσεις έναντι του Openvirtuoso, και επιπλέον θα είναι σε θέσει να απαντήσει ερωτήματα τα οποιά θα είναι αδύνατο να λυθούν με το Openvirtuoso. Στα δικά μας μεγέθη όμως, το κόστος της επικοινωνίας και η επιπλέον διαχείρηση που απαιτείται λόγω του κατανεμημένου περιβάλλοντος φαίνεται να καθυστερούν το χρόνο υπολογισμού.

Ακόμη, εξετάσαμε το πώς συμπεριφέρεται το Openvirtuso με διαφορετικούς τύπους ερωτημάτων. Τα αποτελέσματα για το πρώτο σύνολο δεδομένων φαίνονται στο σχήμα 4.20.



Σχήμα 4.20: Χρόνος Υπολογισμού στο Openvirtuoso

Βλέπουμε ότι η συμπεριφορά των ερωτημάτων είναι αρκετά διαφορετική από αυτήν στο Giraph. Στον πίνακα 4.5 βλέπουμε τα ερωτήματα ταξινομημένα ανά αύξοντα χρόνο υπολογισμού.

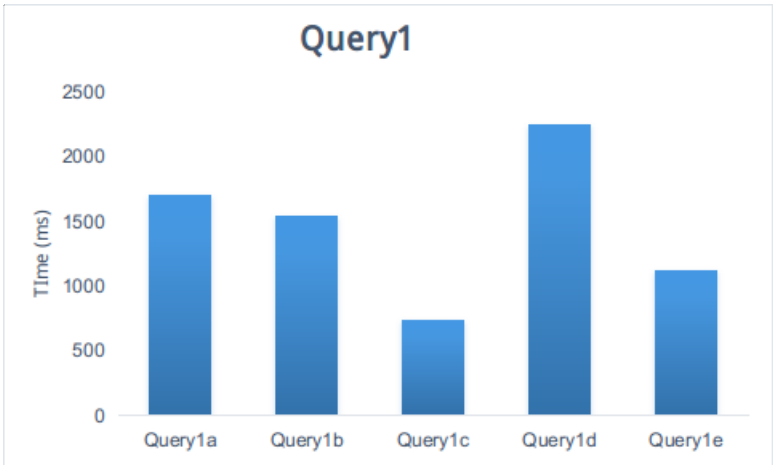
Query7	Query2	Query5	Query1	Query6	Query3	Query4

Πίνακας 4.5: Ερωτήματα ταξινομημένα ανά αύξοντα χρόνο υπολογισμού

Αυτές οι διαφορές οφείλονται στο διαφορετικό τρόπο με τον οποίο το Giraph και τον Openvirtuoso επεξεργάζονται το ερώτημα, καθώς και το γεγονός ότι η φάση του γραψίματος του αποτέλεσματος στην έξοδο απαιτεί περισσότερο χρόνο συγκριτικά με το Giraph. Στο Giraph, ο κάθε εργάτης μπορεί να γράψει στο HDFS το μερικό αποτέλεσμα, κάτι που στο Openvirtuoso πρέπει να γίνει σειριακά.

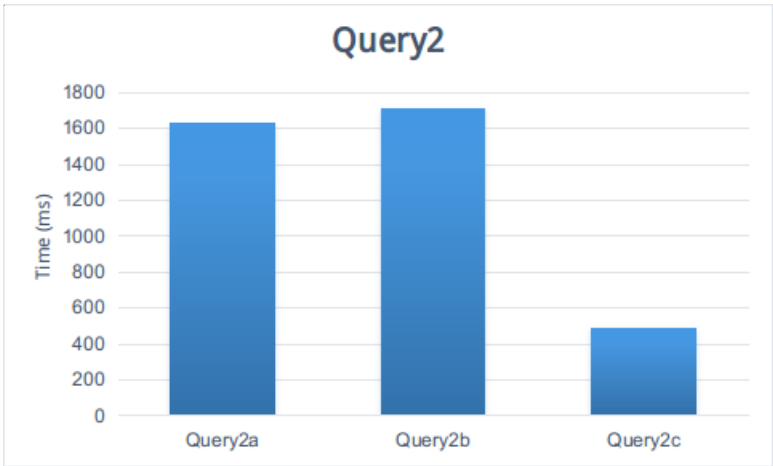
Αυτός είναι και ο λόγος που το Ερώτημα 3, αν και το απλούστερο, είναι αυτό που χρειάζεται το δεύτερο μεγλύτερο χρόνο για να ολοληρωθεί, ενώ το Ερώτημα 7 που είναι ελάχιστα πιο πολύπλοκο αλλά έχει μόνο ένα υποκείμενο ως έξοδο, χρειάζεται τη λιγότερη ώρα.

Τα σχήματα 4.21, 4.22 και 4.23 δείχνουν πώς κυμαίνεται ο χρόνος υπολογισμού για διαφορετικά στιγμιότυπα του ίδιου τύπου ερωτήματος.

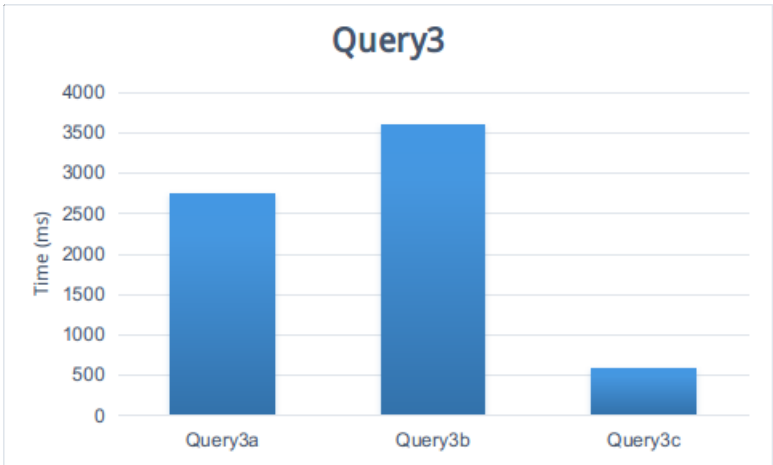


Σχήμα 4.21: Ερώτημα 1 στο Openvirtuoso

Εδώ βλέπουμε ότι οι διαφορές στο χρόνο υπολογισμού μοιάζουν περισσότερο με αυτές στο Giraph, και οι όποιες αποκλίσεις πρέπει να οφείλονται στο γράψιμο του αποτελέσματος, ιδίως όταν πολλές γραμμές επιστρέφονται ως αποτέλεσμα.



Σχήμα 4.22: Ερώτημα 2 στο Openvirtuoso



Σχήμα 4.23: Ερώτημα 3 στο Openvirtuoso

4.2.7 Πληρότητα και Ορθότητα του Ερωτήματος

Τέλος, συγκρίναμε τα αποτελέσματα που πήραμε από το Giraph και το Openvirtuoso, έτσι ώστε να επιβεβαιώσουμε ότι η εφαρμογή μας επέστρεψε όλα τα αποτελέσματα και τα σωστά. Η επιβεβαίωση έγινε και για τα δύο σύνολα δεδομένων.

Κεφάλαιο 5

Συμπεράσματα και Μελλοντικές επεκτάσεις

5.1 Συμπεράσματα

Σε αυτή τη διπλωματική εργασία, πρώτα παρουσιάσαμε τη ραγδαία αύξηση των δεδομένων σις μέρες μας και εκφράσαμε την ανάγκη για την αποθήκευσή και την αξιοποίησή τούς, ώστε να μπορέσουμε να εξάγουμε πολύτιμη πληροφορία και να βελτιώσουμε διάφορους τομείς της ζωής μας. Παρουσιάσαμε διάφορα υπολογιστικά μοντέλα, όπως το Cloud Computing και το Distributed Computing, καθώς σήμερα τα περισσότερα συστήματα βασίζονται σε αυτά, ώστε να επιτύχουν επεξεργασία των δεδομένων με αποδοτικό τρόπο. Στη συνέχεια, παρουσιάσαμε τα πιο δημοφιλή εργαλεία για κατανεμημένη επεξεργασία, όπως το Hadoop και το Giraph, καθώς και τα προγραμματιστικά μοντέλα που υλοποιούν.

Στη συνέχεια, παρουσιάσαμε την δική μας συνεισφορά, έναν απλό επεξεργαστή ερωτημάτων, στόχος του οποίου είναι να αξιοποήσει το κατανεμημένο περιβάλλον ώστε να απαντήσει διαφόρων ειδών ερωτήματα με αποδοτικό τρόπο. Δείξαμε διαφορετικά είδη ερωτημάτων και εισάγαμε αποδοτικές μεθοδολογίες για την επίλυσή τους. Περιγράψαμε τον τρόπο με τον οποίο υλοποιήσαμε τους αλγόριθμούς μας, με τη λογική 'Think Like A Vertex' και διεξάγαμε πειράματα για να μελετήσουμε την απόδοση και την κλιμακωσιμότητα του συστήματός μας σε μια συστάδα υπολογιστών αποτελούμενη από οκτώ κόμβους. Επιπλέον, επιβεβαιώσαμε την πληρότητα και την εγκυρότητα των αποτελεσμάτων μας, μέσω της σύγκρισής τους με αυτών που επιστρέφει ένα κεντρικό σύστημα.

Από τα αποτελέσματα των πειραμάτων μας, εξάγαμε το συμπέρασμα πώς είναι εφικτό να απαντηθούν πλήρως διαφόρων ειδών SPARQL ερωτήματα πάνω σε Linked Data, με κατανεμημένο τρόπο. Ταυτόχρονα είδαμε πώς ο χρόνος υπολογισμού του αποτελέσματος βελτιώνεται καθώς αυξάνεται ο αριθμός των υπολογιστικών πόρων, πράγμα που σημαίνει ότι μπορούμε να βελτιώσουμε την απόδοση επίλυσης των ερωτημάτων απλά χρησιμοποιώντας παραπάνω υπολογιστικούς πόρους εφόσον και όταν τους έχουμε στη διάθεσή μας, πράγμα που δεν είναι εφικτό με κεντρικοποιημένες προσεγγίσεις που εκτελούνται σε μεμονωμένους υπολογιστικούς πόρους. Επιπλέον, είδαμε ότι διαφορετικά είδη ερωτημάτων, διαφορετικά σύνολα δεδομένων ως είσοδος, καθώς και διαφορετικά στιγμιότυπα του ίδιου τύπου ερωτήματος, μπορούν να οδηγήσουν σε διαφορετικού χρόνους υπολογισμού του αποτελέσματος. Τέλος, με μία μικρή σύγκριση με ένα ευρέως γνωστό κεντρικό σύστημα, το Openvirtuoso, είδαμε ότι το Giraph είναι πολύ γρηγορότερο κατά τη φόρτωση του γράφου, ενώ για το χρόνο του υπολογισμού τα αποτελέσματα είναι τα ακριβώς αντίθετα και χρειάζονται περαιτέρω διερεύνηση. Όμως, ο μεγαλύτερος χρόνος υπολογισμού που απαιτεί το Giraph, είναι δικαιολογημένος για μικρά σύνολα δεδομένων όπως αυτά που χρησιμοποιήσαμε, και η πραγματική του δύναμη θα μπορούσε να φανεί με τη χρήση περισσότερων υπολογιστικών πόρων για την απάντηση ερωτημάτων σε αρκετά μεγαλύτερα σύνολα δεδομένων.

5.2 Μελλοντικές επεκτάσεις

Υπάρχουν αρκετές βελτιώσεις που θα μπορούσαν να γίνουν στο μέλλον ώστε το σύστημά μας να έχει μεγαλύτερες δυνατότητες αλλά και να βελτιώσουμε την απόδοσή του κατά την επίλυση ερωτημάτων. Νέες λειτουργίες μπορούν να προστεθούν που δίνουν τη δυνατότητα για εκτέλεση πολυπλοκότερων ερωτημάτων με αυτοματοποιημένο τρόπο για την επίλυση των οποίων χρειάζεται πληροφορία για την ιεραρχία καθώς και λογική συνεπαγωγή. Από την άλλη θα μπορούσαν να γίνουν βελτιώσεις και στον τρόπο υπολογισμού του ερωτήματος. Αντί να έχουμε πολλούς διαφορετικούς τρόπους επίλυσης, καθένας κατάλληλος για ένα συγκεκριμένο είδος ερωτήματος, θα ήταν προτιμότερο να έχουμε μια γενικότερη προσέγγιση, όπου ο αλγόριθμός μας θα είναι ικανός να αναλύσει τον τύπο του ερωτήματος και τις παραμέτρους του, και θα προχωράει στην εύρεση του αποτελέσματος, χωρίς να χρειάζεται παραπάνω πληροφορία από το χρήστη. Παρόμοια, στη διεπαφή με το χρήστη, θα ήταν προτιμότερο, να μπορεί αυτός να εισάγει το επιθυμητό ερώτημα, αντί να καλείται να περιορίζεται σε ένα συγκεκριμένο τύπο από αυτά.

Σχετικά με την απόδοσή του συστήματός μας, θα ήταν χρήσιμο να δοκιμάσουμε διάφορες επιλογές ακόμη στις παραμέτρους που μπορούμε να ρυθμίσουμε είτε στο Hadoop είτε στο Giraph. Επιπλέον, θα μπορούσαμε να έχουμε αρκετά διαφορετικά αποτελέσματα χρησιμοποιώντας διαφορετικό είδος προεπεξεργασίας, (π.χ ομαδοποιώντας τις RDF τριπλέτες ανά υποκείμενο, κατηγορούμενο) ή δοκιμάζοντας διαφορετικούς τρόπους διαμοιρασμού του γράφου ανάμεσα στους διάφορους υπολογιστικούς κόμβους. Επίσης, πολλά περιθώρια έρευνας, έχουμε και στο πεδίο της κλιμακωσιμότητας, καθώς η χρήση μεγαλύτερων και διαφορετικού είδους συνόλων δεδομένων, με τη χρήση ακόμα περισσότερων υπολογιστικών πόρων, θα μπορούσε να αναδείξει περισσότερο τις δυνατότητες του Giraph για επεξεργασία γράφων.Τέλος, πέρα από το Giraph, θα ήταν χρήσιμη η δοκιμή άλλων κατανεμημένων συστημάτων για την επεξεργασία γράφων και η σύγκρισή της απόδοσής τους.

Βιβλιογραφία

[1] "Mapreduce dataflow." http://www.mikepluta.com/hadoop-v1-architecture-overview.

[2] "Apache hadoop yarn." http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[3] "Bsp wikipedia." https://en.wikipedia.org/wiki/Bulk_synchronous_parallel.

[4] A. Ching, "Dynamic graph/iterative computation on apache giraph," *Giraph at Hadoop Summit*, Juny 6 2014.

[5] "Giraph execution example." http://www.ibm.com/developerworks/library/os-giraph, June 10 2013.

[6] "The linking open data cloud diagram." http://lod-cloud.net, 2014.

[7] P. Mell and T. Grance, "Special publication 800-145: The nist definition of cloud computing. recommendations of the national institute of standards and technology," tech. rep., U.S. Department of Commerce, Gaithersburg, MD 20899-8930, September 2011.

[8] A. M. Fox A., Griffith R. and et al., "Above the clouds: A berkeley view of cloud computing," Tech. Rep. No. UCB/EECS-2009-28, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, February 10 2009.

[9] "Apache spark." http://spark.apache.org/.

[10] "Apache storm." http://storm.apache.org.

[11] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," *In SIGMOD Conference*, pp. 135–136, June 6-11 2010.

[12] "Google bigquery." https://developers.google.com/bigquery.

[13] "Apache giraph." http://giraph.apache.org.

[14] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[15] "Apache hadoop." http://hadoop.apache.org.

[16] T. White, *Hadoop: The Definite Guide*. O'Reilly Media, Inc., 4th ed., April 1 2015.

[17] "Mapreduce wikipedia." https://en.wikipedia.org/wiki/MapReduce.

[18] "Jobtracker wiki." https://wiki.apache.org/hadoop/JobTracker.

[19] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," *In HPDC*, pp. 810–818, 2010.

[20] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "The HaLoop approach to large-scale iterative data analysis," *The VLDB Journal*, vol. 21, no. 2, pp. 169–190, 2012.

[21] P. Stutz, A. Bernstein, and W. Cohen, "Signal/collect: Graph algorithms for the (semantic) web," *In International Semantic Web Conference (ISWC)*, pp. 764–780, 2010.

[22] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," *6th Biennial Conference on Innovative Data Systems Research (CIDR)*, pp. 716–727, 2013.

[23] "Apache zookeeper." https://zookeeper.apache.org/.

[24] "Linked data wikipedia." https://en.wikipedia.org/wiki/Linked_data.

[25] "Tim berners-lee. linked data - design issues." https://www.w3.org/DesignIssues/LinkedData.html, 2006.

[26] "Dbpedia." https://www.w3.org/standards/semanticweb/data.

[27] "Lubm." http://swat.cse.lehigh.edu/projects/lubm.

[28] "Virtuoso open-source." https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main.

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

Study of the performance of distributed graph databases for the management of Linked Data

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΖΑΝΑΚΗ ΒΑΣΙΛΙΚΗ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2016

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

Study of the performance of distributed graph databases for the management of Linked Data

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΖΑΝΑΚΗ ΒΑΣΙΛΙΚΗ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31η Οκτωβρίου 2016.

Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επ. Καθηγητής Ε.Μ.Π.	Δημήτριος Τσουμάκος Επ. Καθηγητής Ιόνιο Πανεπιστήμιο

Αθήνα, Οκτώβριος 2016

Τζανάκη Βασιλική

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

Copyright © Τζανάκη Βασιλική, 2016.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

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

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

Περίληψη

Η πρόκληση του σημερινού κόσμου είναι τα δεδομένα. Ένας τεράστιος όγκος δεδομένων παράγεται κάθε δευτερόλεπτο και η δημιουργία τους συνεχίζει να αυξάνεται με ραγδαίο ρυθμό. Εικόνες, βίντεο, μουσική, έγγραφα, emails, αρχεία καταγραφής σφαλμάτων, δεδομένα από αισθητήρες και δορυφόρους, είναι μερικά μόνο από τα δεδομένα που παράγονται από τους ανθρώπους είτε από μηχανές, ως μέρος του Διαδικτύου των Πραγμάτων (Internet of Things). Όλα αυτά τα δεδομένα όμως δεν έχουν καμία αξία, αν δεν μπορούμε να τα μετατρέψουμε σε πολύτιμη πληροφορία και να τη χρησιμοποιήσουμε ώστε να βελτιώσουμε τις ζωές μας.

Ευτυχώς, τη σημερινη εποχή, κρατάμε στα χέρια μας πολύτιμα εργαλεία, τα οποία μπορούν να μας βοηθήσουν στην οργάνωση, στην αποθήκευση και στην ανάλυση όλων αυτών των δεδομένων. Η εισαγωγή του Υπολογιστικού Νέφους (Cloud Computing), καθώς και η ανάπτυξη διαφόρων κατανεμημένων συστημάτων και προγραμματιστικών μοντέλων μας έδωσε τη δυνατότητα να εξάγουμε ουσιαστική πληροφορία και να μετατρέψουμε όλα αυτά τα σκόρπια, φαινομενικά ασύνδετα δεδομένα, σε πολύτιμη γνώση.

Σε αυτή τη διπλωματική εργασία, επικεντρωνόμαστε περισσότερο στα δεδομένα με τη μορφή γράφων. Τα Διασυνδεδεμένα Δεδομένα (Linked Data), δηλαδή τα δεδομένα που δημοσιεύονται στις διάφορες ιστοσελίδες και συνδέονται μεταξύ τους, είναι μια κατηγορία δεδομένων που μπορεί να σχηματίσει τεράστιους γράφους. Αξιοποιούμε τις δυνατότητες που μας προσφέρει το Υπολογιστικό Νέφος, καθώς και τα υπάρχοντα κατανεμημένα συστήματα, ώστε να φορτώσουμε δεδομένα αυτής της μορφής και να υλοποιήσουμε αλγορίθμους που μας απαντάνε σε διαφόρων ειδών ερωτήματα πάνω σε τέτοιου είδους δεδομένα. Στη συνέχεια, μελετάμε την απόδοση του συστήματός μας καθώς ο αριθμός των υπολογιστικών πόρων αυξάνεται και ελέγχουμε την εγκυρότητα των αποτελεσμάτων μας, με την εκτέλεση των ίδιων ερωτημάτων πάνω στα ίδια δεδομένα σε ένα κεντρικό σύστημα. Επιπλέον, ερευνούμε πώς ένα ερώτημα διαφορετικού είδους ή η αντικατάσταση διαφόρων παραμέτρων του, μπορεί να επηρεάσει το χρόνο υπολογισμό του.

Λέξεις κλειδιά

Hadoop, Giraph, Pregel, Openvirtuoso, Υπολογιστικό Νέφος, Κατανεμημένα συστήματα, MapReduce, Bulk Synchronous Model, RDF, Διασυνδεδεμένα Δεδομένα, γράφοι, DBpedia, SPARQL.

Abstract

Today's world challenge is data. A huge amount of data is generated every second and the pace of data creation continues to accelerate rapidly. Images, videos, music, documents, emails, machine logs, sensor data, satellite data are a few only the data produced either by people or by machines as part of the Internt of Things (IoT). All this data though is worth nothing unless we are able to turn it into valuable information and use it in order to improve our lives.

Fortunately, nowadays we are holding in our hands precious tools, which can help us in the organization, storage and analysis of all this data. The introduction of the Cloud Computing model, as well as the development of various distributed frameworks and programming models has enabled us to extract meaningful information and transform all this scattered, seemingly unrelated data into valuable knowledge.

In this thesis, we are focusing more on graph data. Linked data, the data published in various websites and connected between them, are a category of data that can lead to huge graphs. We exploit the Cloud environment, as well as the existing distributed frameworks, in order to load data of this form and implement algorithms which allow us to answer various kinds of queries over such type of data. Then, we study the performance of our system as the number of computer resources increases and verify the validity of our results, by executing the same queries over the same datasets in a centralized system. Furthermore, we investigate how a different kind of query or the change of the various parameters involved, can influence the execution time.

Key words

Hadoop, Giraph, Openvirtuoso, Cloud Computing, Distributed systems, MapReduce, Bulk Synchronous Model, RDF, Linked data, graphs, DBpedia, Pregel, SPARQL.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and problem statement

From the first moments of mankind, we were called to deal with various problems, understand them, analyze them and figure out ways and technologies, which would help us to survive and improve our lives. Today's world challenge is data. A huge amount of data is generated every second and the pace of data creation continues to accelerate rapidly. Each and every one of us is responsible for this tremendous growth of data, by producing and releasing it in various forms through social media, documents, emails, photos, videos, sounds or business apps. Data is not generated only by people though. Although every individual's data footprint grows significantly every year, the amount of data generated by machines is going to be even greater [7]. Servers, Radio Frequence ID (RFID) readers, traffic recording devices, medical devices, satellites, sensors are only some paradigms of huge data generation with high velocity by machines.

Not all of the data are useful. Inside this flood of data, the race is on how to extract insight and value from this abuntant resource. So, the use of advanced systems for their organization and storage is essential in order to help us tackle this challenge. Single computers, no matter how powerful they are, are not able to process the huge amount of the available information in a reasonable amount of time. Thus, distributed systems are becoming more and more popular and new software tools and programming models, able to take advantage of this distributed environment, are developed.

One of the computing models that enables us to use distributed frameworks, is Cloud Computing. It has gained great popularity nowadays, since it can provide new capabilities to both Software as a Service (SaaS) providers and users, that were never feasible before. The formal definition of Cloud Computing as provided by the National Institute of Standards and Technology (NIST) [8] is the following:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

The benefits arising from this elasticity of resources are unprecedented in the history of IT, and have transformed a huge part of the IT industry, making Software as a Service even more attractive and changing completely the way IT hardware is designed and purchased [9]. Developers and companies do not need to be concerned about the large capital outlays in hardware in order to deploy their services or the human expense to operate it. They can start implementing innovative ideas or services, without the need to worry about overprovisioning for services whose popularity does not meet their expectations, or underprovisioning for the ones that gain popularity suddenly, since they can provision and release the desired resources at any point, without human-interaction with each service provider. And all kind of services (e.g. storage, processing, bandwidth, active user accounts) can be monitored, controlled and measured for both the provider and the consumer of the utilized services, thereby providing the capability to be charged in a pay-as-you-go manner.

The introduction of the Cloud Computing model requires to change the way software is written till now. A program does not have to be executed in a single computer for many hours, nor is there the need for the companies to be supplied with supercomputers. Many commodity computers communicating through a network can solve the same problem instead, in a much less amount of time. In order to seize this opportunity, we have to write programs that can scale. That means, large problems should be breaked down into several tasks, where each task is computed in the individual computers of the distributed system.

Without a doubt, writing distributed algorithms is much more complicated, needs algorithm research and demands cluster maintenance. However, the advantages of cost reduction, scalability and reliability are the ones that make distributed programming much more tempting. Inherently parallelizable programming models have been defined, interfaces that allow the programmer to implement the algorithm have been developed, as well as frameworks that run the program and provide fault-tolerance and scalability out of the box. One of the programming models that have gained great popularity nowadays is MapReduce with Hadoop being the framework which implements it in an efficient manner. Hadoop, however, is merely a starting point, other frameworks, like Spark [10] and Storm [11] have been developed as well, each one trying to provide its own way of processing big data.

Such tools along with many others that have been developed and the ones that are going to be introduced in the future, make it possible for the first time to process a large part of this ocean of data that surrounds us. Such processing can provide us with information, that has a great impact in our lives, from the way we live and work, to the way we run businesses, cities, governments. Companies can use this information to predict trends and costumer behaviours, governments to become more efficient, save money, identify fraud, even foil terrorist plots, and every individual will be able to see improvements in all aspects of our lives, like healthcare, medicine, finance, research, education, weather prediction, sports, telecoms, fashion, agriculture and everything in between.

With such valuable tools in our hands, the temptation to start processing everything around us is huge. The real challenge though, is not the data processing by itself, but how data can be translated into valuable knowledge. The way we collect it, scrutinize and interpret it can play a crucial role to the conclusions we reach. Most of the times, the data that are handled as isolated pieces of information have to offer nothing useful. The real treasure, seems to be hidden not in the data themselves, but in the relationships between them. Any kind of network (e.g. social networks, transportation networks, bibliographical networks, knowledge bases), may produce big amount of data interconnected in various ways. Such kind of data are able to form huge graphs which can provide us with a much better insight and understanding of the networks they were extracted from, after the appropriate processing.

The processing of the aforementioned graphs of data in powerful computing data centers can play a crucial role in various aspects. This was also the motive of writing this thesis: To find an efficient way to retrieve useful information from graph data. Search engines, adapted to the users preferences, (personalized information and recommendation systems) and decision-making systems, are only a few of the domains processing of graphs is needed. Graph data has a specific form and various systems have been developed for efficient storage and processing of such data in cloud infrastructures. Some of the most popular ones, are Google Pregel [12], Google Big Query [13], Apache Giraph [14] and Graphlab[15].

In this thesis, we are focusing more in Linked data, i.e., pieces of data published in various websites with connections among them. We are going to use a distributed system, load the graph data, make different kind of queries over datasets of different size, measure the time needed for the response and how it differentiates when the number of computer resources increases, when we make a different kind of query or when the relationships between the data included in the same query are different. Then, we are going to apply the above in a centralized system, so as to see if the behaviour is the same or if it differs and how. The use of an official centralized sytem will also allow us to verify the validity of the results produced by our own algorithms, running in the distributed system.

1.2 Outline

The reminder of this thesis is organized as follows:

Chapter 2 includes the essential background for our work. It presents the well-known Hadoop system, its architecture and the MapReduce Model. It explains, why this model is inappropriate for graph data and introduces us to Giraph and a more appropriate for graph-data programming model - the Bulk Synchronous Model (BSP).

Chapter 3 provides an overview of Linked Data, the reason of their importance, the sources that can produce it and the available ways to describe and query it. It also contains information regarding the various queries we chose to implement. It describes the different characteristics of each one of them and the way we implemented them in Giraph, using the BSP synchronous model.

Chapter 4 describes the experimental setup. The versions of Hadoop, Giraph, Openvirtuoso we installed and how the cluster was configured. It continues with the results of our experiments. The same queries were executed both in Giraph and Openvirtuoso. First, we studied the impact of various Hadoop configuration parameters on the time required to copy the input from the local filesystem to the distributed one, to load the graph and to answer the query. We continued, by examiming the effect of the type of the query, the use of different datasets and the use of different instances from the same type of query on the execution time, as well as how our system scales as the number of the available resources increases. Last, we compare the time required for loading the graphs in both systems and verified the validity and the completeness of our results.

Chapter 5 finally summarizes the above results and the conclusions made. In addition, it presents possible future extensions that could be done, in order to improve our system in terms of efficiency and functionality.

Chapter 2

Theoretical Background

The massive amounts of information available on the Internet nowadays requires the development of scalable platforms that can quickly process massive-scale data. Although, there has been significant work on parallel algorithms and frameworks on high performance computing (HPC) clusters, General Purpose Graphing Processing Units (GPGPUs) and multithreaded shared memory architectures, they have their own limitations, making distributed systems a better solution for big-data processing.

In the context of this thesis, we are going to use Apache Giraph[14], one of the most popular distributed frameworks for graph processing. In order to justify our choice and familiarize the reader with the main concepts concerning distributing systems, in this chapter we are going to provide the necessary background, describing the basic mechanisms and programming models used nowaways, as well as a brief reference to related work.

In specific, in Section 2.1, we are going to present Hadoop and the basic concepts around it. We are going to describe its advantages over parallel programming, present MapReduce, the programming model Hadoop implements, and give a rudimentary description of its architecture. Section 2.2, explains why MapReduce approach is inefficient for graph data, related work done on this field, and the reason Giraph is the system of our preference. Finally, Section 2.3, covers the basic concepts around Giraph, its own programming model of use (Bulk Synchronous Model, BSP) and the corresponding architecture.

2.1 Hadoop

Hadoop [16] is an open-source, Java-based programming framework that supports the processing of large data sets in a distributed computing environment. It was inspired by Google MapReduce and Google File System (GFS) papers.

It provides a distributed file system (HDFS) that stores data on the computed nodes, providing very high aggregate bandwidth across the cluster. In addition, Hadoop implements a parallel computational paradigm named MapReduce which divides the application into many small fragments of work, each of which may be executed or reexecuted on any node in the cluster.

Although the same problems can be solved with a different way, like parallel processing, Hadoop seems to be a better solution from various aspects:

First, developers should take over the splitting of the data, something which is not always easy or obvious. Inappropriate splitting, may lead to chunks of quite different size, and although the processes that have completed their part of computation earlier may be able to pick up further work, the whole run is dominated by the longest chunk. A better approach, would be to split the input into fixed-size chunks and assign each chunk to a process.

Second, the combination of the results from independent processes may require further processing.

Third, the processing capacity of a single machine is limited, not allowing you to go any faster given a specific dataset and number of processors. The use of multiple machines though, can lead to a much better performance with the right configuration.

Last, a great number of datasets, grow beyong the capacity of a single machine, making the use of multiple machines essential.

So, although it is feasible to parallelize the processing, in practice it's complicated. Using a framework like Hadoop to take care of these issues is a great help. Developers can work on the main task of processing the information, while splitting of the data, their distribution and parallelization of the jobs in a distributed environment as well as scalability and fault tolerance are now under the responsibility of MapReduce [7].

2.1.1 MapReduce

MapReduce [17] is a core component of the Apache Hadoop software framework. It is a programming model for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It works by breaking the processing into two basic phases: the map and the reduce phase. The map and reduce functions have the following general form:

$$map : (k1, v1) -> list(k2, v2)$$

$$reduce : (k2, list(v2)) -> list(k3, v3)$$

The map function gets a key/value pair as an input and produces list of intermediate key-value pairs. The reduce function, accepts a key and a list of values and provides a list of key/value pairs as output.

The types of the aforementioned key-value pairs may be chosen by the programmer and can be different with each other. However, the reduce input must have the same types as the map output.

In order to make more comprehensible how processing can be done with the help of MapReduce, we can take a look at the *WordCound* example, which counts the occurences of each word in a set of documents:

```
1  map(String input_key, String input_value):
2      //input_key: document name
3      //input_value: document contents
4      for each word w in input_value:
5          emit (w, 1);
6
7  reduce(String output_key, Iterator intermediate_values):
8      // output_key: a word
9      // output_values: a list of counts
10     int sum = 0;
11     for each v in intermediate_values:
12         sum += v;
13     emit (word, sum);
```

Here, each document is split into words and each word is counted by the map function, using the word as the result key. The framework groups together all the pairs with the same key and feeds them to the reduce function, which in turn sums up all of its input values to find the total appearances of that word.

The whole dataflow is illustrated in figure 2.1:

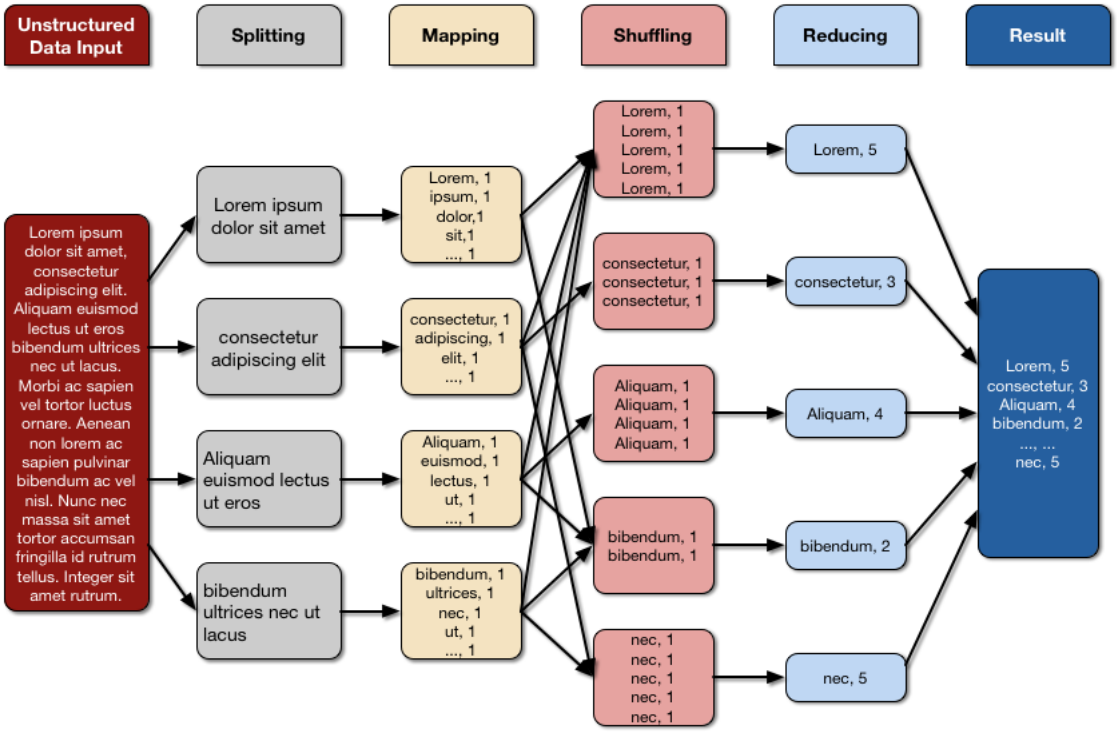


Figure 2.1: MapReduce logical data flow - WordCount Example[1]

In order to run a MapReduce job, the developer simply needs to define the map and the reduce functions and the framework is responsible for the rest. In practice however, having a basic understanding of how the system works, especially the ”shuffle” phase, is crucial should you need to optimize a MapReduce program.

The whole procedure is illustrated in figure 2.2:

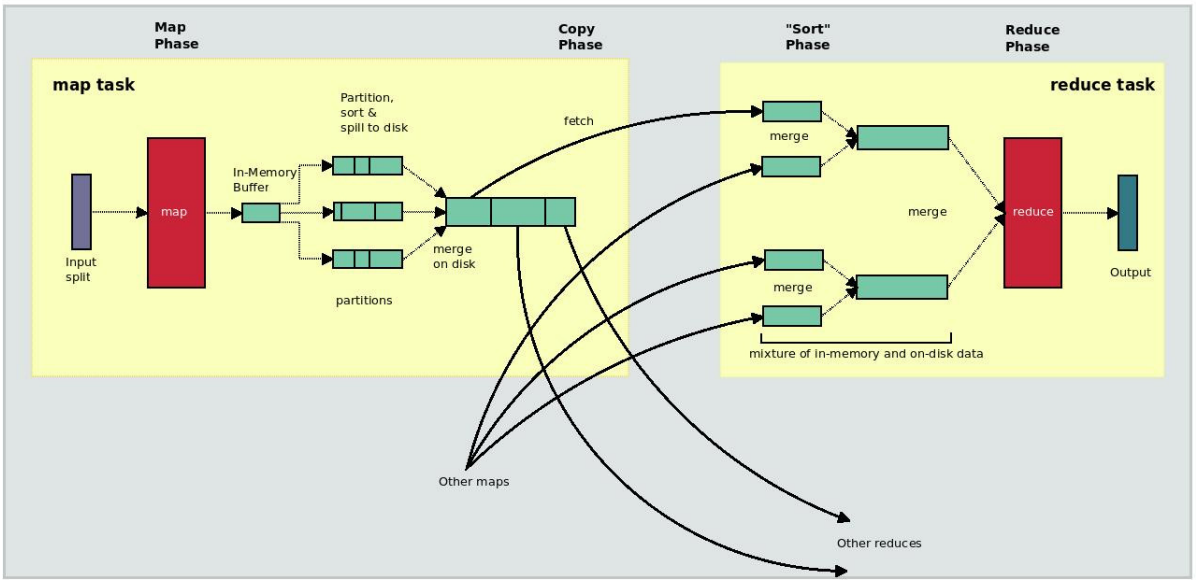


Figure 2.2: Shuffle and Sort in MapReduce

As we can see in the picture, Map task involves the following actions:

Map Processing First, the original input is divided into fixed-sized pieces called *input splits*. Hadoop creates one map task for each split, which runs the user-defined function for each *record* in the split.

Spill The map output is written in a circular memory buffer, associated with each map. When the buffer size reaches a threshold size (default 0.80 or 80%) a background thread starts to spill the contents to disk. While the spill takes place map continues to write data to the buffer, but in case the buffer fills up during this time, the map will block until the split is complete.

Partitioning Depending on whether there is one or multiple reduce tasks *partition* occurs. In specific, before writing to the disk a background thread divides the data into partitions corresponding to the reducers that they will be ultimately sent to. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partioning can be controlled by a user-defined partioning function, but normally the default partitioner, which uses a hash function to bucket the keys, works very well.

Sorting Within each partition, the background thread performs an in-memory sort by key. The sorted output is provided to the combiner function if any, so that less data is written to local disk and transferred to the reducer.

Each time the memory buffer reaches the spill threshold, a new spill file is created, so after the map task has written its last record, there could be several spill files.

Merging Before the map task is finished, the spill files are merged into a single partioned and sorted output file[7].

The Reducer has three phases:

Copy Phase The output file's partitions are transfered (over HTTP) to the place the reduce task or tasks are going to take place. Note, that the reduce task needs the map output for its particular partition from several map tasks across the cluster, that they may finish at different times. So, the reduce task starts copying their outputs as soon as each completes. This is known as the *Copy Phase* of the reduce task.

Map tasks are copied to the reduce tasks JVM's memory if they are small enough, otherwise they are copied to disk. They are merged together and spilled to disk. If a combiner is specified, it will be run during the merge to reduce the amount of data written to disk.

Sort Phase When all map outputs have been copied, the reduce task moves into the *Sort Phase*, which should properly be called the *Merge Phase*, as the sorting was carried out on the map side, which merges the map outputs, maintaining their sorting order.

Reduce Phase When the final round of merges into a single sorted file is done, the merge saves a trip to disk by directly feeding the reduce function in what is the last phase: the *Reduce Phase*. This final merge can come from a mixture of in-memory and on-disk segments.

During the reduce phase, the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output filesystem, typically HDFS. In the case of HDFS, because the node manager is also running a datanode, the first block replica will be written to the local disk. [7]

So, it is clear now that the programs written in this style are automatically parallelized by the MapReduce framework. MapReduce framework does not require any specialized cluster for parallelization, a simple cluster made up of commodity machines, can be used to form a MapReduce cluster.

2.1.2 HDFS

Hadoop is mainly used for processing large datasets. Since those datasets usually outgrow the storage capacity of a single physical machine, it becomes necessary to use a *distributed filesystem*, that manages the storage across a network of machines instead of a single one.

Hadoop comes with a distributed filesystem called HDFS, which stands for *Hadoop Distributed Filesystem*. HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware [7].

HDFS Concepts

Blocks: HDFS, like a filesystem for a single disk, has the concept of the block, but it is much larger - 128 MB by default. Like in a regular filesystem, files in HDFS are broken into block-sized chunks, which are stored as independent units. However, unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage. For example, a 1MB file stored with a block size of 128MB uses 1MB disk space, not 128MB.

This block abstraction of HDFS brings several benefits:

- All disks in the cluster can be utilized, and of course a file can be larger than any single disk in the network.
- Storage system becomes much simpler, able to handle efficiently failure nodes and eliminating metadata concerns since file metadata does not need to be stored with the blocks and can be handled by another system separately.
- Furthermore, blocks are much easier to manage than files, fitting well with replication and providing fault tolerance and availability.

The reason HDFS blocks are large compared to disk blocks is to minimize the cost of seeks. If the block is large enough, the time it takes to transfer the data from the disk can be significantly longer than the time to seek to the start of the block.

Namenodes and Datanodes: An HDFS cluster has two types of nodes operating in a master-worker pattern: a namenode(the master) and a number of datanodes(workers), as we can see in figure: 2.3.

The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located; however it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts.

Datanodes, on the other hand, are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode) and they report back to the namenode periodically with list of blocks they are storing.

Namenode nees to be resilient to failure, since without it the filesystem cannot be used. Without the namenode, all the files, even those residing in different machines, will be lost, since there is no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, there have been developed two mechanisms:

- The files that make up the persistent state of the filesystem metadata are backed up. The usual configuration choice is to write (synchronous and atomic operations) to local disk as well as a remote NFS mount.

- A *secondary namenode* is ran. Despite its name it does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log. In case of total failure of the primary the namenode's metadata files that are on NFS are copied to the secondary and the secondary can be used as the new primary.

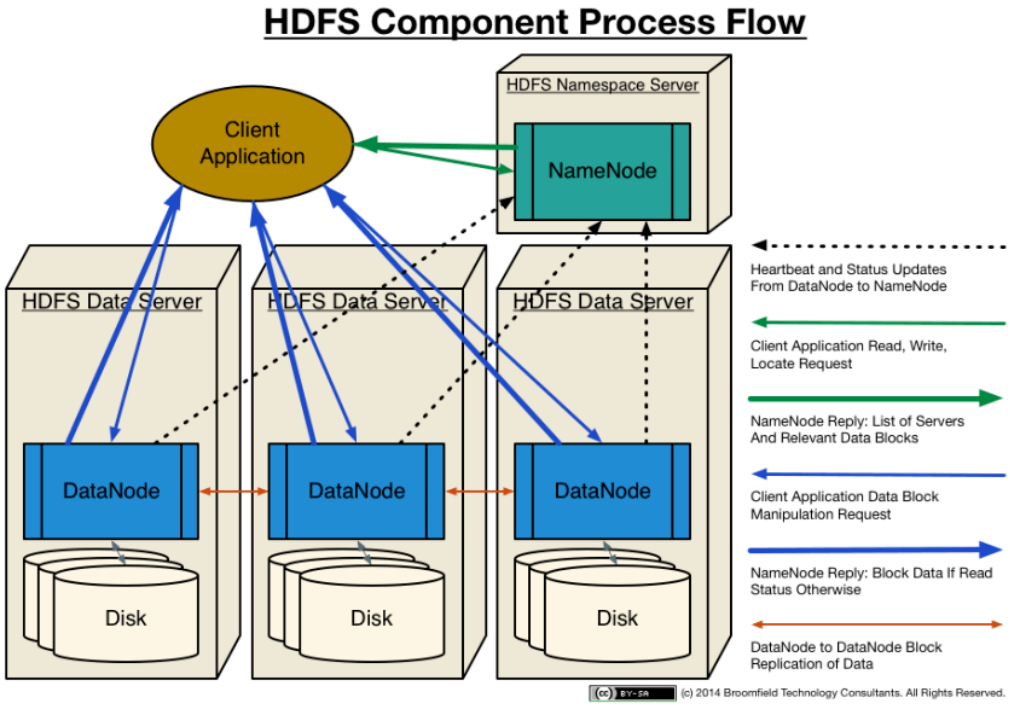


Figure 2.3: HDFS Architecture [1]

Block Caching: Normally a datanode reads blocks from disk, but for frequently accessed files the blocks may be explicitly cached in the datanode's memory, in an off-heap *block cache*. By default a block is cached in only one datanode's memory, although the number is configurable on a per-file basis. Job schedulers can take advantage of cached blocks by running tasks on the datanode where a block is cached, for increased read performance.

HDFS Federation: The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling. HDFS Federation, allows to create a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace.

Under federation, each namenode manages a *namespace volume*, which is made up of metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes. Block pool storage is not partioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

HDFS High Availability: The combination of replicating namenode metadata on multiple filesystems and using the secondary namenode to create checkpoint protects against data loss, but it does not provide high availability of the filesystem. The namenode is still a *single point of failure (SPOF)* and in case of failure, all clients – including MapReduce jobs - would be unable

to read, write or list files. In order to recover from such a failure, HDFS high availability(HA) is added, with a pair of namenodes in an active-standby configuration. If the active namenode fails, the standby can take over very quickly (in a few tens of seconds) because it as the latest state available in memory. The transition from the active namenode to the standby is managed by a new entity in the system called the *failover controller* [7].

When to use HDFS

HDFS may be very valuable but it is not suitable for any kind of applications. HDFS is useful when handling very large files, meaning files that are hundreds of megabytes, gigabytes, when handling very large files, meaning files that are hundreds of megabytes, gigabytes, terabytes in size or even larger. It is not suitable for handling lots of small files. The amount of files in the filesystem is governed by the amount of memory on the namenode since the namenode holds the filesystem metadata in memory.

Furthermore, the write-once, read-many(WORM) pattern is used, which means that it is suitable only in cases that a dataset is generated or copied from the source and then various analyses are performed on that dataset over time. As a result, applications that require low-latency access to data, will not work well with HDFS. HDFS is optimized for delivering a high throughput of data and this may be at the expense of latency.

Last but not least, files in HDFS may be written to by a single writer. Writes are always made at the end of the file, in append-only fashion. Hence, there is no support for multiple writers or for modifications at arbitrary offsets in the file.

2.1.3 Apache Hadoop YARN

Apache Hadoop YARN [2] is the framework used for job scheduling and cluster resource management, as we can see in 2.4.

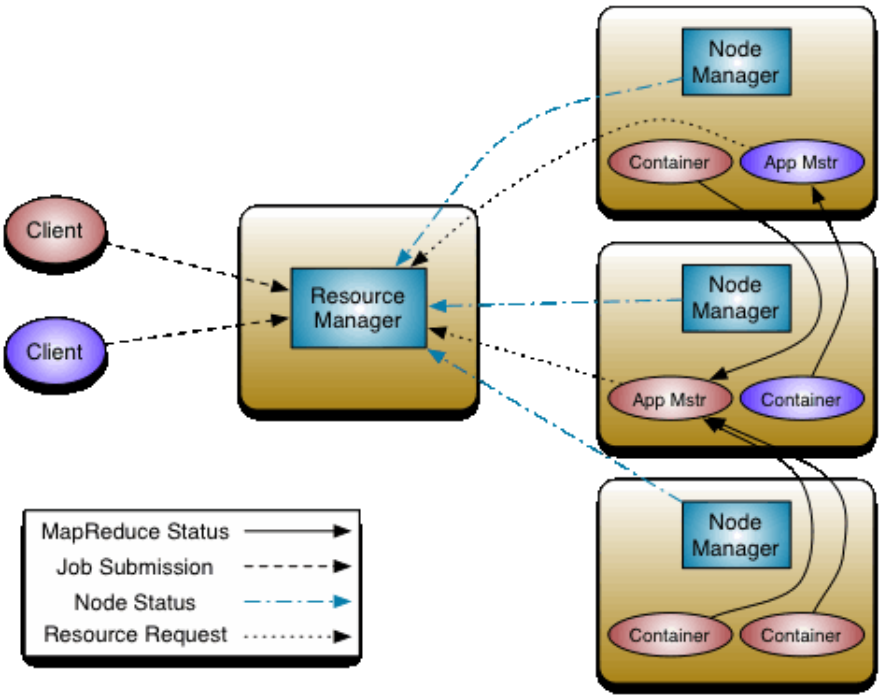


Figure 2.4: Apache Hadoop YARN[2]

The fundamental idea of YARN is to split the functionalities of resource management and job scheduling/monitoring into separate daemons. So we have a global ResourceManager (RM) and per-application ApplicationMaster (AM), where application can be either a single job or a DAG of jobs. In the version of hadoop we used, the JobTracker and the TaskTracker play the role of the ResourceManager and the ApplicationMaster accordingly.

JobTracker JobTracker [18] is the interface between a client application and the Hadoop framework. It is the service within Hadoop that farms out MapReduce tasks to specific nodes in the cluster, ideally the nodes that have the data, or at least are in the same rack. Once code is submitted to the Hadoop cluster, JobTracker is responsible for communicating with the NameNode in order to determine where the data blocks of the input file reside, locating the TaskTracker nodes with available slots at or near the data and submittting the job to the chosen TaskTracker nodes. After the submission of the job to the TaskTracker nodes, those are monitored. In specific, JobTracker expects to see heartbeat signals from each one of them, otherwise it assumes that they have failed and the work is scheduled on a different TaskTracker. TaskTracker from its side, notifies JobTracker when a task fails. In this case, there are three options:

1. the job is resubmitted elsewhere, on a different node if necessary, up to a predefined limit of retries.
2. that specific record is marked as something to avoid
3. or the TaskTracker may be blacklisted as unreliable

After the completion of the job, JobTracker updates its status and client applications can poll the JobTracker for further information.

TaskTracker When the Hadoop cluster is started, along with the DataNode processes, a TaskTracker [19] process is started on each node of the cluster on which data is to be stored. The TaskTracker gets its execution orders (Map,Reduce or Shuffle operations) from the JobTracker. Every TaskTracker is configured with a set of slots, which indicates the number of simultaneous tasks that it can accept. When the JobTracker tries to find somewhere to schedule a task, it first looks for an empty slot on the same server that hosts the DataNode containing the data, and if not, it looks for an empty slot on a machine in the same rack.

The TaskTracker spawns separate JVM processes to do the actual work, so as to ensure that process failure does not take down the TaskTracker. The TaskTracker monitors these spawned processes, by capturing the output and the exit codes. When the process finishes, successfully or not, the TaskTracker is responsible for communicating job execution status back to the JobTracker, along with housekeeping messages such as the number of the avalaible execution slots and periodic heartbeat messages to assure the JobTracker that the TaskTracker is alive and running.

2.2 MapReduce and graph processing

In this thesis our main focus is the analysis of Linked Data. Linked data, have the characteristics that are huge in size and can form huge graphs. So, for the processing of those data, we have to take both those factors into consideration. As already mentioned, the processing of big data, can be done easily and effectively with the help of the MapReduce framework, which provides a simple and powerful model that enables developers to build scalable parallel algorithms capable of running on clusters of commodity machines. However, the inherent graph nature of those data, renders the MapReduce framework suboptimal.

In general, graph processing algorithms are iterative and need to traverse the graph in some way. MapReduce framework though, does not provide direct support for iterative data analysis tasks. Instead, users need to design iterative jobs by manually chaining multiple MapReduce tasks and orchestrating their execution using a driver program. This approach however, is ill suited for graph-processing, since it requires passing the entire state of the graph from one stage to the next, something that except for the need of coordinating the steps of a chained MapReduce job, requires additional communication and the associated serialization overhead.

Even with these limitations, several graph and iterative computing libraries have been built on top of MapReduce due to its ability to run reliably in production environments. Iterative frameworks on MapReduce style computing models is an area that has been explored in Twister [20] and Haloop [21]. However, these approaches remain inefficient for the graph processing case because the efficiency of graph computations depends heavily on inter-processor bandwidth as graph structures are sent over the network after each iteration.

While much data may be unchanged from iteration to iteration, the data must be reloaded and reprocessed at each iteration, resulting in unnecessary wastage of I/O, network bandwidth, and processor resources. In addition, the termination condition might involve the detection of when a fix point is reached. The condition itself might require an extra MapReduce task on each iteration, again increasing the resource usage in terms of scheduling extra tasks, reading extra data from disk, and moving data across the network.

We can see the mapreduce chaining in Figure: 2.5.

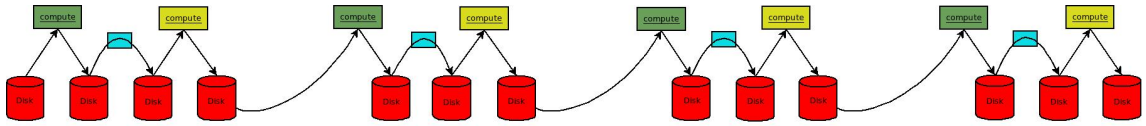


Figure 2.5: MapReduce Chaining

To solve the inherent performance problem of MapReduce, several distributed graph processing systems have been recently introduced. In particular, in 2010, Google has pioneered this area by introducing the Pregel [12] system as a scalable platform for implementing graph algorithms. Pregel relies on a vertex-centric approach, which is inspired by the Bulk Synchronous Model(BSP) [3], where programs are implemented as a sequence of iterations. It aims to batch-oriented processing, the computation is done in-memory, runs on own infrastructure and follows a master-slave architecture.

Unfortunately, the Pregel source code was not made public. Apache Giraph was designed to bring large-scale graph processing to the open source community, based loosely on the Pregel model, while providing the ability to run on existing Hadoop infrastructure.

Apart from Giraph, asynchronous models of graph computing have been proposed in such systems as Signal Collect[22], GraphLab[15] and Grace[23]. While asynchronous graph computing has been demonstrated to converge faster for some applications, it adds considerable complexity to the system and the developer. Most notably, without program repeatability it is difficult to ascertain whether bugs lie in the system infrastructure or the application code. Furthermore, asynchronous message queues for certain vertices may unpredicatably cause machines to run out of memory.

Asynchronous graph processing engines tend to have additional challenges in scaling to larger graphs due to unbounded message queues causing memory overload, vertex-centric locking complexity and overhead, and difficulty in leveraging high network bandwidth due to fine grained computation.

2.3 Giraph

For the scope of this thesis we are going to use Giraph [14]. First we are going to describe the BSP computing model which Giraph implements, then we are going to explain how it works and the reason we chose it chose it among the other available frameworks.

2.3.1 BSP

It is clear that nowadays, with the rapid growth of data and the computations that come along with them, sequential computing is not a viable solution. Therefore, a large number of parallel models have been proposed over the past years. They include: IMD parallelism, synchronous message passing, logic programming, graph reduction, dataflow and various forms of cache-based virtual shared memory. Although each of these approaches has its own benefits, most of them are lacking universal applicability, or make it hard to achieve portability and performance. Those based on message passing are inadequate because of the complexity of correctly creating paired communication actions (send and receive) in large and complex software. Such systems are also prone to deadlock as a result.

Bulk Synchronous Parallel (BSP) though, is quite different from the aforementioned approaches. It provides a discipline for the design of universal, scalable, fully portable programs, which are capable of offering high performance in a predictable way, on any general purpose parallel architecture. It also permits the correctness of parallel programs to be determined in a way which is not much more difficult than sequential programs. The key of its universal applicability across the whole range of parallel architecture, is the fact that it decouples the two fundamental aspects of parallel computation - communication and synchronization.

The BSP programming model

In architectural terms a BSP computer consists of:

1. components capable of processing and/or local memory transactions(i.e processors)
2. a network that routes messages between pairs of such components, and
3. a hardware facility that allows for the synchronization of all or a subset of components.

This is commonly interpreted as a set of processors which may follow different threads of computation, with each processor equipped with fast local memory and interconnected by a communication network. A BSP algorithm relies heavily on the third feature. A computation proceeds in a series of global supersteps, where each superstep is further subdivided into three ordered phases consisting of:

1. Concurrent Computation: every participating processor may perform local computations, i.e., each process can only make use of values stored in the local fast memory of the processor. The computations occur asynchronously of all the others but may overlap with communication.
2. Communication. The processes exchange data between themselves to facilitate remote data storage capabilities.
3. Barrier synchronization. When a process reaches this point (the barrier), it waits untill all other processes have reached the same barrier.

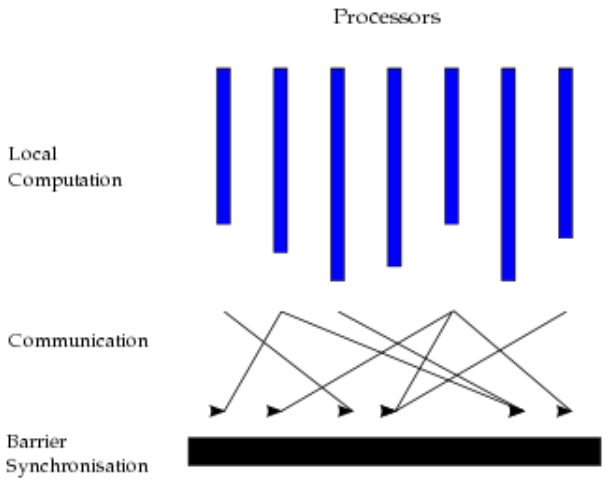


Figure 2.6: Superstep[3]

A superstep is shown in figure 2.6.

The computation and communication actions do not have to be ordered in time. Communication typically takes the form of the one-sided put and get Direct Remote Memory Access (DRMA) calls, rather than paired two-sided send and receive message passing calls. The barrier synchronization concludes the superstep: it ensures that all one-sided communications are properly concluded.

The BSP model is also well-suited to enable automatic memory management for distributed-memory computing through over decomposition of the problem and oversubscription to processors. The computation is divided into more logical processes than there are physical processors, and processes are randomly assigned to processors. This strategy can be shown statistically to lead to almost perfect load balancing, both of work and communication.

Furthermore, since communication and synchronization are decoupled in a BSP program, the programmer does not have to worry about problems such as deadlock, which can occur with synchronous message passing. Debugging a BSP program is also made much easier by this decoupling. The barrier at the end of a superstep provides an appropriate breakpoint at which the global state of the parallel computation is well defined and can be interrogated. Debugging and reasoning about the correctness of a BSP program are, therefore, not much more difficult than for a sequential program.

2.3.2 How it works

Apache Giraph is an iterative graph processing framework. It is a loose implementation of Google's Pregel and utilizes Apache Hadoop's MapReduce implementation to process graphs, as we can seen in figure 2.7.

Intuitively, when designing a Giraph algorithm you should think like a vertex, that knows its local state, its neighbours, can send messages to any other vertex in the graph using the bulk synchronous parallel programming model, can declare when its done and can mutate graph topology.

The input to a giraph computation is a graph composed of vertices and directed edges. Each vertex stores a *value*, so does each edge. The input, thus, not only determines the graph topology, but also the initial values of vertices and edges.

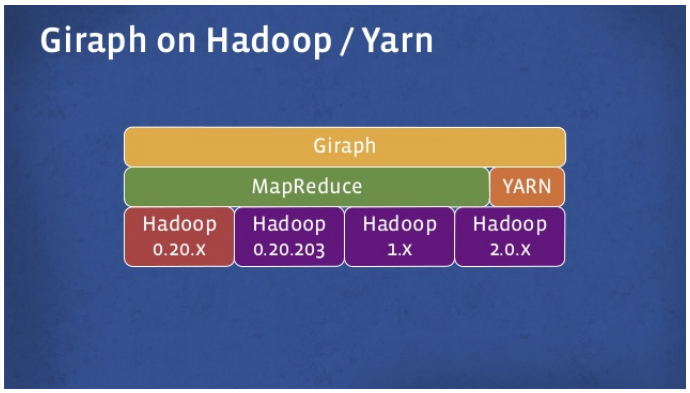


Figure 2.7: Giraph on Hadoop/YARN[4]

In Giraph, graph-processing systems are expressed as a sequence of iterations called supersteps. During a superstep, the framework starts the user-defined compute() function for each vertex, conceptually in parallel. The user-defined functions specifies the behaviour at a single vertex V and a single superstep S.

The Compute method:

- receives and reads the messages sent to the vertex in the previous superstep S-1.
- computes using the messages, and the vertex and outgoing edges values, which may result in modifications to the values, and
- may send messages to other vertices, which are received at superstep S+1.

The compute method does not have direct access to the values of other vertices and their outgoing edges. Inter-vertex communication occurs by sending messages. Messages are typically sent along outgoing edges, but the program can send a message to any vertex with a known identifier. In principle, each superstep represents atomic units of parallel computation.

There is a *barrier* between consequtive supersteps. By this we mean that:

1. the messages sent in any current superstep get delivered to the destination vertices only in the next superstep, and
2. vertices start computing the next superstep after every vertex has completed computing the current superstep.

The graph can be mutated during computation by adding or removing vertices or edges. Values are retained across barriers. That is, the value of any vertex or edge at the beginning of a superstep is equal to the corresponding value at the end of the previous superstep, when graph topology is not mutated.

Giraph applies a master/worker architecture where the master node assigns partitions to workers, coordinates synchronization, requests checkpoints, aggregates aggregator values and collects health statuses. It uses Zookeeper for synchronization. In general, Giraph programs run as Hadoop jobs without the reduce phase. In particular, Giraph leverages the task scheduling component of Hadoop clusters by running workers as special mappers, that communicate with each other to deliver messages between vertices and synchronize in between supersteps.

During program execution, graph vertices are partioned and assigned to workers. The default partioning mechanism is hash-partitioning. However, custom partion can be also applied. Initially, all vertices

are assigned an active status at superstep 1 of the executed program. Each vertex can deactivate itself by voting to halt and turn to the inactive state at any superstep if it does not receive any message. A vertex can return to the active status if it receives a message in the execution of any subsequent superstep. This process continues until all vertices have no messages to send, and become inactive. Hence, program execution ends when at one stage all vertices are inactive. Each vertex outputs some local information, which usually amounts to the final vertex value. Each machine that performs computation, it keeps vertices and edges in memory and uses network transfers only for messages. Therefore, the model is well-suited for distributed implementations because it does not involve any mechanism for detecting the order of execution within a superstep, and all communication is from superstep S to superstep S+1.

The Apache giraph dataflow is illustrated in figure 2.8

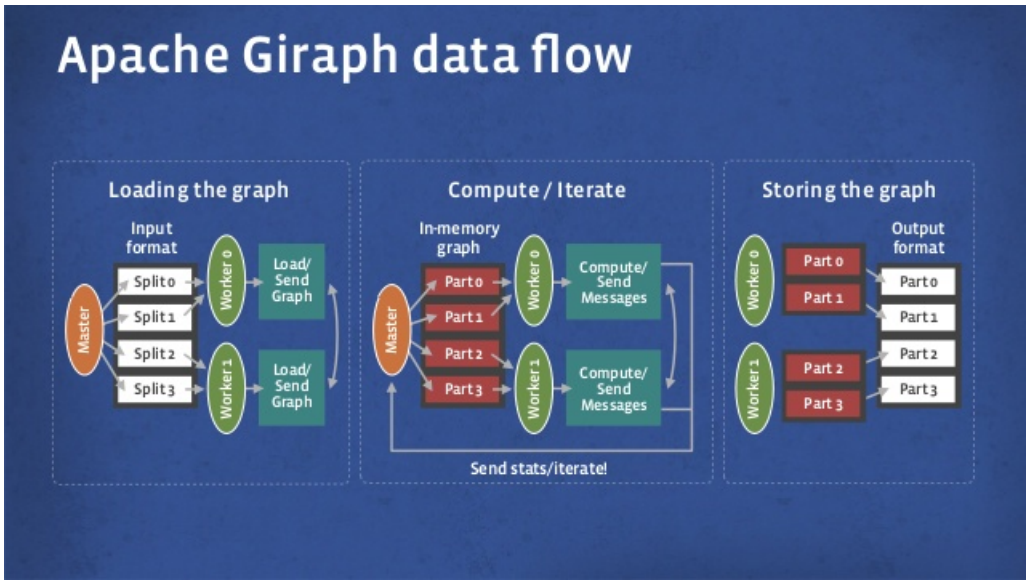


Figure 2.8: Apache Giraph data flow [4]

In 2.9, we can see how the computation of the maximum value is done in Giraph. Dotted lines are messages, and shaded vertices have voted to halt. Initially, all vertices are active, and each one send its value to its neighbour. In the next superstep, each vertex computes the max value, by comparing the values from its incoming messages and its own. In case, its own value is the biggest one, it votes to halt. Otherwise, it replaces its own value with the max found till now, and propagates it to its neighbours. The same process continues, till all vertices have become inactive.

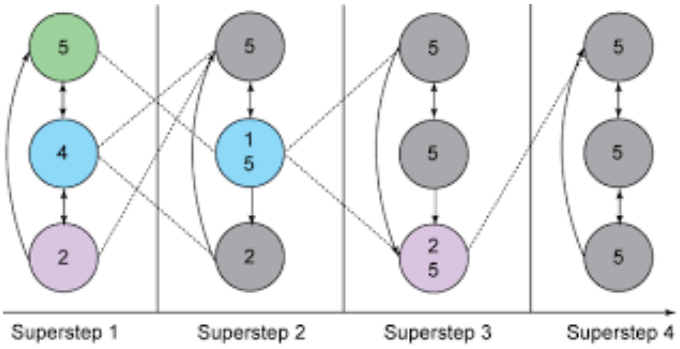


Figure 2.9: Giraph Execution Example - Max Value Computation [5]

2.3.3 Why Giraph

There are many reasons why Giraph is the system of our preference. First of all, unlike Pregel system, it is open-source and well-suited for graph processing. Since computations take place in-memory, it is stateful and only the intermediate values are sent. The disk is hit only during input, output, and checkpointing, unlike when running a MapReduce program. In addittion, it can go out-of-core, when the necessary memory is not available, resulting in slower performance though. Graph mutation is supported as well.

Except from the above mentioned features of Giraph, a great advantage is that it is fully integrated with Hadoop. In fact, it leverages 100 percent of the existing hadoop infrastructure, thus Hadoop clusters can be reused, unlike other similar systems like Hama. Since, it is built on the top of Hadoop, it inherits the benefits that come with its use, like fault-tolerance and scalability. Furthermode the message-based communication results in no locks, while the global synchronization renders the use of semaphores unnecessary. In addition, it is much simpler than the asynchronous models, a very important factor for debugging and troubleshooting in large clusters.

Chapter 3

Distributed processing of queries over Linked Data

As mentioned in Chapter 1, the main purpose of this thesis is to exploit the ability of distributed systems for fast processing of graphs, introduce methods for resolving various types of queries in such systems (e.g., Apache Giraph) over graph data and study the performance of the distributed processing of such queries. In this chapter, we are going to briefly describe what Linked Data is, the best ways to publish it, the model used for its description, the existing serialization formats and the SPARQL query language. Subsequently, we are going to present the datasets we chose for our queries and focus on introducing efficient methodologies of resolving representative types of queries on such datasets, as met in the bibliography. In particular, we are going to present the basic factors we took into account, as well as various performance metrics. Finally, we are going to present the chosen types of queries resolved in a distributed manner using Apache Giraph and the methodology proposed for their resolution.

3.1 Linked Data

3.1.1 The Rationale for Linked Data

We live in a world which constantly produces a huge amount of data. A great, if not the biggest part of this data, is produced through the web. This data is not so useful when we treat it as isolated pieces of information, but it can be of great importance when we find the relationships that exist within. This allows us to extract meaningful information and come to important conclusions, to which we could not reach otherwise.

Linked Data is exactly about that: using the Web to connect related data that was not previously linked or using the Web to lower the barriers to linking data currently linked using other methods. This can be achieved by publishing structured data using standard Web technologies such as Hypertext Transfer Protocol (HTTP), Resource Description Framework (RDF), and uniform resource identifiers(URIs) so that data can be interlinked and become more useful through semantic queries[24].

At this point the following obvious question arises: Why do we need Linked Data and don't try to use or combine one or more of the current mechanisms for sharing and reusing data on the Web? In order to answer this question we have to consider what kind of data are published on the Web, in which form and which are the current mechanisms of retrieving and processing such kind of data.

First of all, it is essential that the data published on the Web is in some way structured. The more regular and well-defined the structure of the data is the more easily people can create tools to reliably process it for reuse [25]. Structured data is made available in the Web today in various forms. Data is published as CVS dumps, excel spreadsheets and in a multitude of domain-specific data formats. It is also embedded into HTML pages using Microformats, or accessible via Web APIs made by various data providers. So, it is clear, that already exist well-established publishing technologies, which can make data available

in the Web, in a structured form, most of the times sufficient to reveal also information, like if there are correlations within data and of what kind.

In specific, most Web sites have inherently some degree of structure, due to the language in which they are created, HTML. However, HTML is oriented towards structuring textual documents rather than data. As data is intermingled into the surrounding text, it is hard for software applications to extract snippets of structured data from HTML pages [25]. This issue, can be solved to some extent, with the use of a variety of microformats that have been invented and can be used to publish structured data, describing specific types of entities, such as people, organizations, events, reviews, ratings and other, through embedding of data in HTML pages. This approach however, has the drawback that microformats are restricted to representing data about a small set of different types of entities and various kinds or relationships cannot be expressed through them. Therefore, microformats are not suitable for sharing arbitrary data on the Web.

The use of Web APIs on the other hand, can provide a more generic approach to making structured data available on the Web. Web APIs provide simple query access to structured data over the HTTP protocol. The weak point of this approach though is that there are needed specialized applications that can combine data from several resources, each of which is accessed through an API specific to the data provider. This requires significant effort from the programmer, since he should be able to understand all the methods available to retrieve data from each API and write custom code for accessing data from each data source.

Furthermore, many Web APIs refer to items of interest using identifiers that have only local scope and they are meaningless outside of the context of that specific API. In such cases, there is no standard mechanism to refer to items described by one API in data returned by another. In addition, although the format the structured data is provided, such as XML and JSON, have extensive support in a wide range of programming languages, they have their own limitations. For example, data cannot be the 'followed' in the way links are followed in HTML. They exist as isolated fragments, lacking onward links signposting the way to related data. Hence, browsers and search engine crawlers cannot exploit such kind of information, traverse one or more links and retrieve valuable information from the referenced data [25].

Consequently, although there are various approaches of discovering, retrieving and processing data available in the Web, they are inadequate since either they cannot describe all kind of relationships between the data, either require huge effort by the programmer in order to integrate data from large number of formerly unknown data sources, or they cannot provide the right linkage between the data and thus making them discoverable. Linked Data on the other hand, provide mechanisms to overcome the aforementioned issues. Before, describing those mechanisms though, it is preferrable to explain how Linked Data 'work'.

3.1.2 Principles of Linked Data

The term *Linked Data* refers to a set of best practices for publishing and interlinking structured data on the Web. These best practices were introduced by Tim Berners-Lee, director of the World Wide Web Consortium (W3C), in his Web architecture note 'Linked Data'[26] in 2006, paraphrased along the following lines:

1. Use URIs as names for things.
2. Use HTTP URIs so that those names can be looked up (interpreted, 'dereferenced')
3. Provide useful information about what a name identifies when it's looked up, using open standards such as RDF, SPARQL, etc.

4. Refer to other things using their HTTP URI-based names when publishing data on the Web, so more things can be discovered.

A specific category of Linked Data is *Linked Open Data*, content of which is open. Large Linked Open Data sets include DBpedia and Freebase.

Those practices have become known as the *Linked Data principles* or rules, but, in essence, they are expectations of behavior. Breaking them does not destroy anything, but misses an opportunity to make data interconnected [26].

So, let's describe them in a bit more detail: The first Linked Data principle advocates using URI references to identify things, where 'things' does not refer only to Web documents and digital content. It can be also real world objects, such as people, places, cars, or abstract concepts, such as the relationship type of knowing somebody. So, it is clear, that it allows to express any kind of object or relationship, something not possible when using microformats.

The second principle, basically, provides us with a way to discover and access data. With the use of the well-known HTTP protocol, objects and abstract contents can be dereferenced and thus it is possible to extract additional valuable information.

The third rule refers to the use of a single data model for publishing structured data on the Web. This is essential, since in order to enable a wide range of applications to process Web content, we have to agree on standardized content formats. The Resource Description Framework (RDF), a simple graph-based data model that has been designed for use in the context of the Web, is appropriate for this role.

The fourth rule describes a way to connect things, via the use of hyperlinks. It is important that those hyperlinks are not only between Web documents, but they can indicate relationships between any type of thing, such as a person and a company [25].

It should be clear now why Linked Data is preferable instead of the use of the other publishing technologies: it provides a more generic, more flexile publishing paradigm which makes it easier for data consumers to discover and integrate data from large number of data sources.

In particular, Linked Data provides:

A unifying data model It relies on RDF, which has been especially designed for the use case of global data sharing. In contrast, other methods for publishing the data on the Web rely on a wide variety of different data models, making the integration process too complex.

A standardized data access mechanism Linked Data makes data accessible through the use of the HTTP protocol. Hence, data sources can be accessed using generic data browsers, and the complete data space can be crawled by search engines, something not possible with the use of Web APIs, which can be accessed only different proprietary interfaces.

Hyperlink-based data discovery Hyperlinks enable Linked Data applications to discover new data sources at run-time, by connecting entities in different data sources into a single global data space, instead of having data in proprietary formats which remain isolated in data islands.

Self-descriptive data Linked data eases the intergration of data from different sources by relying on shared vocabularies, making the definitions of these vocabularies retrievable, and by allowing terms from different vocabularies to be connected to each other by vocabulary links.

Compared to the other methods of publishing data on the Web, these properties of the Linked Data architecture make it easier for data consumers to discover, access and integrate data. However, it is

important to remember that the various publication methods represent a continuum of benefit, from making data available on the Web in any form, to publishing Linked Data according to the principles described here [25].

Figure 3.1 illustrates the April 2014 scale of the *Linked Data Cloud* originating form the Linking Open Data project and classifies the data sets by topical domain, highlighting the diversity of data sets present in the Web of Data. The graphic shown in this figure is available online at http://lod-cloud.net. Updated versions of the graphic are published on this website in regular intervals.

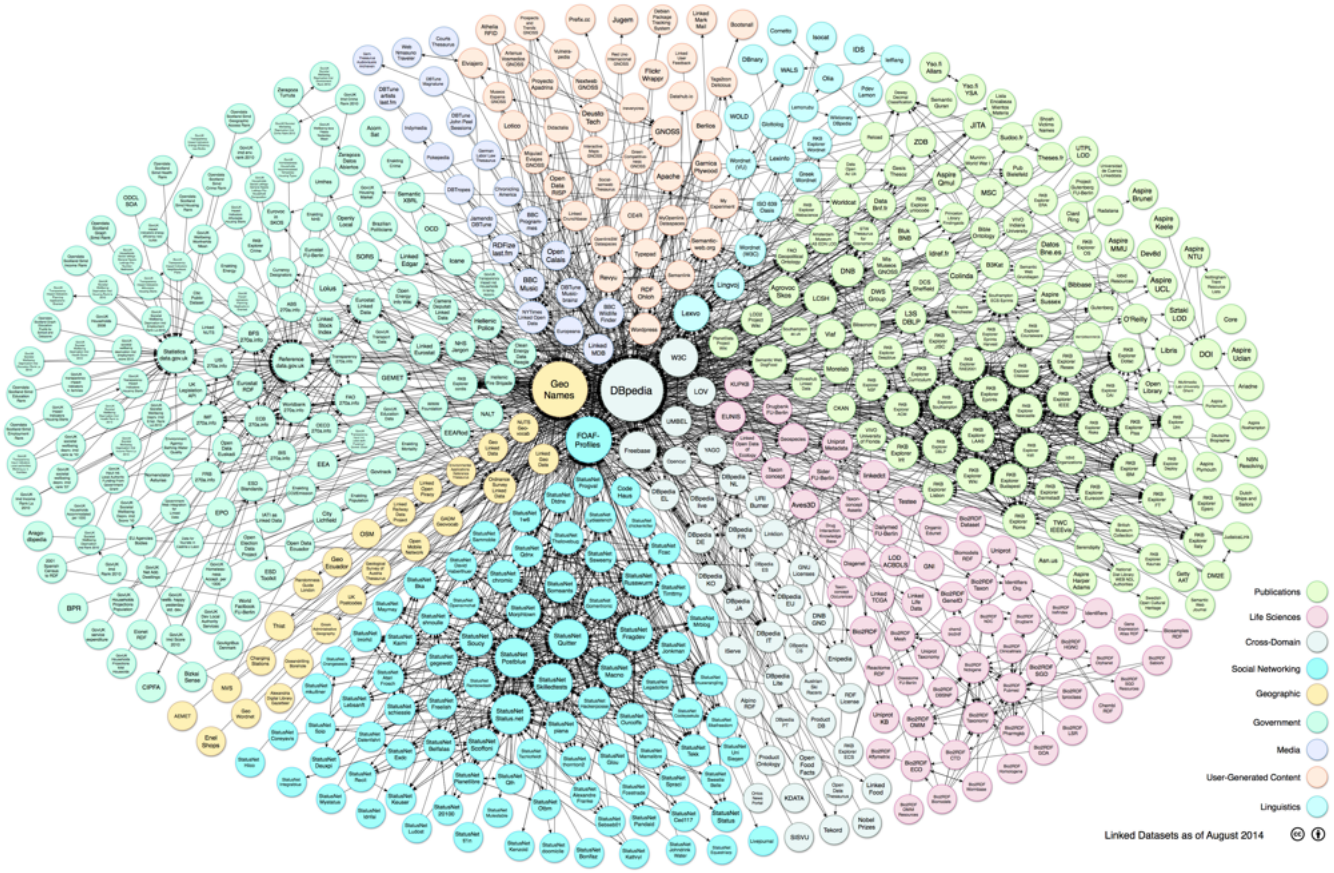


Figure 3.1: State of the LOID Cloud 2014 [6]

3.1.3 RDF

As mentioned above, linking data distributed across the Web requires a standard mechanism for specifying the existence and meaning of connections between items described in this data. This mechanism is provided by the Resource Description Mechanism (RDF). RDF is one of the basic building blocks for forming the web of the semantic data. It is powerful because of its simplicity, ability to express complex schemas and relationships and suitability for modeling all external data frameworks for unstructured, semi-structured and structured data.

In comparison with HTML documents and conventional Web APIs, the key features of RDF worth noting in this context are the following:

RDF links things, not just documents: therefore, RDF links would not simply connect the data fragments from each API, but assert connections between the entities described in the data fragments.

RDF links are typed: HTML links typically indicate that two documents are related in some way, but mostly leave the user to infer the nature of the relationship. In contrast, RDF enables the data publisher to state explicitely the nature of the connection, making data significantly more discoverable and usable [25].

RDF provides a data model that is extremely simple on the one hand but strictly tailored towards Web architecture on the other hand. To be published on the Web, RDF data can be serialized in different formats. The two RDF serialization formats most commonly used to publish Linked Data on the Web are RDF/XML and RDFa.

In essence, RDF defines a type of a graph database. This means, that resourses are related to other resources, with no single resource having any particular intrinsic importance over another (arbitrary object relations), as happens for most other types of data storage (where the concept of hierarchy dominates and some elements are having more precedence or importance over other elements).

In RDF, a description of a resource is represented as a number of triples. The three parts of each triple are called its *subject, predicate* and *object*. A triple mirrors the basic structure of a simple sentence, such as this one: 'Katerina Markaki has nick name Kate', where Katerina Markaki is the subject, has nick name is the predicate, and Kate is the object.

The subject of a triple is the URI identifying the described resource. The object can either be a simple *literal value*, like a string, number or a data; or the URI of another resource that is somehow related to the subject. The predicate, in the middle, indicates what kind of relation exists between subject and object, e.g., this is the name or date of birth (in the case of literal), or the employer or someone the person knows (in the case of another resource). The predicate is also identified by a URI. These predicate URIs come from vocabularies, collections of URIs that can be used to represent information about a certain domain.

Two principal types of RDF triples can be distinguished. *Literal Triples* and *RDF Links*:

1. **Literal Triples** have an RDF literal such as a string, number, or data as the object. Literal triples are used to describe the properties of resources. For instance, literal triples are used to describe the name or date of birth of a person. Literals may be plain or typed: A plain literal is a string combines with an optional language tag. The language tag identifies a natural language, such as English or German. A typed literal is a string combined with a datatype URI. The datatype URI identifies the datatype of the literal. Datatype URIs for common datatypes such as integers, floating point numbers and dates are defines by the XML Scheme datatypes specification.

2. **RDF Links** describe the relationship between two resources. RDF links consist of three URI references. The URIs in the subject and the object position of the link identify the related resources. The URI in the predicate position defines the type of relationship between the resources. Note that a useful distinction that can be made between *internal* and *external* RDF Links. Internal RDF Links connect resources within a single Linked Data source. Thus, the subject and object URIs are in the same namespace. External RDF links connect resources that are served by different Linked Data sources. The subject and the object URIs of external RDF links are in different namespaces. External RDF links are crucial for the Web of Data as they are the glue that connects data islands into a global, interconnected data space.

One way to thing of a set of RDF triples is as an RDF graph. The URIs occuring as subject and object are the nodes in the graph, and each triple is a directed arc that connects the subject and the object. As Linked Data URIs are globally unique and can be dereferenced into sets of RDF triples, it is possible to imagine all Linked Data as one global graph, as proposed by Tim Berners-Lee. Linked data applications operate on top of this giant graph and retrieve parts of it by dereferencing URIs as required [25].

RDF Serialization formats

It is important to remember that RDF is not a data format, but a data model for describing resources in the form of subject, predicate, object triples. In order to publish an RDF graph on the Web, it must first be serialized using an RDF syntax. This simply means taking the triples that make up an RDF graph, and using a particular syntax to write these out to a file (either in advance for a static data set or on demand if the data set is more dynamic). Two RDF serialization formats - RDF/XML and RDFa - have been standardized by the W3C. In addition several other non-standard serialization formats are useful to fulfill specific needs. The relative advantages and disadvantages of the different serialization formats are dsicussed below, along with a code sample showing a simple graph expressed in each serialization.

RDF/XML The RDF/XML syntax is standardized by the W3C and is widely used to publish Linked Data on the Web. A weakness of this syntax though, is that its not human friendly for read and and writing. The RDF/XML syntax is descripted in detail as part of the W3C RDF Primer [27].

RDFa RDFa is a serialization format that embeds RDF tripples in HTML documents. The RDF data is not embedded in comments within the HTML document, as was the case with some early attempts to mix RDF and HTML, but is interwoven within the HTML Document Object Model (DOM). This means that existing content within the page can be marked up with RDFa by modifying HTML code, thereby exposing structured data to the WEb. A detailed introduction into RDFa is given in the W3C RDFa Primer [28].

Turtle Turtle is a plain text format for serializing RDF data. Due to its support for namespace prefixes and various other shorthands, Turtle is typically the serialization format of choice for reading RDF triples or writing them by hand. A detailed introduction to Turtle is given in the W3C Team Submission document Turtle - Terse RDF Triple Language [29]. The MIME type for Turtle is text/turtle; charset=utf-8.

An example of its usage can be seen below:

```
1
2  1 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3  2 @prefix foaf: <http://xmlns.com/foaf/0.1> .
4  3
5  4 <http://biglynx.co.uk/people/dave-smith>
6  5 rdf:type foaf:Person ;
7  6 foaf:name 'Dave Smith' .
```

N-Triples N-Triples is a subset of Turtle, minus features such as namespace prefixes and shorthands. The result is a serialization format with lots of redundancy, as all URIs must be specified in full in each triple. Consequently, N-Triples files can be rather large relative to Turtle and even RDF/XML. However, this redundancy is also the primary advantage of N-Triples over other serialization formats, as it enables N-Triples files to be parsed one line at a time, making it ideal for loading large data files that will not fit into main memory. The redundancy also makes N-Triples amenable to compression, thereby reducing network traffic when exchanging files. There two factors make N-Triples the *de facto* standard for exchanging large dumps of Linked Data, e.g., for backup or mirroring purposes. The complete definition of the N-Triples syntax is given as part of the W3C Test Cases recommendation[30].

A corresponding example is shown below:

```
1  1 <http://biglynx.co.uk/people/dave-smith> <http://www.w3.org
     /1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
2  2 <http://biglynx.co.uk/people/dave-smith> <http://xmlns.com/foaf/0.1/
     name> 'Dave Smith' .
```

RDF/JSON RDF/JSON refers to efforts to provide a JSON(Javascript Object Notation) serialization for RDF. Availability of JSON serialization of RDF is highly desirable, as a grwoing number of programming languages provide native JSON support, including staples of Web programming such as Javascript and PHP. Publishing RDF data in JSON therefore makes it accessible to Web developers without the need to install additional software libraries for parsing and manipulating RDF data. It is likely that further efforts will be made in the near futur to standardize a JSON serialization of RDF [25].

3.1.4 SPARQL

Since the amount of Linked Data increases significantly year by year, the use of query language in order to retrieve the desired information is essential. SPARQL [31], which stands for **SPARQL Protocol and RDF Query Language**, is the basic query language for Linked Data. It is a semantic query language, able to retrieve and manipulate data stored in RDF format. It bears close resemblance to SQL, but it is applicable to an RDF data graph. It was made a standard by the RDF Data Access Working Group (DAWG) of the World Wide Web Consortium, and is recognized as one of the key technologies of the semantic web.

SPARQL allows for a query to consist of triple patterns, conjuctions, disjunctions, and optional patterns. Implementations for multiple programming languages exist. There exist tools that allow one to connect and semi-automatically construct a SPARQL query for a SPAQRL endpoint, for example ViziQuer. In addittion, there exist tools that translate SPARQL queries to other query languages, for example to SQL, and to XQuery.

It provides a full set of analytic query operations such as JOIN, SORT, AGGREGATE for data whose schema is intrinsically part of the data rather than requiring a separate schema definition. Scheme information(the ontology) is often provided externally, though, to allow different datasets to be joined in an unambiguous manner. In addition, SPARQL provides specific graph traversal syntax that can be though of as a graph and figure. It also allows users to write queries against what can loosely be called ‘key-value’ data or, more specifically, data that follows the RDF specification of the W3C. The entire database is thus a set of ‘subject-predicate-oject’ triples. This is analogous to some NoSQL databases, such as MongoDB.

The example below demonstrates a simple query that leverages the ontology definition ‘foaf’, often calles the ‘friend-of-a-friend’ ontology. Specifically, the following query returns names and emails of every person in the dataset:

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?name ?email
3 WHERE {
4   ?person a foaf:Person.
5   ?person foaf:name ?name.
6   ?person foaf:mbox ?email.
7 }
```

This query joins together all of the triples with a matching subject, where the type predicate, ‘a’, is a person (foaf:Person) and the person has one or more names(foaf:name) and mailboxes(foaf:mbox). The chosen variable name here is ‘person’ for readable clarity, however it could be anything else, as long as it is the same on each line of the query to signify that the query engine is to join triples with the same subject. The result of the join is a set of rows - ?person, ?name, ?email. This query returns the ?name and ?email because ?person is often a complex URI rather than a human-friendly string. Note that some of the ?people may have multiple mailboxes, so in the returned set, a ?name row may appear multiple times, once for each mailbox. This query can be distributed to multiple SPARQL endpoints

(services that accept SPARQL queries and return results), computed and results gathered, a procedure known as federated query. Whether in a federated manner or locally, additional triple definitions in the query could allow joins to different subject types, such as automobiles, to allow simple queries, for example, to return a list of names and emails for people who drive automobiles with a high fuel velocity [31].

3.1.5 DBpedia

A typical case of a large Linked Dataset is DBpedia [32], whose aim is to extract structured content from the information created as part of the Wikipedia project, and makes this content available on the World Wide Web. It is one of the most famous parts of the decentralized Linked Data effort, as it includes not only Wikipedia data, but also incorporates links to other datasets on the Web (e.g. Geonames).

The resulting DBpedia knowledge base currently describes over 2.6 million entities. For each of these entities, DBpedia defines a globally unique identifier that can be dereferenced over the Web into a rich RDF description of the enity, including human-readable definitions in 30 languages, relationships to other resources, classifications in four concept hierarchies, various facts as well as data-level links to other Web data sources describing the entity. Over the last year, an increasing number of data publishers have begun to set data-level links to DBpedia resources, making DBPedia a central interlinking hub for the emerging Web of data. Currently, the Web of interlinked data sources around DBpeida provides approximately 4.7 billion pieces of information and covers domains such as geographic information, people, companies, films, music, genes, drugs, books, and scientific publications.

The idea behind this project lies in the fact that Wikipedia is one of the central knowledge knowledge sources of mankind. It is maintained by thousands of contributors, whereas most of the other knowledge bases cover only specific domains, are created relatively by small groups of knowledge engineers and are very const itensice to keep up-to-date as domains change. Hence, The DBpedia project leverages this gigantic source of knowledge and the result is to have a knowledge base which covers many domains, represents community agreement, it is truly multilingual, it is accessible on the Web and automatically evolves as Wikipedia changes. Advantages missing from most of the existing knowledge bases [33].

Access to the DBpedia knowledge base can be provided through the following access mechanisms:

Linked Data. DBpedia resource identifiers are set up to return (a) RDF descriptions when accessed by Semantic Web agents (such as data browsers or crawlers of Semantic Web search engines), and (b) a simple HTML view of the same information to traditional Web browsers.

SPARQL endpoint. Client applications can send queries over the SPARQL protocol to the SPARQL endpoint at http://dbpedia.org/sparql. In addition to the standard SPARQL, the endpoint supports several other extensions of the query language that have been proved to be useful for developing client applications, such as full text search and aggregate functions, notably COUNT(). The endpoint is hosted using Virtuoso Universal Server.

RDF Dumps. Parts of the DBpedia knowledge base in N-Triple serialisation format are available for download on the DBPedia website.

Lookup Index. DBpedia offers a lookup service at http://lookup.dbpedia.org/api/search.asmx, which proposes DBpedia URIs for a given label, thus making it easier for Linked Data publishers to find DBpedia resource URIs to link to.

In order to enable DBpedia users to discover further information, the DBpedia knowledge base is interlinked with various other data sources on the Web according to the Linked Data principles. The knowledge base currently contains 4.9 million outgoing RDF links that point at complementary information about DBpedia entities, as well as meta-information about media items depicting an entity. Over the last year, an increasing number of data publishers have started to set RDF links to DBpedia entities. These incoming links, together with the outgoing links published by the DBpedia project, make DBpedia one of the central interlinking hubs of the emerging Web of Data. These RDF links lay the foundation for:

Web of Data Browsing and Crawling. RDF links enable information consumers to navigate from data within one data source to related data within other data sources using a Linked Data browser. RDF links are also followed by the crawlers of Semantic Web search engines, which provide search and query capabilities over crawled data.

Web Data Fusion and Mashups. As RDF links connect data about an entity within different data sources, they can be used as a basis for fusing data from these sources in order to generate integrated views over multiple sources.

Web Content Annotation DBpedia entity URIs are also used to annotate classic Web like blog posts or news with topical subjects as well as references to places, companies and people. As the number of sites that use DBpedia URIs for annotation increases, the DBpedia knowlesdge base could develop into a valuable resource for discovering classic Web content that is related to an entity.

Concluding, we showed that a rich corpus of diverse knowledge can be obtained from the large scale collaboration of end-users, who are not even aware that they contribute to a structured knowledge base. The resulting DBpedia knowledge base constantly evolves when Wikipedia content is changed, covers a wide range of different domains and connects entities across these domains. More and more valuable information can be extracting by allowing complex queries against the Wikipedia content, making it an exciting test-bed to develop, compare and evaluate data integration, reasoning and uncertainty management techniques, and to deploy operation Semantic Web applications [33].

With this way we are going to use it also in the context of this thesis, since it allows us to use Giraph to do queries over graph data with rich content and include various domains and kinds of relationships between the entities. In addition, since Wikipedia is constantly evolving we can do our experiments, by using each time datasets of different size.

3.2 Selection of queries to study

In order to evaluate the performance of our framework for distributed processing over existing centralized approaches, we chose to split the processing phases for a set of representative types of queries with different characteristics according to the following criteria. The goal is to implement popular types of queries in a distributed manner and define the profile of queries that benefit at most from distributed processing.

1. **Input Size**. This is measured as the proportion of the class instances involved in the query to the total class instances in a given dataset. Here we refer to not just the class instances explicitly expressed but also those that are entailed by the knowledge base.
2. **Selectivity**. This is measured as the estimated proportion of the instances involved in the query that satisfy the query citeria. Of course, whether the selectivity is high or low for a query may depend on the dataset used.

3. **Complexity**. We use the number of the classes and properties that are involved in the query as an indication of complexity. Since, we do not assume any specific implementation of the repository, the real degree of complexity may vary by systems and schemas. For example, in a relational database, depending on the schema design, the number of classes and properties may or may not directly indicate the number of table joins, which are significant operations.

So, we have chosen test queries that cover a range of properties in terms of the above criteria. At the same time, we have emphasized queries with large input and high selectivity. Some subtler factors have also been considered in designing the queries such as the way classes and properties chain together in the query.

3.3 Performance Metrics & Chosen Queries

We consider a set of performance metrics including:

- load time
- repository size
- query response time
- query completeness and soundness.

Among these metrics: the first three are standard database benchmark metrics - query response time and load time and repository size have been commonly used in other database benchmarks; query completeness and soundness are new metrics introduced for our experiments. These metrics are addressed below:

1. Load Time
2. Repository Size
3. Query Responses Time
4. Query Completeness and Soundness

We have chosen to implement different queries in terms of the aforementioned criteria. Here we are going to present the different type of queries, focusing more on their complexity and the way classes and properties chain together. Those can be better described through appropriate figures, where each query is visualized as a graph. The vertices of the graph represent the subjects or the objects involved in the query, while the edges of the graph represent the involved predicates. The other two factors, input size and selectivity, depend on the datasets used and the specific class instances involved in the query, so they may vary, even for queries with the same graphical representation. For the design of our queries, as well as the various factors and performance metrics we used, have been chosen with the help of the Lehigh University Benchmark(LUBM) [34].

Those visualizations can be seen in Figures 3.2 - 3.8. In those figures, wherever we use symbols like Y, Z, Y1, etc, we assume that it can be any VertexID, which is connected with its neighbours in the way shown in the figure. On the other hand, symbols of the form ‘Object1’, ‘Predicate1’ are considered to be the exact values of the VertexID and the Edge accordingly.

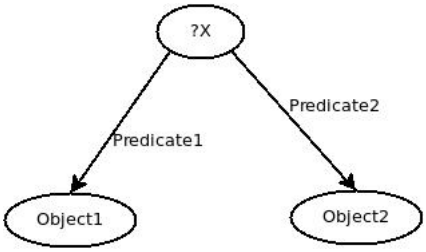


Figure 3.2: Query1

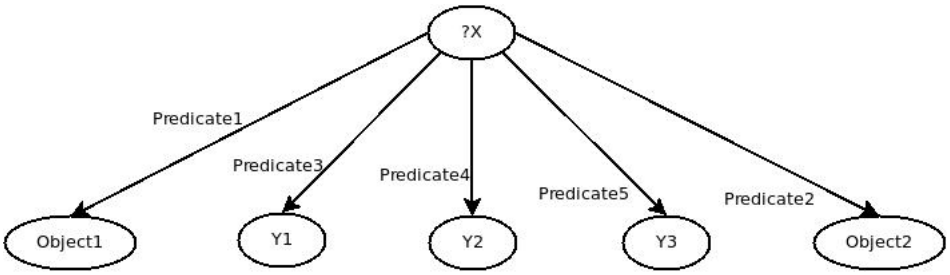


Figure 3.3: Query2

Query 1 (Figure 3.2) is quite simple, asking about any subject which has 'Object1' and 'Object2' as objects, with 'Predicate1' and 'Predicate2' being the values of the outgoing edges that connect the subject with the aforementioned objects accordingly.

Query 2 (Figure 3.3) queries about multiple properties of the same subject. It can be considered as an extension of Query 1, since it is the same, with the difference that the desired subject-answer X, should have three more outgoing edges with specific values, connected to any object. It is obvious that the since the number of classes and properties involved in the query is much bigger, the complexity increases. Note that, although from a first look, it seems like the classes involved in this query are just three more than in Query 1, it is not exactly like that. Since each one of those three additional outgoing edges may lead to a huge number of different objects, the actual number of classes involved in the end, might be quite large.

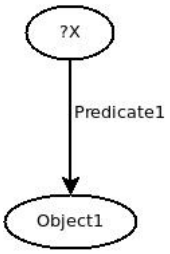


Figure 3.4: Query3

Query 3 (Figure 3.4) is the query with the lowest complexity. This query searches for any subject that is connected to a specific object 'Object1', with the edge that connects them having the value 'Predicate1'.

Query 4 (Figure 3.5) is another query with increased number of classes and properties, thereby complexity and probably selectivity are high. In specific, the subjects-answer should have at least two outgoing edges. One with value 'Predicate1' and another one with value 'Predicate4', pointing to the objects 'Object1' and Y accordingly. Y here does not refer to a specific object, but to any vertice, which has an outgoing edge with value 'Predicate2' connecting it with the vertice 'Object2', and an incoming edge with value 'Predicate3' from a specific subject 'Object3'. Once again, although three

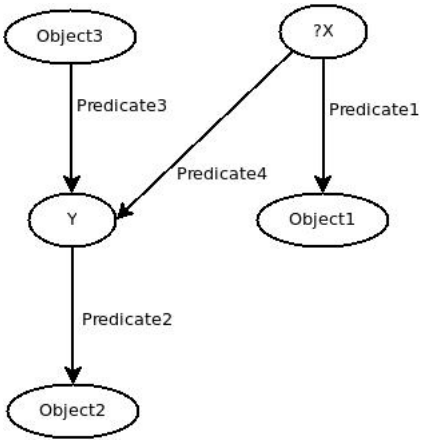


Figure 3.5: Query4

of the vertices involved in the query are specific, one of them (Y) is not, meaning that there may be many vertices Y with the desired characteristics, thus the actual number of vertices involved in the query may vary a lot depending on the dataset.

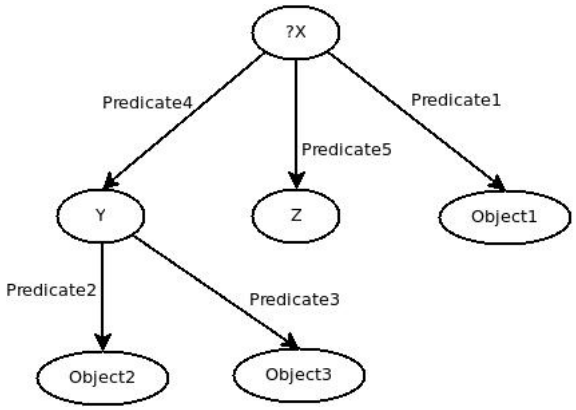


Figure 3.6: Query5

Query 5 (Figure 3.6) is further more complex than Query 4, including one more property and class. The desired answer should be connected two three objects, one specific: 'Object1', with edge of value 'Predicate1', and two others Y and Z, which can be any objects with some specific characteristics. In specific, the vertice Y, must be connected to two other vertices: 'Object2' and 'Object3' with edges having the values: 'Predicate2' and 'Predicate3' correspondingly. Z can be any object as long as it is connected to the X-answer with an edge of value 'Predicate5'.

Query 6 (Figure 3.7) is almost the same as the Query5, but less complicated, since it involves one less property and class.

Query 7 (Figure 3.8) is one of the simplest types of queries to study. It asks for a vertex Y, which can has an incoming edge with value 'Predicate1' from a vertex 'Object1', and an outcoming edge with value 'Predicate2' towards a vertex 'Object2'.

Note that none of the aforementioned queries assumes any hierarchy information or inference. In addition, it is obvious that the labels used in the figures that describe the queries, are just examples for readable clarity, rather than real ones. In our experiments, those labels (e.g. Object1, Predicate1) take actual values, different each time, so as to end up having a set of 15 different queries of different complexity, input, and selectivity. We execute those queries both in a centralized system, Openvirtuoso and a distributed one, Giraph, using datasets of different size, so as to study how the execution time

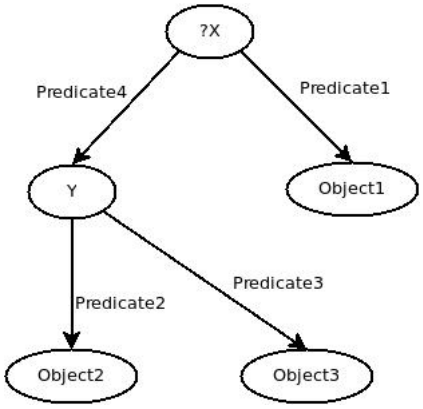


Figure 3.7: Query6

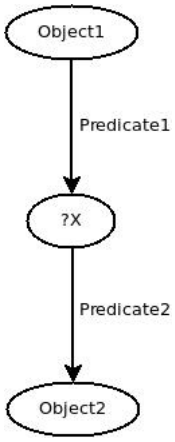


Figure 3.8: Query7

varies with queries of different characteristics.

3.4 Distributed execution of graph queries in Apache Giraph

3.4.1 Our System

As mentioned earlier, the goal is to of our system is to resolve the aforementioned queries, in a distributed manner harvesting the power of multiple commodity processing nodes. The input is going to be the RDF datasets, while the output, is going to be answer to our queries, as shown in Figure 3.9

For the processing of the aforementioned types of sparql queries, we chose to use Giraph, which allows us to find the answer in a distributed but simple way. We can focus on the algorithm we are designing, without the need to worry about the graph representation in memory, the way our algorithm is executed in parallel across the distributed system and how fault tolerance is achieved. Our algorithm is agnostic to the way data is shared across the processing units and the way code is executed concurrently. Thus, thre is no need for locking or concurrent coordination by our side.

Figure 3.10 shows the conceptual stacked organization.

Our implementation consists of the two layers on the top of the figure. In specific:

- **Query Parser**, which basically is a script calling the user to select which type of query he wants to run, provide the necessary arguments, depending on the query he chose, as well as how

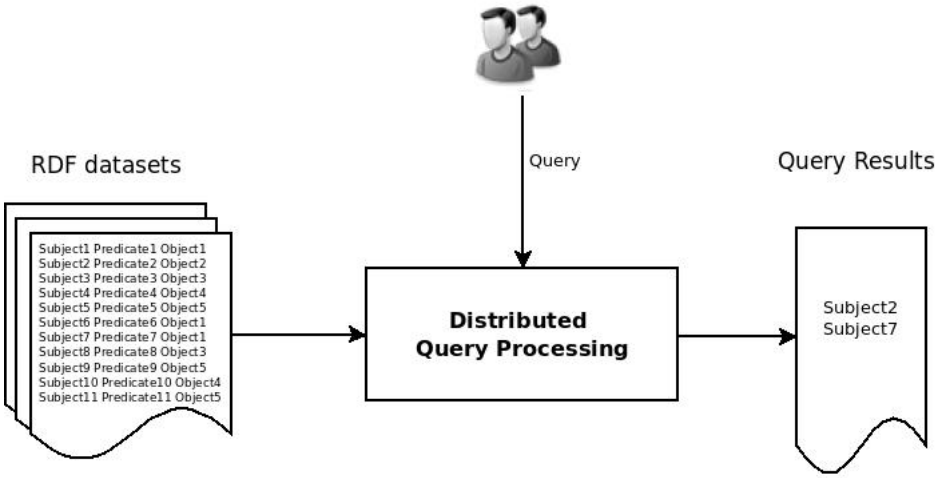


Figure 3.9: Our System - Generic DataFlow

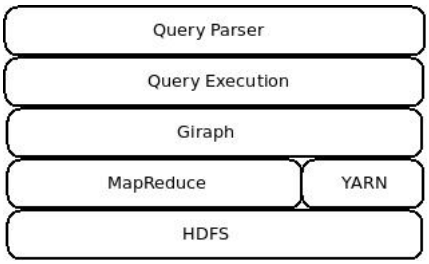


Figure 3.10: Conceptual stacked organization of Giraph applications

many times he wants to execute it, and the number of workers he wants to use. With the use of custom arguments, we are able to run the same type of query, using different values for the subjects/predicates/objects, involved in the query, thus there is no need to implement the query execution part many times, with different arguments each time.

- **Query Execution** This involves the code run after the user has inserted his preferences, and the job is submitted. At this point, runs the necessary code for loading the data, do the calculations and write the output. Note that this code is run in all workers, (the number of which is defined by the user initially).

3.4.2 Preprocessing

As mentioned before, the input to our system is the RDF datasets. This is not exactly the truth, in the sense that RDF datasets are copied to HDFS after a pre-processing is done, and the query execution operates on this processed dataset which contains exactly the same information as the original datasets, but in a different form. This pre-processing is required, since Giraph cannot automatically translate into graph those datasets, if they are in any of the existing serialization formats. In our experiments, we chose to use datasets in N-Triples format, and implemented a MapReduce job which runs over the data sets and groups RDF statements by object. The grouping could have been done by subject, or even predicates, but the chosen way will help us more in the implemention of our queries that follows. The result of this pre-processing is a group of adjacency lists, where the first element is the object, followed by its initial value: zero in our case, and subject-predicate pairs. The initial value, could be different, and any other value other than ‘one’, or one of the values with special meaning, used through the exchanged messages among vertices during computation time.

This procedure is illustrated in Figure 3.11.

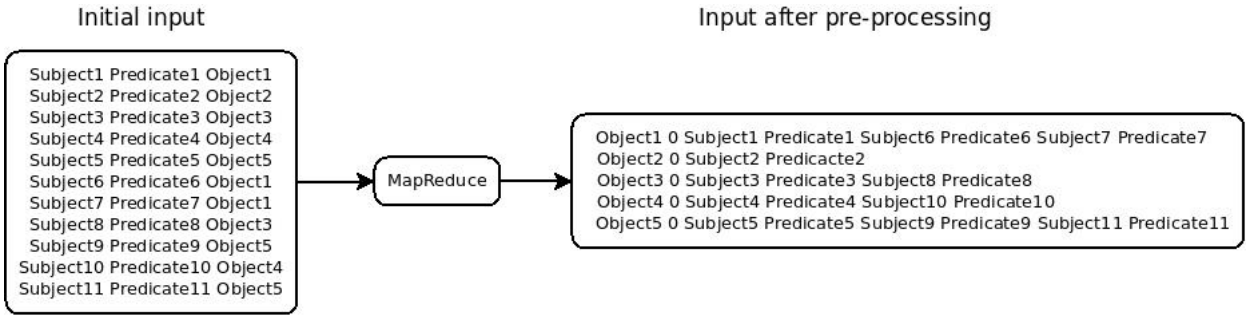


Figure 3.11: Pre-processing

3.4.3 Implementation of Queries

In order to implement the aforementioned queries in Giraph three basic things are required from our side:

1. Appropriate Input Format
2. Appropriate Output Format
3. Vertex Algorithm

Input Format Graphs may be stored in a storage system in a variety of formats. Giraph needs to have a way to read this data from the storage system and translate it to a graph. In essense, it needs a way which specifies how to read the data and convert them to *Vertex* and *Edge* objects. This is done by the various *Input Formats*.

Giraph already offers an API for this purposes, however it cannot be directly used in order to read the RDF datasets in N-Triple (or any other format). The existing API needs a way to retrieve and alter already created nodes and it is assumed that it will get exactly one line for each vertex to create. That one line is assumed to hold all the information necessary to create the vertex. The N-Triples format though does not work that way, as it can use multipe lines to describe the same node. This is the reason, we chose to do a pre-processing using a MapReduce job. Now, all the information regarding an object can be found in the same line, and we can write an appropriate input format, which can translate this information into a graph.

In specific, now (after pre-processing) our data consist of mulitple lines of the form:

Object1 0 Subject1 Predicate1 Subject2 Predicate2 ... SubjectN PredicateN

Here, Object1 will now denote a Vertex with VertexId: Object1 and initial value zero. Each of the following subject-predicate pairs are the vertex-neighbours of the initial vertex, having VertexIds: Subject1, Subject2, ..., SubnectN accordingly. Similarly, the outgoing edges values are Predicate1, Predicate2, ..., PredicateN.

The corresponding graph representation is illustrated in Figure 3.12

The Giraph API for vertex-based graph representations is *Vertex Input Format*. Figure 3.13 shows a small sample of the vertex-based input format classes in Giraph code base.

The white boxes represent existing abstract classes, while the colored box: *TextIntTextAdjacencyListVertexInputFormat* is owr own implementation, which extends the abstract class *AdjacencyListTextVertexInputFormat*.

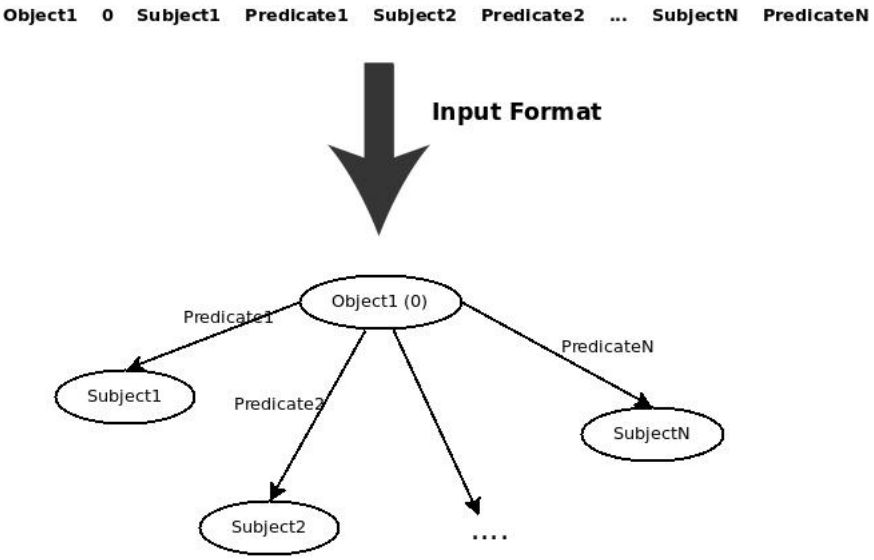


Figure 3.12: Graph Representation

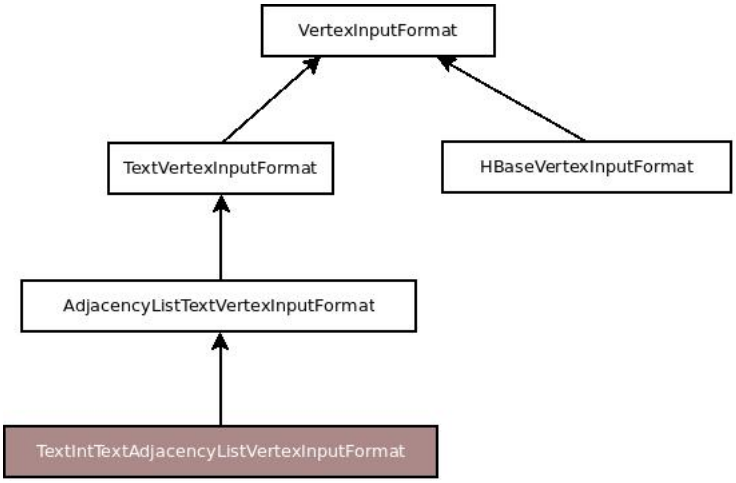


Figure 3.13: Giraph Input Format

Output Format The exact oppossite procedure, that is translating the graph into a format which can be saved in a storage system, does the *Output Format*.

In specific, the result of our queries exist in the Vertex and Edge objects in the graph. In our case, all the Vertices with value '1', satisfy the query criteria, so as output we would like to have only the VertexId of the vertices that satisfy the query criteria.

This graph transformation into our output can be seen in Figure 3.14

Similarly to the input format, we wrote our custom output format by extending the abstract class *TextVertexOutputFormat*.

Vertex Algorithm - Compute Method At this point each worker has been assigned one or more partitions of the graph, and now is responsible for calling the compute function for all the vertices resident on that worker. This compute function performs the whole calculation of the answer, and is run by every worker. In order to implement the Compute Method, which every Vertex is going to run, we used the following approach.

We see the sparql queries as trees, with root the subject/object we are looking for, some intermediate vertices/edges and leafs. As a general rule, in order to solve the query in giraph, we start with the

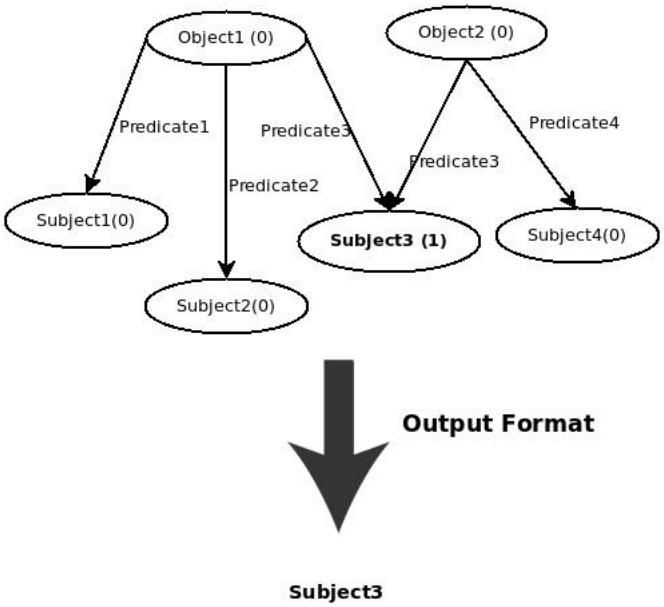


Figure 3.14: Giraph Output Format

leafs. In the first superstep, all vertices check if they are the desired leafs in the current query and if their outgoing edges match those of the current query. If so, they send appropppriate messages to their neighbours and they halt. From the second superstep till the problem is solved, the vertices which have received the initial messages from the vertices-leafs become active, check if they match specific characteristics and depending on the results they might send appropriate messages to their neighbours. The same procedure continues, till the result is found. In the final superstep the vertices that satisfy the query criteria have value one. No other messages are exchanged and the vertices that satisfy the query criteria halt. Since, computation has been completed, output is written and the bsp program is finished.

We are going to see step by step how this procedure is followed, in order to make it clear to the reader. Let's take for example: Query6, which is illustrated in Figure 3.7.

First of all, with the Input Format, we load the whole data set as a graph. Now, with the Compute method, we are basically searching inside this graph for a subgraph of the form of the graph shown in Figure 3.7, with the difference that edges have the exact opposite direction, as shown in Figure 3.15.

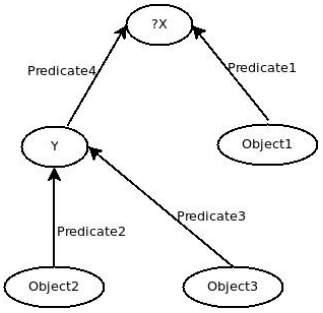


Figure 3.15: Query6 - Giraph

So, let's say, a part of the graph loaded in Giraph is the following 3.16.

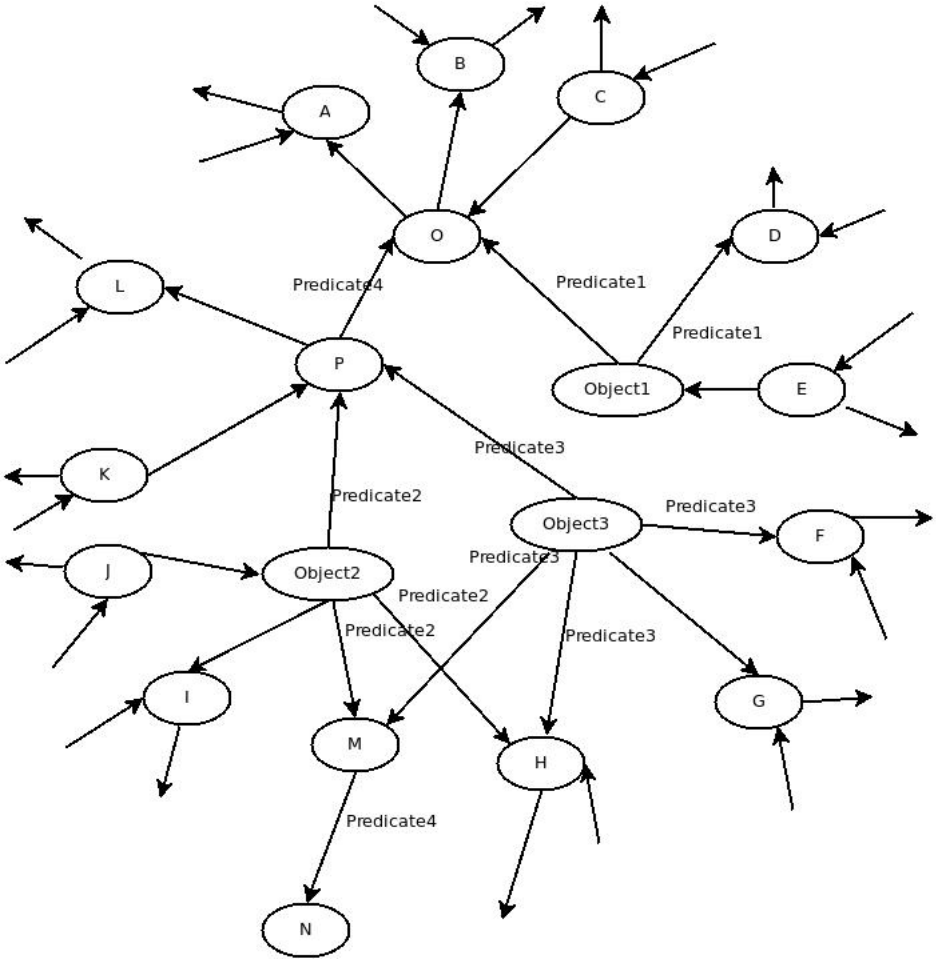


Figure 3.16: Part of the graph loaded in Giraph

The query execution in this type of query is completed in three supersteps.

Superstep 0

In the initial superstep, each vertex checks its own VertexID. If it is not one of the Object1, Object2, Object3, the vertex does nothing, and it simply votes to halt. If it is Object1, then checks if Predicate1 is one of its outgoing edges. If so, it sends the value '4' to the vertex connected with the edge with Value 'Predicate1', and votes to halt. Similarly vertex with Id: Object2 and outgoing edge Predicate2, sends the value '1' to the corresponding vertex-neighbour - possible Y, and vertex with Id: Object3 and outgoing edge Predicate3 sends the value '2' to the corresponding vertex-neighbour - possible Y. Both those vertices, halt right after they send the messages. We can see the following in Figure 3.17

Note that here the values sent through the messages from the one vertex to another, can be different. In our case we do not want to pass any specific information through the message. We just use it as a way to denote that a vertex has received a message from a vertex that is part of our query, and also in order to make a possibly inactive vertex, active again. Hense, instead of '4', any other Integer value can be used, probably part of an enum type, which in our implementation code has specific meaning. For example, here '4' means that 'I received a message from Vertex with VertexID: Object1, with which I am connected through an edge with value Predicate1'.

Superstep 1 Now in superstep 1, all the vertices are inactive (since all have voted to halt in the previous superstep), except for those who received a message. Hence, in our example, now active are O, D,

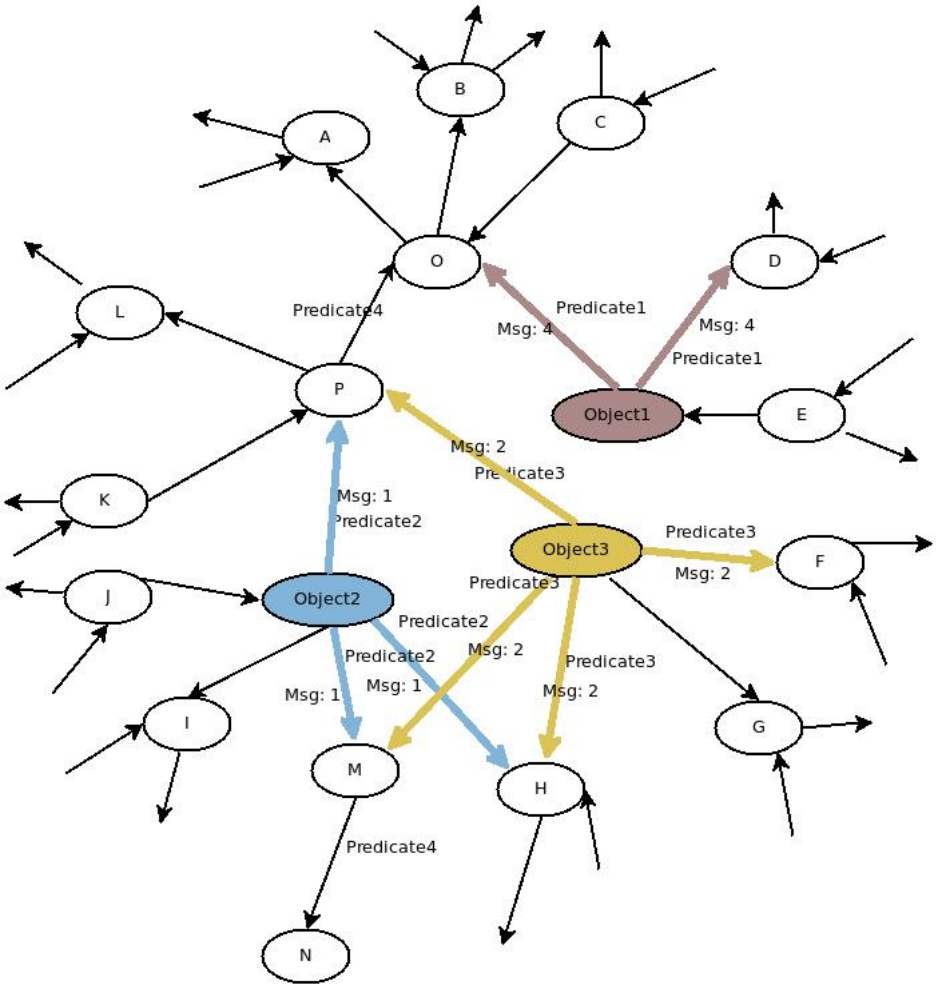


Figure 3.17: Superstep 0

P, M, H and F. O and D have received message with value: 4 from vertex with Id: Object1. P, M and H have received message with value: 1 from vertex with Id: Object2. P, M, H and F have received message with value: 2 from vertex with Id: Object3.

In this superstep, each of the active vertices, checks the messages it got and acts accordingly. O and D have received message with value: 4 from vertex with Id: Object1. O and D, are for now, canditates for being the answer to the query. In this superstep they just set their own value to 5 and vote to halt.

Similarly, all vertices which received a message both from Object2, and Object3, (in our case P, M and H) check if the have outgoing edge with Value: Predicate4 and if so, they send to the possible X message with Value '3'. After that they vote to halt. Note that F, does nothing, since it has received message only from Object3. Correspondingly H, although received both required messages from Object2 and Object3, and it does not have an outgoing edge with value: Predicate4, hence it votes to halt as well, without sending any other message.

The above can be seen in Figure 3.18

Superstep 2

In our final superstep active are only the vertices, which have received a message in the previous superstep. In our example, those are: O and N. They check if they received a message with the desired value (3) and if so they check their own value. If their own value is '5', meaning that they have received a message in superstep0 from Object1, then they fullfil all the essential requirements, satisfy the query criteria, are part or the asnwer to our query and they set their own value to 1. Here, this is the case with

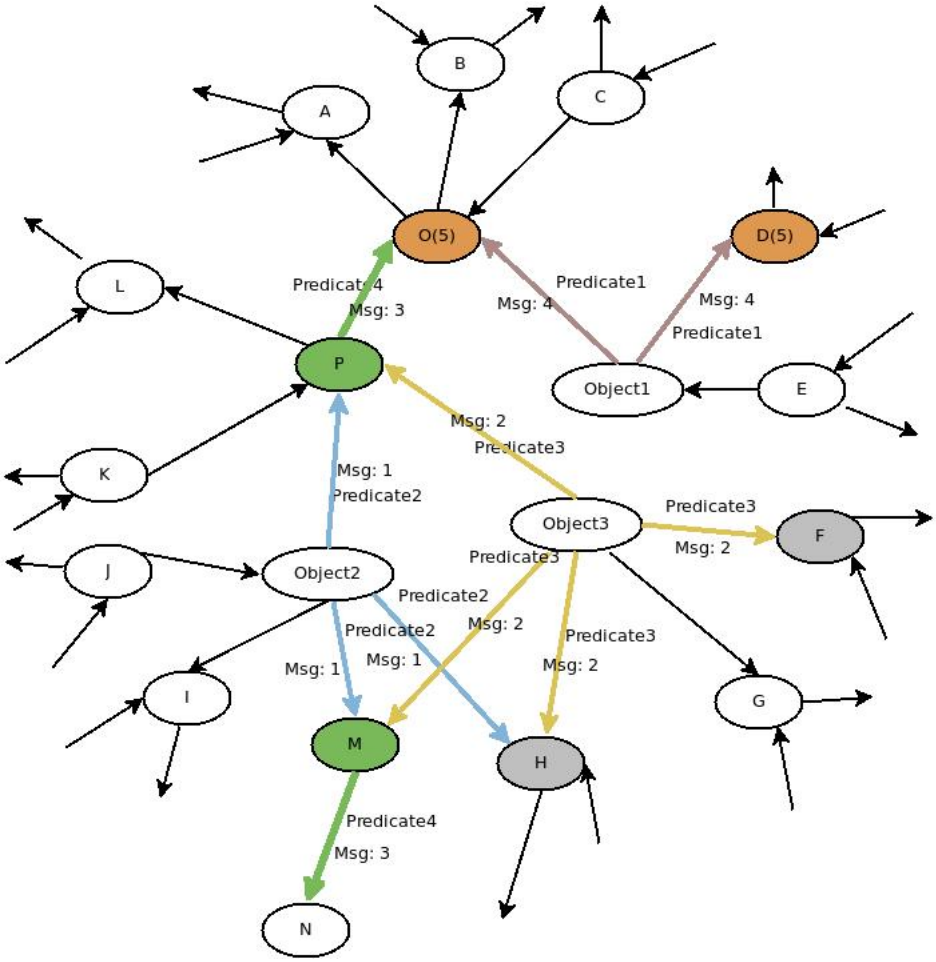


Figure 3.18: Superstep 1

vertex O. Vertex N does receive a messsage with the desired value, but it is not connected to Object1, so it is not part of the answer. Similarly, vertex D which was marked as possible answer in superstep0 since it was connected to Object1, is not part of the answer, since it is not connected with a vertex, which has the 'characteristics' of Y, as a result it has not received any message and is inactive in the final superstep.

We can see the aforementioned changes in Figure 3.19

After superstep 2 all vertices have become inactive, so this is also our final superstep. After its completion, the output-answer to our query is written to HDFS. Obviously, here is just an example and only vertex O satisfies the query criteria, something that may be different in a real dataset, where more than one vertices can be part of the answer.

3.5 Giraph Times

In order to be able to see how much time is needed for the completion of a query using giraph, we used the pre-existing giraph timers. They are shown in stdout, right after the completion of the job.

So, let's describe them, in order to know what they measure, and which of them are useful to us.

- **Initialize** It measures the time spent by the job waiting for resources. In a shared pool the job you launch may not get all the machines needed to start the job. So for instance, you want to

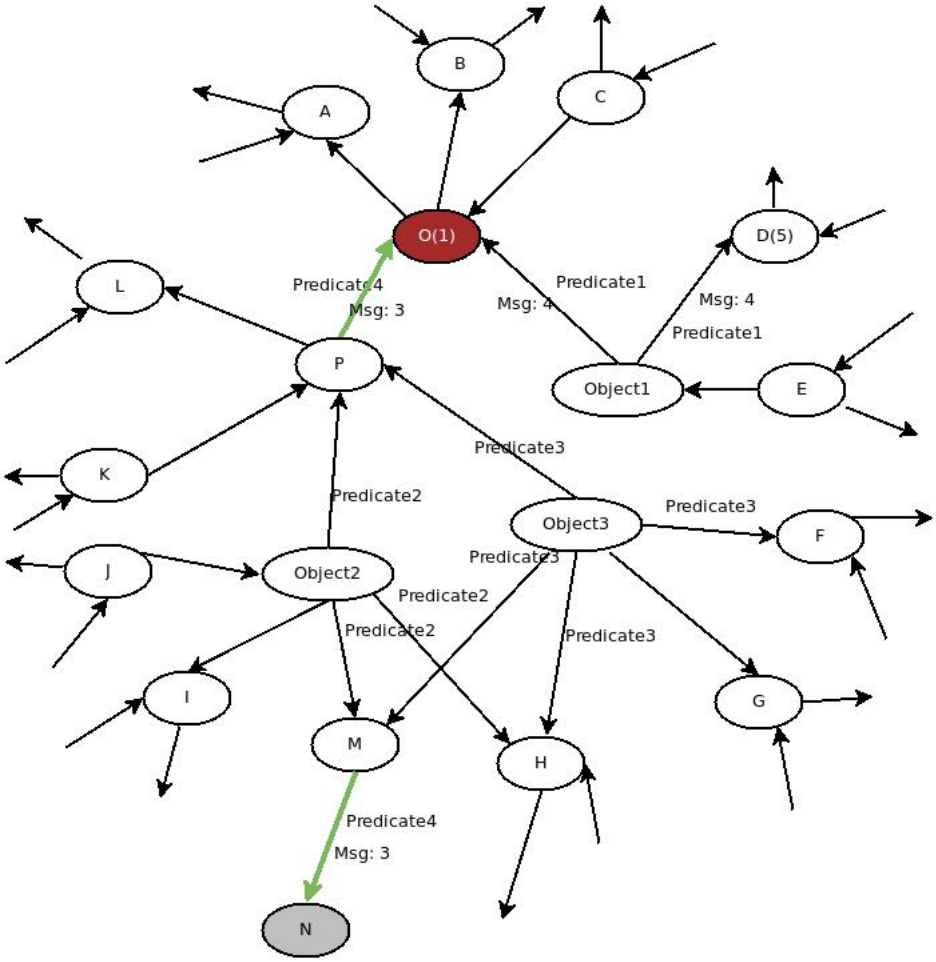


Figure 3.19: Superstep 2

run a job with 200 workers, giraph does not start until all the workers have are allocated and registered with the master.

- **Setup** It is the time it takes before starting to read the input, once you have all the machines allocated.

- **Input Superstep** The time needed to read the input(vertices/edges) provided into memory on individual workers, assign vertices to partitions and partitions to workers, moving all partitions to a worker (which own the partition) and doing some bookeeping of internal data structures to be used during computation.

- **Superstep i** Here the user-define function compute takes place.

- **Shutdown** The time it takes to stop and verify that everything id done, resources are shutdown and user is notified, once you have written your output. This could mean the time needed to wair for all network connections to close, all threads to join, etc.

- **Total** It's the actual time take to run by your application after it got all resources. It does not include initialize or shutdown time, so it's basically the sum of Setup, Input Superstep, and the sum of the time in all supersteps.

3.6 Running in openvirtuoso

To evaluate the achieved performance of our system to a widely used centralized approach for querying such datasets, we chose OpenVirtuoso [35]. OpenVirtuoso is by far the most widely deployed SPARQL endpoint. Among others, Virtuoso is implemented as the SPARQL endpoint for DBpedia and DBpedia Live, LinkedGeoData, Sindice, the BBC, BioGateway, data.gov, CKAN, and the LOD Cloud Cache [36].

In OpenVirtuoso, the queries were executed using SPARQL. Below you can find the exact queries we executed, defined in SPARQL.

```
Query1a
sparql select ?x where { ?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> . ?x
  <http://dbpedia.org/property/birthPlace>
  <http://dbpedia.org/resource/Queens> .};

Query1b
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> . ?x
  <http://dbpedia.org/property/birth>
  '1979'^^<http://www.w3.org/2001/XMLSchema#date> .};

Query1c
sparql select ?x where {?x <http://xmlns.com/foaf/0.1/name>
  'Sokrates'@de . ?x <http://purl.org/dc/elements/1.1/description>
  'griechischer Philosoph'@de .};

Query1d
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/City108524735> . ?x
  <http://dbpedia.org/property/populationAsOf>
  '2006'^^<http://www.w3.org/2001/XMLSchema#Integer> .};

Query1e
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/Emirate108557396> . ?x
  <http://dbpedia.org/property/mapCaption>
  <http://dbpedia.org/resource/Saudi_Arabia> .};

Query2a
sparql select ?x where {?x <http://xmlns.com/foaf/0.1/name> 'Verne,
  Jules'@de . ?x <http://dbpedia.org/property/deathPlace>
  <http://dbpedia.org/resource/Amiens> . ?x
  <http://xmlns.com/foaf/0.1/givenname> ?y1 . ?x
  <http://xmlns.com/foaf/0.1/surname> ?y2 . ?x
  <http://dbpedia.org/property/birthPlace> ?y3 .};

Query2b
sparql select ?x where {?x <http://dbpedia.org/property/influences>
  <http://dbpedia.org/resource/Plato> . ?x
  <http://dbpedia.org/property/influenced>
  <http://dbpedia.org/resource/Alexander_the_Great> . ?x
  <http://dbpedia.org/property/shortDescription> ?y1 . ?x
```

```
  <http://dbpedia.org/property/notableIdeas> ?y2 . ?x
  <http://dbpedia.org/property/mainInterests> ?y3 .};

Query2c
sparql select ?x where {?x <http://dbpedia.org/property/influences>
  <http://dbpedia.org/resource/Plato> . ?x
  <http://dbpedia.org/property/influenced> ?y1 . ?x
  <http://dbpedia.org/property/shortDescription> ?y2 . ?x
  <http://dbpedia.org/property/notableIdeas> ?y3 . ?x
  <http://dbpedia.org/property/mainInterests>
  <http://dbpedia.org/resource/Science> .};

Query3a
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/Book> .};

Query3b
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> .};

Query3c
sparql select ?x where {?x <http://xmlns.com/foaf/0.1/homepage>
  <http://www.educause.edu/> .};

Query4
sparql select ?x where {<http://dbpedia.org/resource/\%21\%21\%21>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/Group100031264> . ?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/Group100031264> . ?x
  <http://dbpedia.org/property/genre>
  <http://dbpedia.org/resource/Alternative_rock>
  . <http://dbpedia.org/class/yago/Group100031264>
  <http://www.w3.org/2000/01/rdf-schema#label> 'Group' .};

Query5
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> . ?x
  <http://xmlns.com/foaf/0.1/name> ?z . ?x
  <http://dbpedia.org/property/birthPlace> ?y . ?y
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/City108524735> . ?y
  <http://dbpedia.org/property/populationRef>
  <http://www.statistik.baden-wuerttemberg.de> .};

Query6
sparql select ?x where {?x
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/Person> . ?x
  <http://dbpedia.org/property/birthPlace> ?y . ?y
  <http://dbpedia.org/property/populationRef>
  <http://www.statistik.baden-wuerttemberg.de> . ?y
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://dbpedia.org/class/yago/City108524735> .};

```

```
106 Query7
107 sparql select ?x where
108 {<http://dbpedia.org/resource/Ferdinand_Rudolph_Hassler>
109   <http://dbpedia.org/property/birthPlace> ?x . ?x
110   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
111   <http://dbpedia.org/class/yago/Capital108518505> .};
```

Chapter 4

Experiments

4.1 Experimental Setup

4.1.1 Datasets

In our experiments we used two datasets with different size from the DBpedia project:

1. Dataset 1: http://downloads.dbpedia.org/2.0/
2. Dataset 2: http://downloads.dbpedia.org/3.0/en/

Those datasets contain articles, abstracts, images, links to external web pages, information that has been extracted from Wikipedia infoboxes, information about persons, links between geographic places in DBpedia and data about them in the Geonames data base, as well as links to RDB Book Mashup, to DBLP, Eurostat, Project Gutenberg, MusicBrainz and many others. More info about the content of the aforementioned datasets can be found in the wiki of the the DBpedia community. Specifically, in the following links:

- http://wiki.dbpedia.org/data-set-20
- http://wiki.dbpedia.org/data-set-30

Both of them contain N-Triple data. The total disk size of the datasets before and after the MapReduce pre-processing can be seen in table 4.1:

Dataset Name	Original Disk Size	Disk-Size After Pre-processing
Dataset 1	6.8GB	5.2GB
Dataset 2	18.0GB	13.0GB

Table 4.1: Disk-size of used datasets

4.1.2 Testing environment

All of our experiments were conducted in cloud environment, using Openstack as software platform. We created 8 virtual machines (VMs) hosted in the Computer System Laboratory of NTUA. The VMs were running Ubuntu 14.04.2 x64, had 8GB RAM each, 4VCPUs and 80.0GB disk storage. For our experiments we used:

- Hadoop version: 0.20.203.0
- Giraph version 1.2.0 and
- Openvirtuoso version 6.1.

Since we were concerned about the variability of the results, we executed each query five times and then we took the average time as the final result.

4.2 Experimental Results

4.2.1 Choosing hadoop parameters

Giraph exploits the existing Hadoop infrastructure and our queries are submitted as map-only Hadoop jobs. A Hadoop job may have quite different performance depending on the chosen values for its configuration parameters. Hence, it is reasonable to assume that finding the right values for some configuration parameters may be crucial in order to have a good performance in Giraph job. Here, we studied the impact, if any, of the following parameters:

1. dfs.replication
2. dfs.block.size

The first specifies how many times the local filesystem is copied to the HDFS, mostly to ensure high availability of the data, whereas the second specifies the size of the blocks into HDFS.

We investigated the impact of the aforementioned parameters in the following phases:

1. Copy the input from the local file system to HDFS
2. Load the graph in Giraph
3. Query Computation time

Copy to HDFS

In Figure 4.1 is illustrated how the copy time changes, when the replication factor increases.

As expected, as the replication factor increases, it takes more time to copy the local filesystem to HDFS. However, we observed that with replication factor equal to two, the copy time is comparable to that of replication factor one, while at the same we have a basic high availability of the data, so it is obviously preferable. Bigger values of the replication factor add considerably more time to the copy phase.

In Figure 4.2 is illustrated how the copy time changes, when the block size increases.

It seems that block sizes from 128Mb to 1408Mb have comparable performance, with the copy phase needing less time when the blocksize is equal to 1408Mb (or 1.375GB). Bigger blocksize though doubles the time needed from the copy phase.

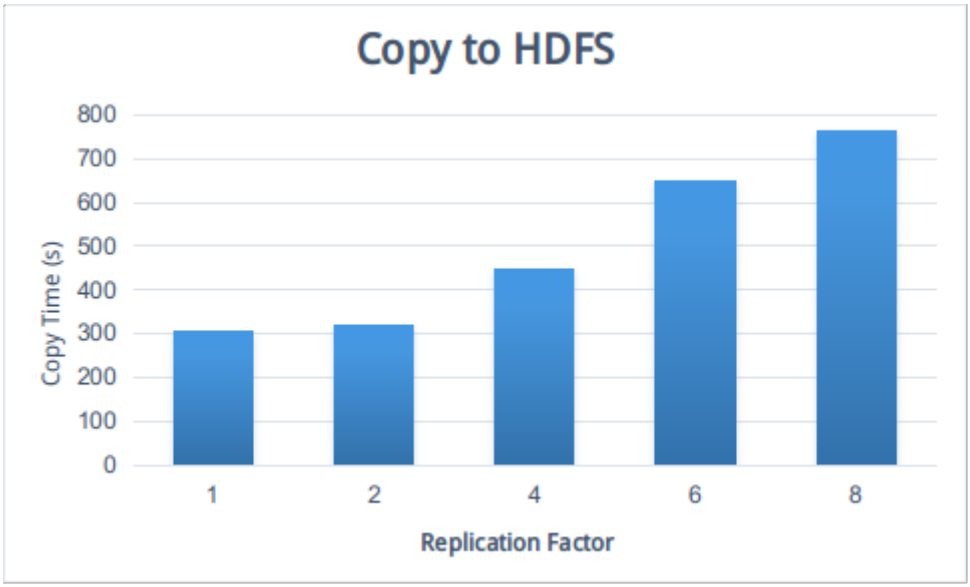


Figure 4.1: Copy from local filesystem to HDFS vs dfs.replication

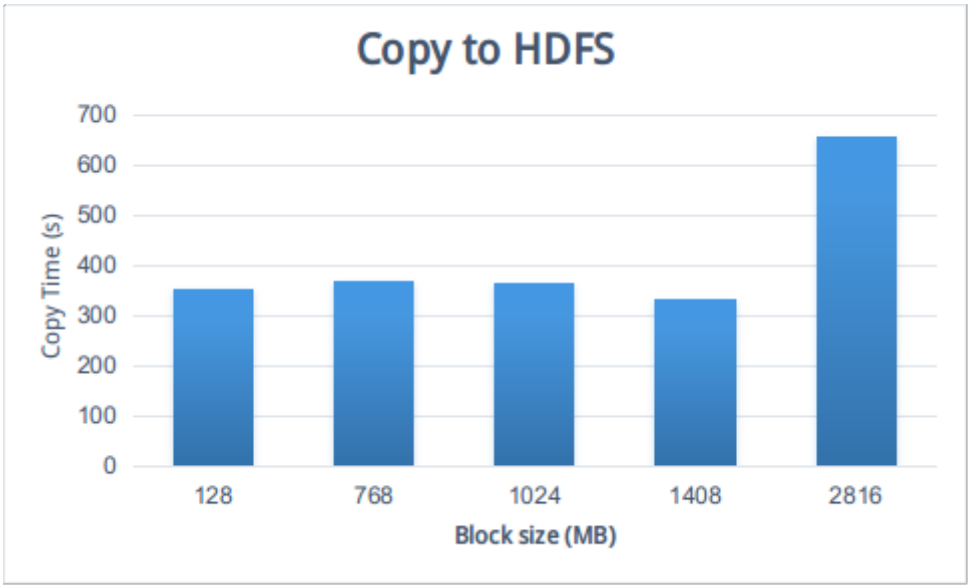


Figure 4.2: Copy from local filesystem to HDFS vs dfs.block.size

Load Graph in Giraph

In Figure 4.3 is illustrated how the time to load the graph changes, when the replication factor increases.

We can see that loading the graph takes significantly more time when the replication factor is one, compared to any other value. This is expected, since data reside only in one node and each worker needs to load the graph data according to its assigned input splits, making the read time significantly lower.

All the other values of replication factor are closer to each other, with the value *two* continuing to be the best option.

In Figure 4.4 is illustrated how the time to load the graph changes, when the block size increases.

With blocksise 2816MB, all runs failed. This is expected, since each worker has to load the input splits

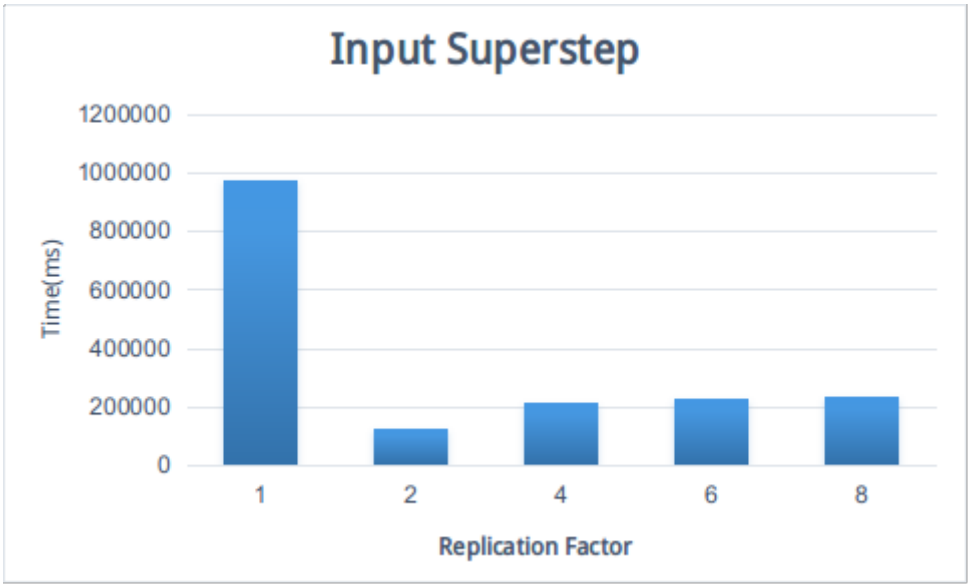


Figure 4.3: Input Superstep vs dfs.replication

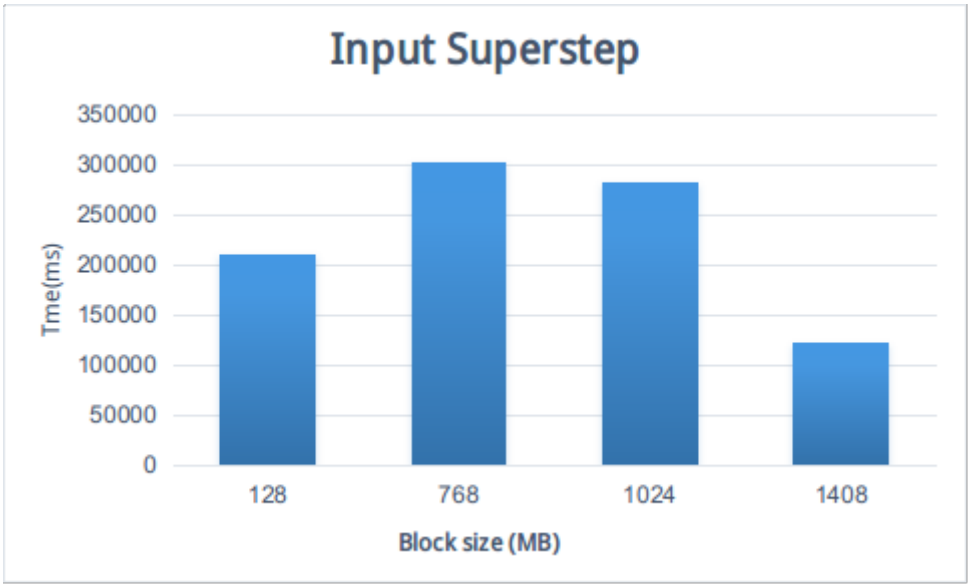


Figure 4.4: Input Superstep vs block size

assigned to this node by the master to its own local memory Apart from that memory should be allocated in order to store the Vertex and Edge Objects, with their values, as well as the incoming/outgoing messages, making the memory required to exceed the 8GB memory it has.

Regarding the rest of the values, we can see that there is no consistent behaviour of the load time as the block size increases. The best results are achieved with block-size: 1408MB. It seems to be a good option, since its size is big enough, to have less metadata and the handling that comes with them, but also small enough, to fit into a single worker's memory and thus not causing failures, due to memory overload.

Computation Time

In Figure 4.5 is illustrated how the computation time changes, when the replication factor increases.

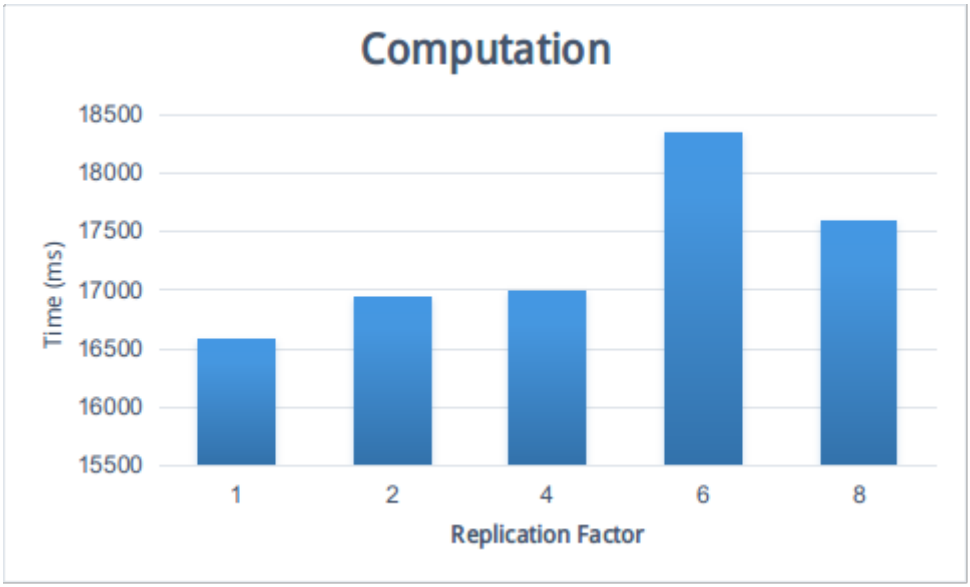


Figure 4.5: Computation Time vs replication factor

It is a bit suprising that the computation seems to take less time, when the replication factor is one. In this case the copy to HDFS results in having all the HDFS data stored in blocks in the same node they were stored before the copy. Thus, the biggest part of the input graph resides in one worker and the communication between the rest of them is the minimum required.

In Figure 4.6 is illustrated how the computation time changes, when block size increases.

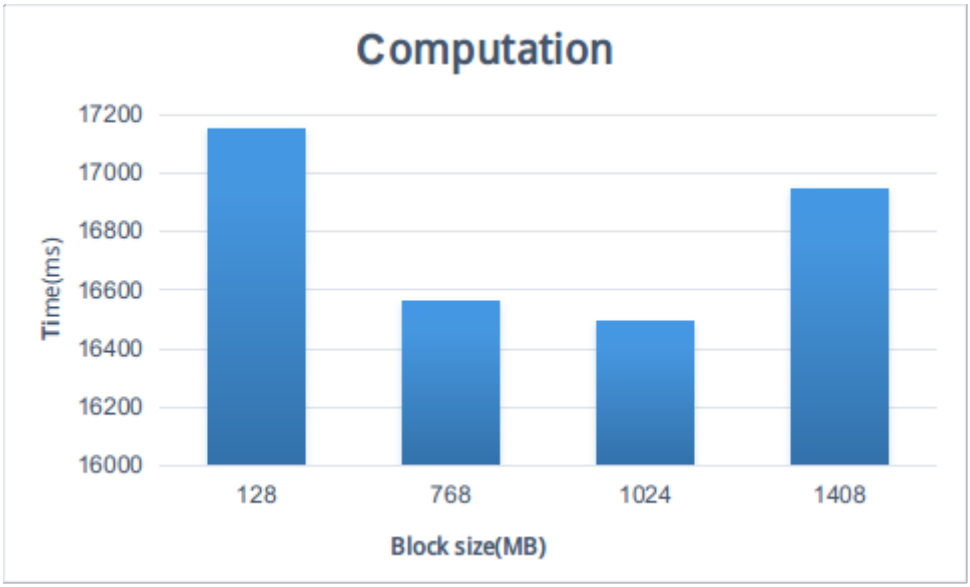


Figure 4.6: Computation Time vs block size

In this figure, we observe that computation takes less time as the block size increases. This is valid, for block sizes beginnign from 128MB till 1024Mb. Block sizes bigger than the last value result in bigger computation time. This occurs, since there are less options for partioning the graph, resulting in worse load-balancing, meaning that we have workers that need much more time to calculate their own results than others.

The aforementioned measurements, were taken from runs in which we used only Dataset 1. However,

we executed some of the runs using also the bigger dataset, and it showed that the same behaviour is valid there as well.

4.2.2 Time vs Different Type of Queries

Aim of this section is to identify the differences in computation time between queries of different type, if the same behaviour remains when the number of nodes increases, and how it can be explained.

The results concern all of the seven different types of queries and they are illustrated in Figures 4.7 and 4.8 for Dataset 1 and Dataset 2 accordingly.

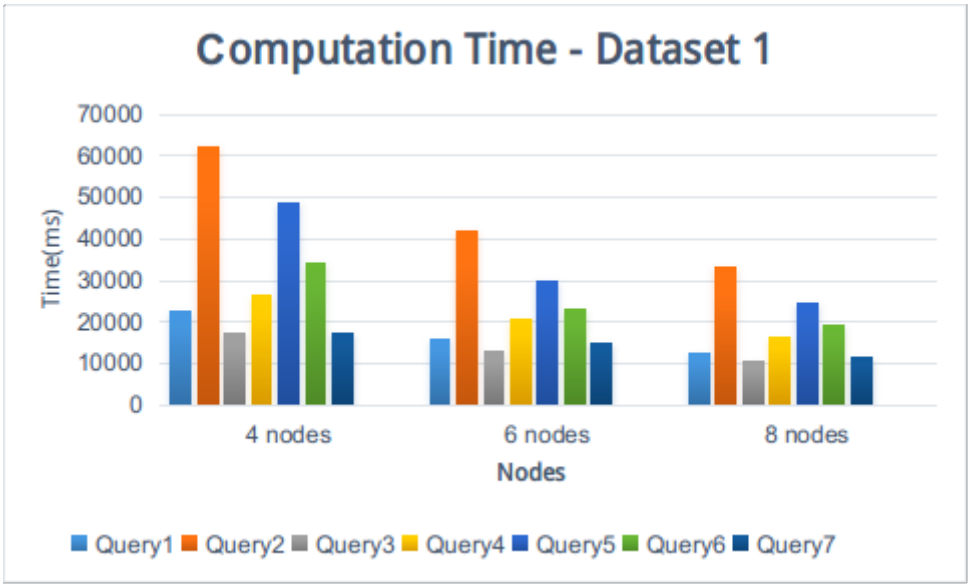


Figure 4.7: Computation Time - Dataset1

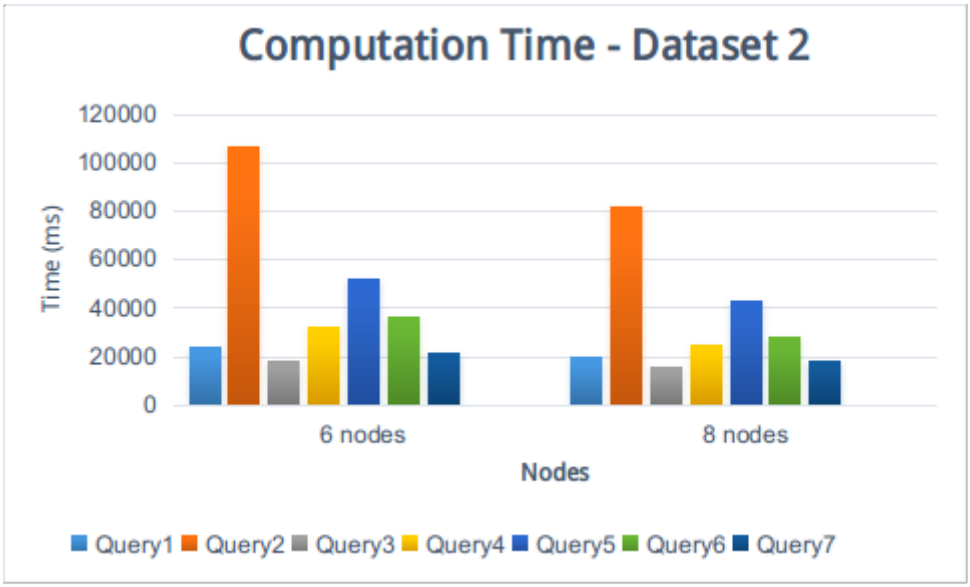


Figure 4.8: Computation Time - Dataset2

We can see that in all cases the behavior of the queries in terms of computation time. is the same, either we have four, either six or eight numbers of nodes. This means, that if Query1 takes less time to be executed than Query2 in a cluster composed of 4 nodes, then it would also need less time than

Query2 in a cluster of 6 or 8 nodes. The same is valid for the Dataset 2. In Table 4.2 the different types of queries are shown, ordered in a row starting from the one needing the less execution time and ending with the one requiring the most.

Query3	Query7	Query1	Query4	Query6	Query5	Query2

Table 4.2: Queries, sorted by computation time

Since, this behaviour is valid in every case, it is safe to assume that it is not random, but based on the inherent characteristics of those queries. The characteristics that seem to affect at most the computation time are:

1. The number of supersteps, or let's say the 'height of the tree' as can be seen in the given figures.
2. The complexity of the query, which is mostly determined by the number of vertices and edges involved in the query and how they are chained together.

Regarding the first factor, it is reasonable to assume that the less the number of the required supersteps are, the less the computation time is going to be. This is because with each superstep comes along a synchronization barrier, where each worker waits for the rest of them to finish, something that is necessary on the one hand, but may result in much bigger computation time, especially in cases one worker is much slower than the rest of them. In our case, Query 1, Query 2, Query 3 and Query 7 need two supersteps, while Query 4, Query 5 and Query 6 need three. We see that in general the queries with two supersteps need less time to complete from the others with three. The only exception to this rule, seems to be Query2 which has the biggest computation time, but needs only two supersteps. Other factors, which are going to be explained later, play most important role in this case.

Complexity seems to play also a crucial role in the query computation time. Let's take the queries one by one. Query 3, is the simplest one and it is expected to require the less computation time. In particular, it just requires for exactly one vertex with specific VertexId to send message or messages to all the vertices which are connected with it with edge that has a specific value. All the vertices that received a message are part of the answer.

Query 7, slightly more complex than Query 3, including one more vertex and edge. The computation is performed in the same way as Query 3, but with one additional condition: The vertices that have received a message in the previous superstep are part of the answer only if they have are connected to a specific vertex with a specific outgoing edge. Thus, it is expected, to require a little more time than Query 3.

Query 1, does is not more complex than Query 7, in the sense that the same number of vertices and edges are involved. However, the way the vertices are chained together is different, involving much more active vertices in the initial superstep, and an increased number of messages sent among the workers, thus leading to more computation time.

Query 4, increases much more the complexity, since it contains four properties (edges values), three vertices with specific VertexID, and a set of vertices 'Y', which are connected with the other vertices involved in a specific way. With a little more observation, we can see that it is a combination of Query 7 and Query 1, thus it is expected to require considerable more time than the previous three queries.

Query 6, has the same number of vertices and edges as Query5, thus complexity is not that increased. It differs in how the the vertices are chained together, and it can be considered as a combination of two queries similar to Query 1, justifying why it required a little more time than Query 4.

Query 5, is again more complex than all the previous ones, since it contains one more property, and another set of vertices 'Z' with specific characteristics. Thereby, it requires more time than all the others.

Last but not least, Query2, which surprisingly requires much more time to be completed than all the others. As discussed before it requires only two supersteps and the way vertices and edges are chained together seems to be quite simple. However, the number of properties involved is large (five), and a large number of exchanged messages is required from different kind of vertices. Thus, the final number of vertices involved is quite large and all of them send message to the exact same vertex-answer. All those messages have to be handled by the same worker, the one which has in its memory the vertex-answer, hence reducing the level of parallelism and requiring more time for the completion.

4.2.3 Time vs Number of Nodes

Obviously, since we are in a distributed environment, we want to see how the computation time differentiates as the number of nodes increases. In our experiments, we run all types of queries, some of them with different choices as predicates, objects or subjects in different number of nodes. Each query was executed five times, and we took the average as the final result.

The results are illustrated in figure 4.9. Blue, orange and grey colors, represent that the number of nodes is 4, 6 and 8 accordingly.

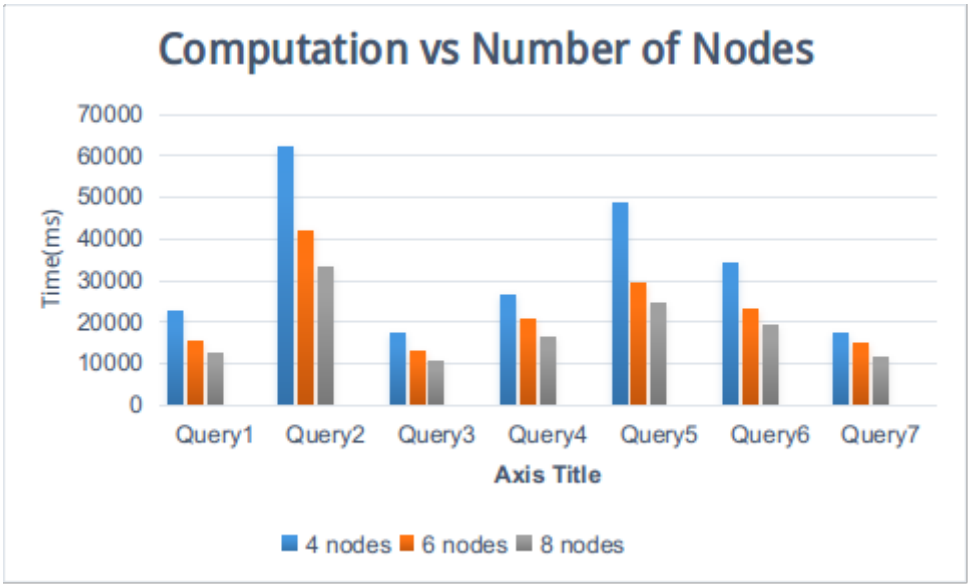


Figure 4.9: Computation Time vs Number of Nodes - Dataset 1

As expected, for all type of queries, the computation time decreases as the number of nodes increases for all types of queries.

The results for the bigger dataset are illustrated in Figure 4.10. In this case, the dataset was proven to be bigger than our memory could handle, thus the execution with only 4 nodes was impossible. It was always resulting on failing Giraph jobs. The results for 6 and 8 nodes though, were as expected, with the computation time being much less in all type of queries when the number of nodes is 8.

Time vs Different Instances of the same Query type

In order to see how the specific - instances influence the result, we run Query1, Query2 and Query3, using different parameters, and took the results in 4, 6 and 8 nodes. The results for those queries are illustrated in Figures 4.11, 4.12, 4.13.

Similarly for Dataset 2, we have figures 4.14, 4.15, 4.16, accordingly:

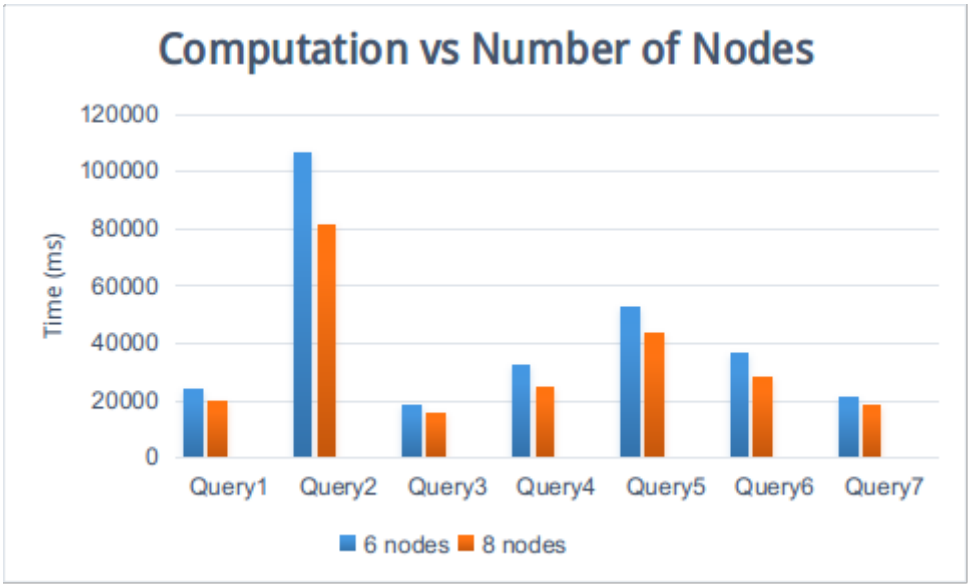


Figure 4.10: Computation Time vs Number of Nodes - Dataset 2

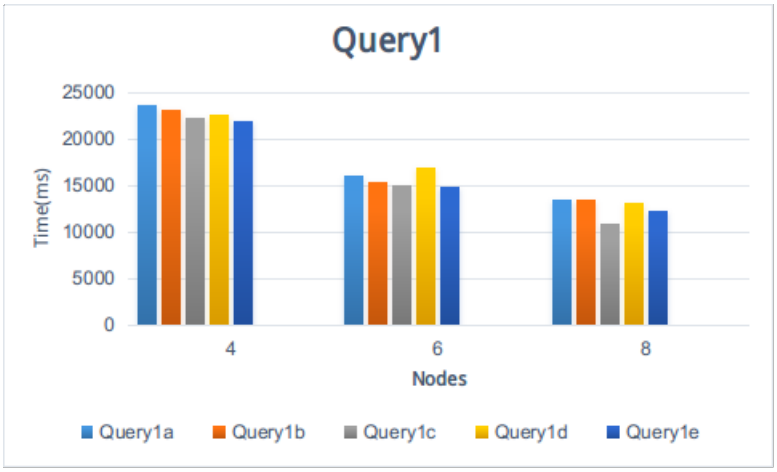


Figure 4.11: Query1 - Computation Times

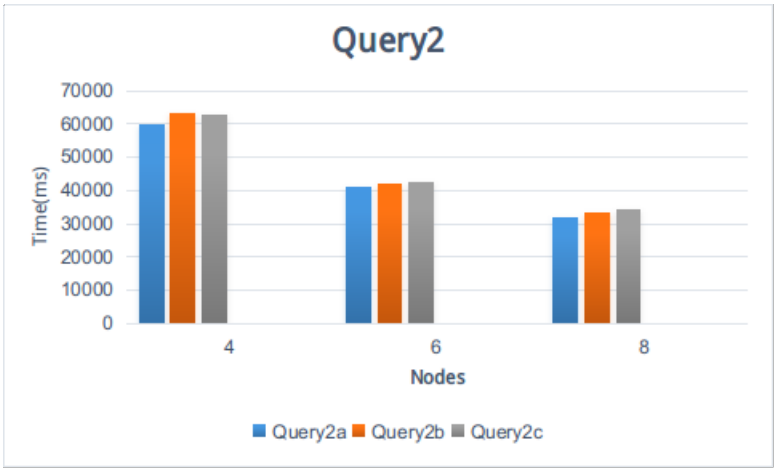


Figure 4.12: Query2 - Computation Times

From the above figures we can see how the choice of the subjects, predicates and objects we use can

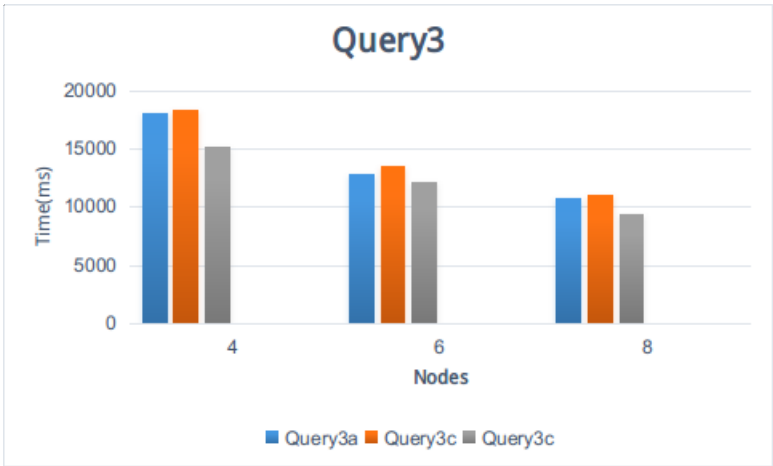


Figure 4.13: Query3 - Computation Times

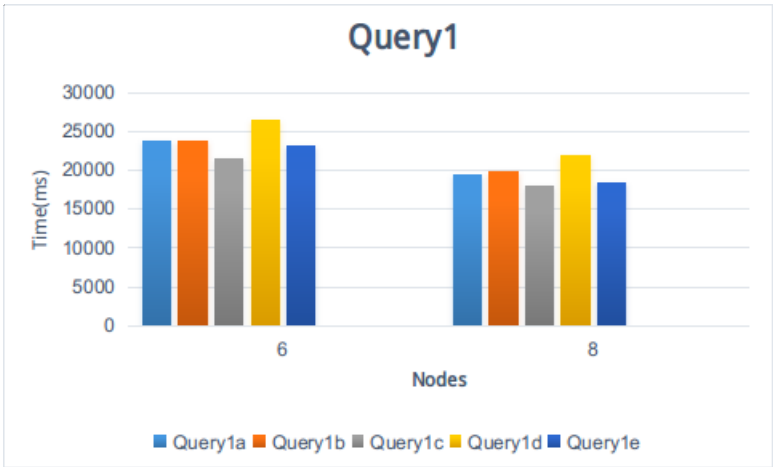


Figure 4.14: Query1 - Computation Times

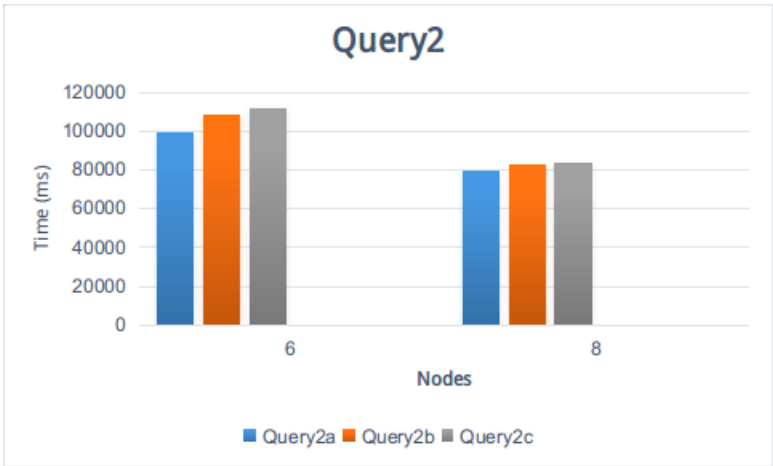


Figure 4.15: Query2 - Computation Times

lead to a quite different computation time. In all of these cases, the structure of the query is the same, only the various values differ. In specific, we can see in the following table 4.3 the various query instances, sorted by computation time.

Those differences can most of the times be justified by the number of the vertices involved in the

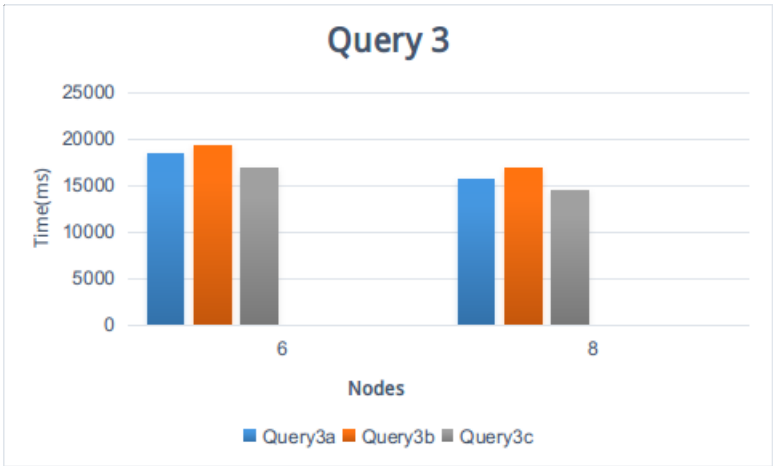


Figure 4.16: Query3 - Computation Times

Query1	Query1e	Query1c	Query1d	Query1b	Query1a
Query2	Query2a	Query2b	Query2c		
Query3	Query3c	Query3a	Query3b		

Table 4.3: Queries, sorted by execution time

query. As we can see in Figures 4.17, 4.18 and 4.19, for Query1 and Query3, the less the number of vertices is, the less the computation time is. Surprisingly enough, this is not valid for Query2, in which Query2a, takes the least time to execute comparing to the other two, although more vertices are involved in the query and more messages are exchanged. In this case, maybe other factors play more important role, like how the graph is partiotioned and how the data are distributed across the workers.

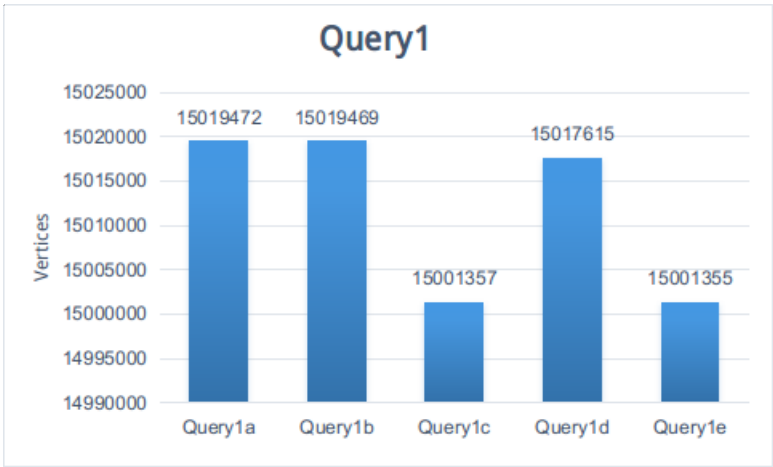


Figure 4.17: Query1 - Number of Vertices

The exact same behaviour can be observed in Dataset 2 as well.

4.2.4 Openvirtuoso

In order to verify that our way of implementation of the queries returns all the results, as well as the right ones, we compared those with the ones in openvirtuoso. In all cases, the results where the same.

Similarly, we wanted to see that the queries, behave the same way in both systems, and if not, the

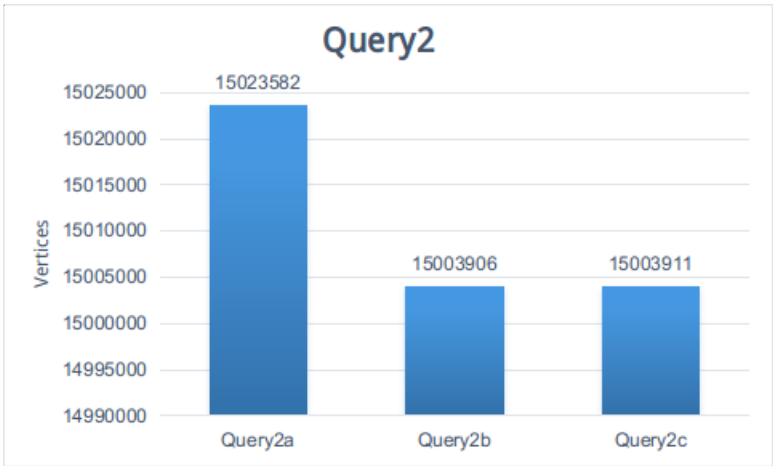


Figure 4.18: Query2 - Number of Vertices

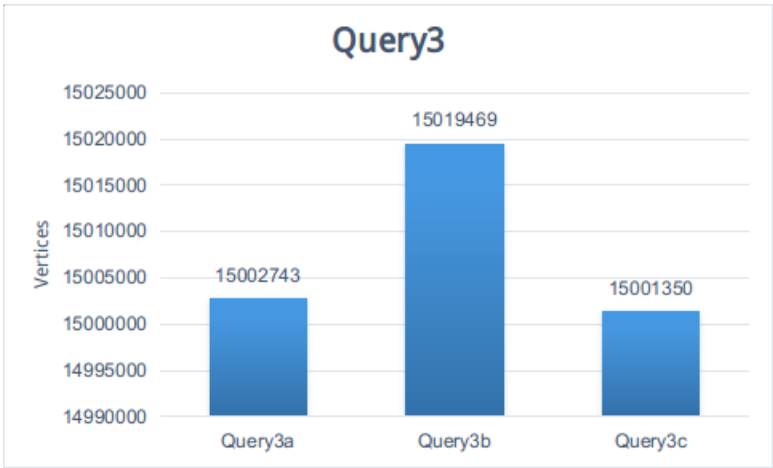


Figure 4.19: Query3 - Number of Vertices

reason underneath.

4.2.5 Load Graph

The time required from Openvirtuso to load all the RDF data, is quite big. We loaded the dbpedia graph, containing all the RDF datasets we used in Giraph, seven times and took the average as the final result. It seems that it needs about 6 hours to load the whole graph. The individual results for Dataset 1 can be seen in table 4.4. Although the comparison is not fair, in the sence that in Giraph we may use four times bigger resources than those in openvirtuoso, 6 hours seem to be too much, comparing to the 2 minutes that Giraph needs for the same dataset when we choose the appropriate configuration parameters.

4.2.6 Computation

On the other hand, the query response time is much better in OpenVirtuoso. The Computation part in Giraph can be even five times slower that this of Openvirtuoso, although we utilize the 8 nodes of the cluster. We should consider though, that this can be expected to a point, since Giraph is suitable for processing huge datasets, which may reside in hundreds of nodes. In such datasets, we assume that

Load Time (ms)	
Openvirtuoso	**Giraph**
23233681	118553
25289059	105188
12021112	128640
25706906	118150
23536321	134792
18211069	119050
22010159	114458
21429758.14	**119833**

Table 4.4: Load Time - Openvirtuoso - Giraph

we would see Giraph outperfoming Virtuoso in terms of computation time, and maybe even answer queries that openvirtuoso cannot even do. In such sizes, the communication overhead and the extra handling needed, in order to manage this distributed environment, is big enough, slowing down the whole procedure.

In addittion, we wanted to see how Openvirtuoso behaves with the different types of queries. The results for Dataset 1 are shown in Figure 4.20.

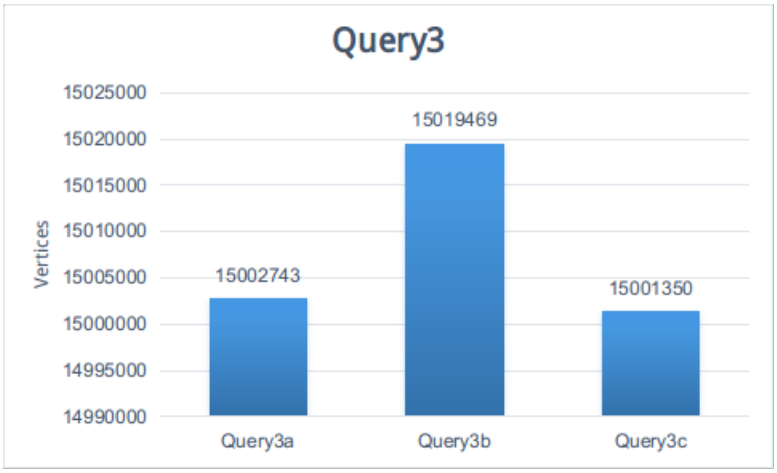


Figure 4.20: Computation time - Openvirtuoso

We see that the behaviour of the queries is quite different from that in Giraph. Here the queries sorted by computation time, can be seen in 4.5

Query7	Query2	Query5	Query1	Query6	Query3	Query4

Table 4.5: Queries, sorted by computation time

Those differences are due to the different way Giraph and Openvirtuoso process the query, as well as that due to the fact that writing the output in openvirtuoso needs much more time, comparing with that Giraph needs. Giraph workers can write their partial output to HDFS simultaneously, while in Openvirtuoso this has to be done sequentially. This may be the reason why query3, although the simplest one, is the one which requires the second biggest time to complete, whereas Query7 which is slightly more complex than Query3 and has only one row as output, takes the less time.

Figures 4.21, 4.22, 4.23 show how the computation time varies for different instances of the same type of Query.

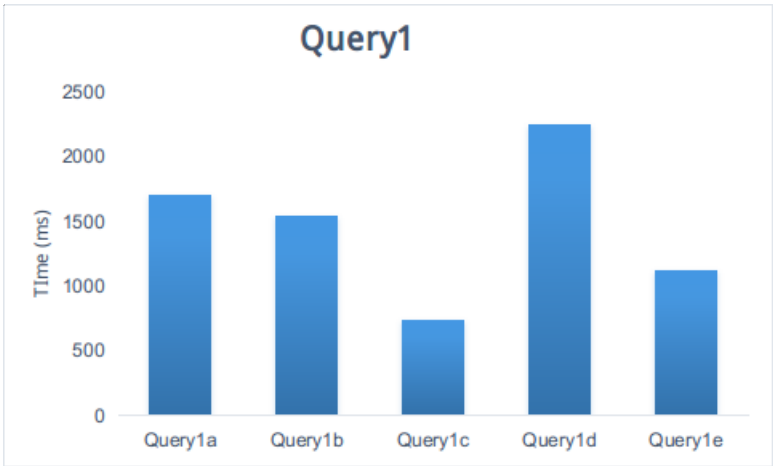


Figure 4.21: Query 1 - Openvirtuoso

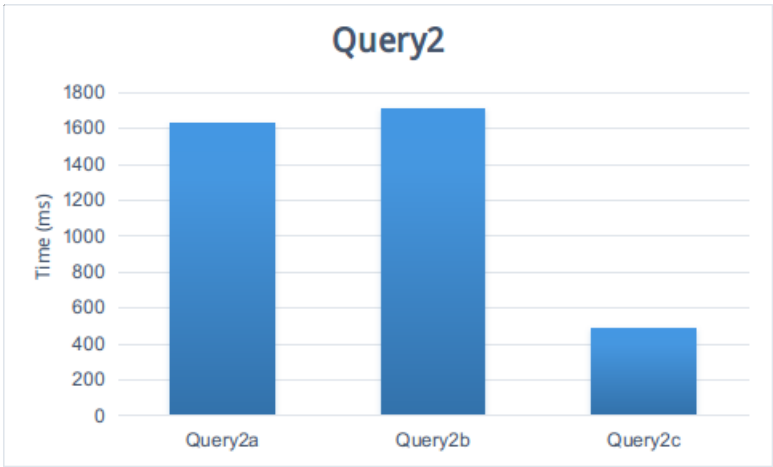


Figure 4.22: Query 2 - Openvirtuoso

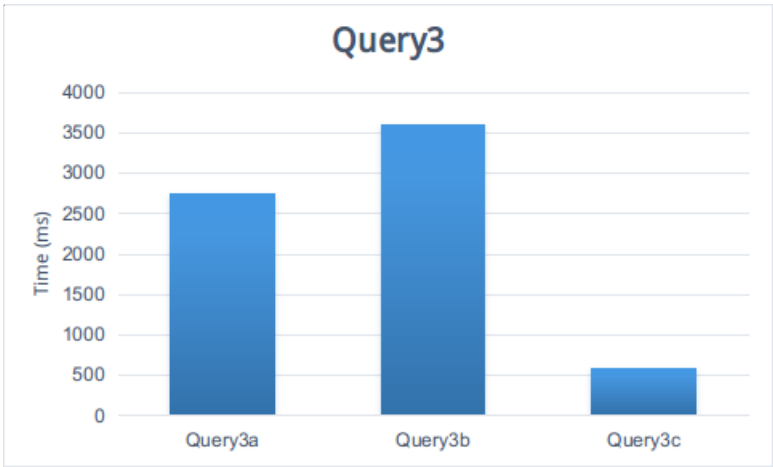


Figure 4.23: Query 3 - Openvirtuoso

Here we see that the differences in execution time are more alike with those in Giraph, and any variations seem to be due to the time needed to write the output in the cases many rows are returned as results.

4.2.7 Query Completeness and Soundness

Last but not least we compared the results we took from Giraph and Openvirtuoso, so as to verify that our application returned all the results and right ones. The verification was done in both datasets.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we first provided an overview of the rapid growth of data nowadays and expressed the need to store and exploit them in order to extract meaningful information and impove various aspects of our lives. We presented computing models, like Cloud Computing and Distributed Computing, since currently most systems are based on them in order to achieve big data processing in an efficient way. We continued by providing information about the most popular tools used for distributed processing, like Hadoop and Giraph, as well as the Programming models they implement.

Then, we presented our own contribution, a simple query processor, whose goal is to exploit the distributed environment in order to answer various queries in an efficient manner. We presented different kind of queries over DBpedia datasets and introduced efficient methologies in order to resolve them. We described the way we implemented our algorithms, in a 'Think Like A Vertex' manner and conducted experiments in order to study the performance and scalability of our system in an Openstack cluster consisting of 8 nodes. We also verified the validity and completeness of our results, by comparing them with those an existing centralized system would return.

From the results of our experiments, we came over to the conclusion that it is feasible to use distributed algorithms in order to provide complete answers for different types of SPARQL queries over Linked Data. At the same time, we saw that the query execution time improves as the number of the computer resources increases, which means that we can achieve much better performance in the query response time, by simply using more computer resources, something that it is not possible with centralized approaches, which can exploit only one single machine. In addition, we saw that different type of queries, different datasets as input, as well as different instances of the same types of queries can lead to different computation times. Last, we made a small comparison with Openvirtuoso, a well-known centralized system, which showed that Giraph outperforms Openvirtuoso in loading the graph datasets, but it is much less efficient in terms of computation, something that requires further investigation. We assume though that this extra time required by Giraph in order to provide complete answers to the queries, is justified by the overhead of coordinating many nodes, when we use small datasets as input and we would be able to see its real power with the use of much bigger datasets.

5.2 Future Work

There are many extensions which could be done in order to improve our system by providing better performance and functionality. New functionalities can be added, which would let us execute more comlex queries, like ones that assume hierarchy information and logical inference, in an automatic way. On the other hand, improvements can be made in the way of calculating the query result. Instead of using many different algorithms, each of them suitable only for a specific type of query, it would

be preferable to have a more generic approach, where our algorithm would be capable to read the SPARQL query, analyse it, find its type and its parameters, and would proceed to the query execution without the need to ask any more information from the user. Similarly, an appropriate user-friendly API could be created, which would let user to give the desired query without restricting him to choose a specific one.

Regarding the performance of our system, it would be useful to examine more configuration options either in Giraph or Hadoop. Furthermore, we could have different results either with the use of different kind of pre-processing (i.e. grouping by subject or predicate instead of object) or by trying different ways of distributing the graph partitions among the workers, or by exploting the capability of graph mutation in order to reduce the number of supersteps required. Research can be also made in the field of scalability by using even bigger datasets with more computing resources. Last but not least, we can enrich our study, by examining the performance of other distributed graph systems as well and compare it with that of Giraph over the same datasets.

Bibliography

[1] "Mapreduce dataflow." http://www.mikepluta.com/hadoop-v1-architecture-overview.

[2] "Apache hadoop yarn." http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[3] "Bsp wikipedia." https://en.wikipedia.org/wiki/Bulk_synchronous_parallel.

[4] A. Ching, "Dynamic graph/iterative computation on apache giraph," Giraph at Hadoop Summit, Juny 6 2014.

[5] "Giraph execution example." http://www.ibm.com/developerworks/library/os-giraph, June 10 2013.

[6] "The linking open data cloud diagram." http://lod-cloud.net, 2014.

[7] T. White, Hadoop: The Definite Guide. O'Reilly Media, Inc., 4th ed., April 1 2015.

[8] P. Mell and T. Grance, "Special publication 800-145: The nist definition of cloud computing. recommendations of the national institute of standards and technology," tech. rep., U.S. Department of Commerce, Gaithersburg, MD 20899-8930, September 2011.

[9] A. M. Fox A., Griffith R. and et al., "Above the clouds: A berkeley view of cloud computing," Tech. Rep. No. UCB/EECS-2009-28, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, February 10 2009.

[10] "Apache spark." http://spark.apache.org/.

[11] "Apache storm." http://storm.apache.org.

[12] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," In SIGMOD Conference, pp. 135–136, June 6-11 2010.

[13] "Google bigquery." https://developers.google.com/bigquery.

[14] "Apache giraph." http://giraph.apache.org.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," PVLDB, vol. 5, no. 8, pp. 716–727, 2012.

[16] "Apache hadoop." http://hadoop.apache.org.

[17] "Mapreduce wikipedia." https://en.wikipedia.org/wiki/MapReduce.

[18] "Jobtracker wiki." https://wiki.apache.org/hadoop/JobTracker.

[19] "Tasktracker wiki." https://wiki.apache.org/hadoop/JobTracker.

[20] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," In HPDC, pp. 810–818, 2010.

[21] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "The HaLoop approach to large-scale iterative data analysis," The VLDB Journal, vol. 21, no. 2, pp. 169–190, 2012.

[22] P. Stutz, A. Bernstein, and W. Cohen, "Signal/collect: Graph algorithms for the (semantic) web," In International Semantic Web Conference (ISWC), pp. 764–780, 2010.

[23] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," 6th Biennial Conference on Innovative Data Systems Research (CIDR), pp. 716–727, 2013.

[24] "Linked data wikipedia." https://en.wikipedia.org/wiki/Linked_data.

[25] "Linked data: Evolving the web into a global data space." http://linkeddatabook.com/editions/1.0/.

[26] "Tim berners-lee. linked data - design issues." https://www.w3.org/DesignIssues/LinkedData.html, 2006.

[27] F. Manola and E. Miller, "Rdf primer," W3C, https://www.w3.org/TR/rdf-primer, February 2004.

[28] B. Adida and M. Birbeck, "Rdfa primer- bridging the human and data webs - w3c recommendation," http://www.w3.org/TR/xhtml-rdfa-primer/, 2008.

[29] D. Beckett and T. Berners-Lee, "Turtle - terse rdf triple language," http://www.w3.org/TeamSubmission/turtle/, 2008.

[30] "Rdf test cases." https://www.w3.org/TR/rdf-testcases/#ntriples, 2004.

[31] "Sparql wikipedia." https://en.wikipedia.org/wiki/SPARQL.

[32] "Dbpedia." https://www.w3.org/standards/semanticweb/data.

[33] C. Bizer, J. Lehman, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellman, "Dbpedia - a crystallization point for the web of data," Submitted to Elsevier, May 25 2009.

[34] "Lubm." http://swat.cse.lehigh.edu/projects/lubm.

[35] "Virtuoso open-source." https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main.

[36] "Openlink virtuoso." http://techbus.safaribooksonline.com/book/databases/business-intelligence/9781484210499/setting-up-your-own-sparql-endpoint/sec23_html_3?uicode=nokia.