



Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Πληροφορικής και Τεχνολογίας Υπολογιστών

Προσαρμοστικοί Αλγόριθμοι Εξισορρόπησης Φόρτου σε Κατανεμημένα Περιβάλλοντα (Δίκτυα Ομοτίμων και Υπολογιστικά Νέφη)

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

ΙΩΑΝΝΗΣ Η. ΚΩΝΣΤΑΝΤΙΝΟΥ

Αθήνα, Ιούνιος 2011



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

Προσαρμοστικοί Αλγόριθμοι Εξισορρόπησης Φόρτου σε Κατανεμημένα
Περιβάλλοντα (Δίκτυα Ομοτίμων και Υπολογιστικά Νέφη)

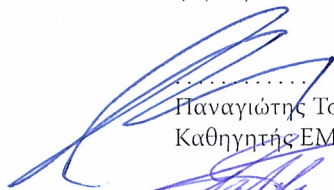
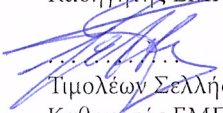
ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

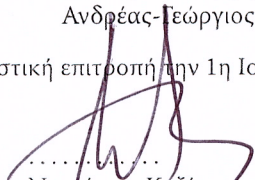
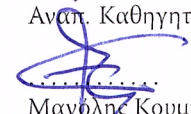
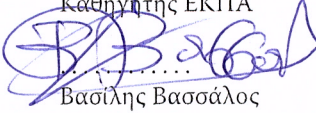
ΙΩΑΝΝΗΣ Η. ΚΩΝΣΤΑΝΤΙΝΟΥ

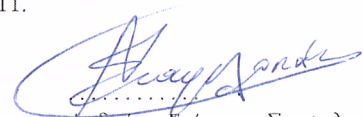
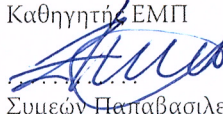
Συμβουλευτική Επιτροπή:

Παναγιώτης Τσανάκας
Νεκτάριος Κοζύρης
Ανδρέας-Γεώργιος Σταφυλοπάτης

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 1η Ιουνίου 2011.


.....
Παναγιώτης Τσανάκας
Καθηγητής ΕΜΠ

.....
Τιμολέων Σελλής
Καθηγητής ΕΜΠ


.....
Νεκτάριος Κοζύρης
Αναπ. Καθηγητής ΕΜΠ

.....
Μανόλης Κουμπάρκης
Καθηγητής ΕΚΠΑ

.....
Βασίλης Βασάλος
Αναπ. Καθηγητής ΟΠΑ


.....
Ανδρέας-Γεώργιος Σταφυλοπάτης
Καθηγητής ΕΜΠ

.....
Συμεών Παπαβασιλείου
Αναπ. Καθηγητής ΕΜΠ

Αθήνα, Ιούνιος 2011

.....

ΙΩΑΝΝΗΣ Η. ΚΩΝΣΤΑΝΤΙΝΟΥ

Διδάκτωρ Εθνικού Μετσόβιου Πολυτεχνείου

Copyright © ΙΩΑΝΝΗΣ Η. ΚΩΝΣΤΑΝΤΙΝΟΥ, 2011
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.

*Στους γονείς μου
Ηλία και Ολυμπία
Στα αδέρφια μου
Κική, Κατερίνα και Νίκο
και στο Μαράκι μου*

Contents

1 Ευχαριστίες	1
2 Εκτεταμένη Περίληψη	3
2.1 Εξισορρόπηση φόρτου σε κατανεμημένες δομές δεδομένων που υποστηρίζουν την δρομολόγηση ερωτημάτων εύρους τιμών	8
2.1.1 Συμβολή	11
2.2 Κατανεμημένη αποθήκευση και δεικτοδότηση διαφορετικού τύπου δεδομένων μεγάλου όγκου	12
2.2.1 Συμβολή	13
3 Introduction	17
3.1 Load Balancing In Distributed Range-Partitioned Data Structures	21
3.2 A Distributed Framework for Indexing Webscale Datasets	24
4 Load Balancing In Distributed Range-Partitioned Data Structures	29
4.1 Notation and Problem Setup	30
4.1.1 Notation	30
4.1.2 Problem statement – Metrics	31
4.2 Load Balancing Using Neighbor Item Exchange and Node Migration	32
4.2.1 Neighbor Item Exchange	35
4.2.2 Node Migration	35
4.2.3 Analysis	35

4.3	The NIXMIG Algorithm	40
4.3.1	Algorithm	42
4.3.2	Enhancements	46
4.3.3	Theoretical Analysis	47
4.4	Case study: NIXMIG over Skip Graphs	49
4.5	Experimental Results	51
4.5.1	Measuring the effectiveness of <i>NIXMIG</i>	53
4.5.2	<i>NIXMIG</i> scalability	59
4.5.3	<i>NIXMIG</i> performance under dynamic workload	59
4.5.4	Smart underloaded node location mechanism	61
4.5.5	Smart remote node placement mechanism	61
4.5.6	<i>NIXMIG</i> in realistic workloads.	63
4.6	Related Work	64
4.6.1	Data Replication	64
4.6.2	Data Migration	65
4.7	Conclusions	74
5	A Distributed Framework for Indexing and Serving Large and Diverse Datasets	75
5.1	Architecture	75
5.2	Experimental Results	80
5.2.1	Content table creation	83
5.2.2	Index table creation	83
5.2.3	System performance under query load	85
5.3	Related Work	86
5.3.1	MapReduce based data analysis frameworks.	87
5.3.2	Distributed indexing frameworks based on NoSQL and MapReduce combinations.	89
5.4	Discussion	91
6	Conclusions and Future Work	93
6.1	Conclusions	93
6.2	Future Directions	95
6.2.1	Adaptive replication of distributed range partitioned data structures. . .	95
6.2.2	NoSQL adaptive replication of popular regions.	96
7	Publications	99
	Appendix	123

A	On the Elasticity of NoSQL Databases over Cloud Management Platforms	123
A.1	Introduction	123
A.2	Architecture	125
A.3	NoSQL Overview	127
A.3.1	HBase	127
A.3.2	Cassandra	127
A.3.3	Riak	128
A.4	Experimental Results	128
A.4.1	Metrics affected during stress-testing	129
A.4.2	Cost of adding and removing nodes	131
A.4.3	Cluster resize performance measurements	132
A.5	Related Work	137
A.6	Discussion	138
A.6.1	Monitoring module	138
A.6.2	Database elasticity	139
A.7	Conclusions	141

List of Figures

2.1	Μια αναλογική απεικόνιση της αύξησης των δεδομένων για την τρέχουσα δεκαετία.	4
2.2	Ο κατατεμαχισμός εξισορροπεί μια άνιση κατανομή αλλά καταστρέφει την τοπικότητα του περιεχομένου.	9
2.3	Παράδειγμα Μετανάστευσης Κόμβων. Ο κόμβος D τοποθετείται μεταξύ των κόμβων A και B και μοιράζεται μέρος του φορτίου τους.	10
2.4	Παράδειγμα Ανταλλαγής Αντικειμένων μεταξύ Γειτόνων. Διαδοχικές ανταλλαγές κλειδιών μεταξύ των ζευγών (A,B), (B,C) και (C,D) τελικά δημιουργούν μια εξισορροπημένη κατανομή.	10
2.5	Επισκόπηση μιας κατανεμημένης πλατφόρμας δημιουργίας ευρετηρίου.	13
3.1	A proportional representation of data growth estimation for the current decade.	18
3.2	Hashing may balance a skewed distribution, but destroys content locality.	22
3.3	Node Migration example. Node D is placed between nodes A and B and shares part of their load.	22
3.4	Neighbor Item Exchange example. Iterative key exchanges between (A,B), (B,C) and (C,D) node pairs produce a balanced load.	23
3.5	Overview of a distributed indexing platform.	25
4.1	Overlay maintenance communication cost for migration of node N_m next to node N_p	33
4.2	Key exchange between two nodes	34
4.3	Balancing effect of a chain of <i>NIX</i> operations	36

4.4	Balancing effect of a single <i>MIG</i> operation	36
4.5	Worst case of initial setup and converged balanced network for the <i>NIX</i> case	37
4.6	Worst case of initial setup and converged balanced network for the <i>MIG</i> case	38
4.7	A successful <i>NIXMIG</i> operation	41
4.8	Load exchange between splitter N_{i-1} and helper N_i , splitter node in critical condition	44
4.9	Load exchange between splitter N_{i-1} and helper N_i , splitter node not in critical condition	44
4.10	Smart remote node placement: A scans its range and decides to place B on its forward direction, minimizing the number of transferred items $ r_2 - r_b $	45
4.11	Pointers before and after boundary exchange	50
4.12	Pulse workloads of various width. The width is used to create more or less skewed distributions	52
4.13	Zipfian workloads of various θ . The θ regulates the distribution's skewness	52
4.14	Completion time, exchanged messages and items and <i>MIG</i> to <i>NIX</i> ratio of <i>NIXMIG</i> , plain <i>NIX</i> , plain <i>MIG</i> and <i>Item Balancing</i> for various pulse widths.	54
4.15	Completion time, exchanged messages and items and <i>MIG</i> to <i>NIX</i> ratio of <i>NIXMIG</i> and <i>Item Balancing</i> for various zipfian θ values.	55
4.16	Load snapshot at t=800 sec for a 3% pulse	56
4.17	Number of exchanged messages during time for a 3% pulse	56
4.18	Load snapshot at t=800 sec for a zipfian of $\theta = 4.5$	56
4.19	Number of exchanged messages during time for a zipfian of $\theta = 4.5$	56
4.20	The Gini variation for the dynamic setting	57
4.21	Number of overloaded peers over time, dynamic setting	57
4.22	Number of exchanged messages over time, dynamic setting	57
4.23	Number of exchanged keys over time, dynamic setting	57
4.24	Load distribution as it progresses in time: Snapshot taken at times t={800, 850, 1000, 1181} sec	58
4.25	A number of popular areas of 10% width and 50% width in the ID space.	62
4.26	Load distribution of the AOL dataset with a query prefix size of 4 characters.	63
5.1	Combining MapReduce with NoSQL to create a fully distributed indexing system.	76
5.2	System Architecture	77
5.3	Indexing rules. Record boundaries split input into distinct entities and attribute types define record regions to index.	78
5.4	Index size for various datasets	82
5.5	Index time for various datasets	82

5.6	Response time of queries vs system load in queries/s, original data size and number of indexed attributes.	83
5.7	Overview of a simple MapReduce operation. Input chunks are fed to a number of mappers, intermediate output is sorted and assigned to a usually smaller number of reducers which produce the final output.	87
6.1	NIXMIG balancing. Nodes N_3 , N_4 and N_5 leave their place and take a part of the overloaded area previously served by N_1 and N_2	96
6.2	Balancing through replication. Nodes N_3, N_4 and N_5 leave their place and, together with N_1 , handle requests for the overloaded area of the first 8 items of the ID space.	97
A.1	CPU-RAM usage and mean query throughput-latency for various λ query rates of the UNIFORM_READ workload for an HBase-Cassandra Cluster of 8 nodes.	129
A.2	Query latency, query throughput and CPU usage per time for an HBase cluster of 8 nodes after adding 8 and 16 more nodes for the UNIFORM_READ and UNIFORM_UPDATE workload with a query rate of $\lambda = 180$ Kreqs/sec	134
A.3	Query latency, query throughput and CPU usage per time for a Cassandra cluster of 8 nodes after adding 8 and 16 more nodes for the UNIFORM_READ and UNIFORM_UPDATE workload with a query rate of $\lambda = 180$ Kreqs/sec	136

List of Tables

4.1	<i>NIXMIG</i> variables	43
4.2	Exchanged items and messages and completion time for various <i>overThres</i> and <i>ttl</i> values	53
4.3	Ratio of exchanged messages, completion time and transferred items for various network sizes compared to a 500 node setting.	59
4.4	Exchanged items and completion time for the dynamic setting for various trigger times and new pulse positions	61
4.5	Number of probing messages and success rate of <i>SmartUNL</i> vs <i>NaiveUNL</i> for various pulse widths	61
4.6	Ratio of transferred items using <i>SmartRNP</i> vs <i>RandomRNP</i> and <i>AdversarialRNP</i> for various “hot” range percentages	62
4.7	Completion time, number of exchanged messages and <i>MIG</i> to <i>NIX</i> ratio of <i>NIXMIG</i> for various prefix lengths.	63
5.1	Content table example. Row key is MD5Hash of the content, and cell content is the record data.	78
5.2	Index table example. Row key is the index term followed by the attribute type and row content is the list of the MD5Hashes of the documents that contain this keyword in this attribute.	79
5.3	Index size and creation time for different numbers of attribute types (5GB HTML).	81
5.4	DB Index creation time vs number of nodes.	81
5.5	Content table creation time for various dataset sizes and types	83

A.1	CPU-RAM usage and mean query throughput-latency under different workload types with a fixed query rate of $\lambda = 180$ Kreqs/sec for an HBase-Cassandra Cluster of 8 nodes.	131
A.2	Completion time, total moved data, final average query throughput and latency for a 8+8 node cluster resize operation in HBase, Cassandra and Riak with and without data rebalancing	132

List of abbreviations

ACID Atomicity, Consistency, Isolation, Durability

AOL America Online

AWS Amazon Web Services

CDN Content Delivery Network

DBMS Database Management System

DHT Distributed Hash Table

EC2 Elastic Compute Cloud

HDFS Hadoop Distributed File System

MIG Node Migration

NIX Neighbor Item Exchange

NoSQL Not only SQL

OLAP Online Analytical Processing

P2P Peer to Peer

PBS Portable Batch System

S3 Simple Storage Service

SHA Secure Hash Algorithm

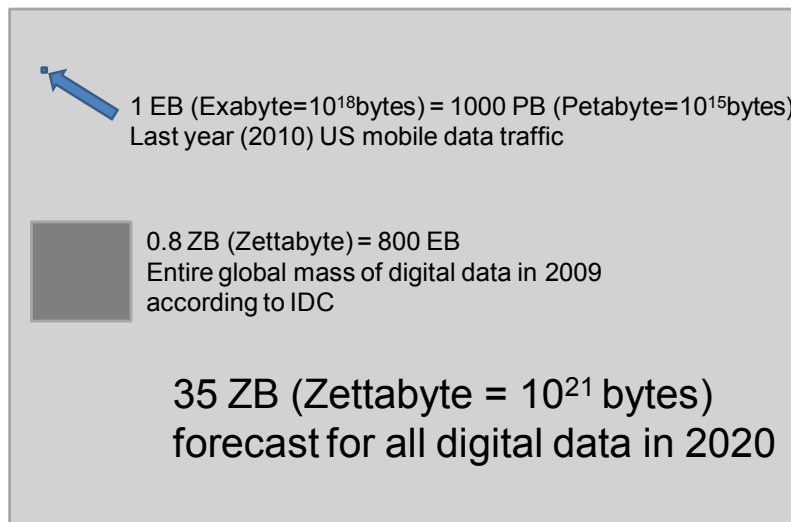
SME Small to Medium sized Enterprises

Ευχαριστίες

Καθώς η διδακτορική μου διατριβή έχει φτάσει σχεδόν στο τέλος της, αισθάνομαι την ανάγκη να ευχαριστήσω όλους εκείνους που στάθηκαν δίπλα μου σε όλη την διάρκεια αυτού του ταξιδιού. Καταρχάς θα ήθελα να ευχαριστήσω μέσα από την καρδιά μου τον Καθηγητή μου Νεκτάριο Κοζύρη, ο οποίος φρόντιζε έτσι ώστε συνέχεια να έχω πρόσβαση στους απαραίτητους πόρους για την έρευνά μου. Η εμπιστοσύνη και η ελευθερία κινήσεων που μου έδωσε ο Άρης κατά την ενασχόλησή μου με ερευνητικά θέματα οδήγησαν στην βελτίωση της κριτικής μου ικανότητας, με βοήθησαν να παίρνω πρωτοβουλίες και μου άνοιξαν νέους δρόμους. Επίσης, θα ήθελα να ευχαριστήσω όλα τα μέλη της συμβουλευτικής επιτροπής για τα πολύτιμα σχόλια τους. Επιπλέον, θα ήθελα να ευχαριστήσω τον συνεργάτη μου Δημήτρη Τσουμάκο για τις εποικοδομητικές συζητήσεις μας αλλά και τις συγκρούσεις μας που με βοήθησαν αφενός να εστιάσω την ερευνητική μου πορεία και αφετέρου να παρουσιάζω τα αποτελέσματά μου με τρόπο κατανοητό, επιστημονικό και αδιαφιλονίκητο. Πέρα από τον Δημήτρη, θα ήθελα ευχαριστήσω και όλους τους συνεργάτες μου στο εργαστήριο τόσο για την βοήθειά τους σε ερευνητικά θέματα όσο και για τις ευχάριστες στιγμές που έχουμε περάσει μαζί. Όλοι μου οι άλλοι φίλοι αξίζουν ένα μεγάλο ευχαριστώ, και ένα μεγαλύτερο αυτός που με στήριζε σε πραγματικά “ζόρικες” καταστάσεις. Τέλος, ευχαριστώ από τα βάθη της ψυχής μου την οικογένειά μου και το Μαράκι μου, οι οποίοι στάθηκαν δίπλα μου όλον αυτό τον καιρό και στους οποίους αφιερώνω την διατριβή αυτή.

Εκτεταμένη Περίληψη

Τον τελευταίο καιρό παρατηρούμε μια τεράστια αύξηση του όγκου της ψηφιακής πληροφορίας που δημιουργείται και καταναλώνεται παγκοσμίως. Η εποχή μας χαρακτηρίζεται από μια τεράστια έκρηξη δεδομένων. Σύμφωνα με τον Διευθύνων Σύμβουλο της Google [Kir10], 5 hexabytes δεδομένων που δημιουργήθηκαν από την ανακάλυψη των υπολογιστών μέχρι το 2003 τώρα πλέον παράγονται κάθε δυο μέρες. Η ετήσια έρευνα της IDC για το “Ψηφιακό Σύμπαν” (Digital Universe) που κυκλοφόρησε τον Μάιο του 2010 [GR10] αναφέρει ότι το 2009, παρά την παγκόσμια ύφεση, το σύνολο των ψηφιακών δεδομένων που δημιουργήθηκαν αυξήθηκε κατά 62% σε σχεδόν 800.000 petabytes. Το 2010 έφτασε τα 1,2 εκατ. petabytes, ή 1,2 zettabytes, και μέχρι το τέλος της δεκαετίας θα είναι 44 φορές το μέγεθος του 2009. Στο σχήμα 2.1 μπορούμε να παρατηρήσουμε μια σχηματική απεικόνιση αυτής της αύξησης. Η μικρή κουκκίδα αντιπροσωπεύει 1 Exabyte δεδομένων, το οποίο αντιστοιχεί στα δεδομένα που μεταφέρθηκαν όλο το 2010 από το δίκτυο κινητής τηλεφωνίας των ΗΠΑ, ενώ ολόκληρο το γκρίζο κουτί αντιπροσωπεύει 35 Zettabytes, η εκτίμηση για το σύνολο των ψηφιακών δεδομένων το 2020. Τα δεδομένα αυτά περιλαμβάνουν αρχεία καταγραφής από ιστοσελίδες, ρεύματα επιλογών χρηστών (click streams) από δικτυακούς τόπους κοινωνικής δικτύωσης όπως το Facebook [Fac11] και το Twitter [twi11] και μεγάλης κλίμακας e-commerce ιστοσελίδες όπως το Amazon [Ama11e] και το eBay [eBa11], ανεπεξέργαστα δεδομένα που παράγονται από ετικέτες RFID και δίκτυα αισθητήρων, αρχεία καταγραφής τηλεφωνικών κλήσεων, ιατρικά αρχεία, οπτικοακουστικά αρχεία, χρηματοοικονομικές συναλλαγές και δεδομένα από επιστήμες όπως Αστρολογία, Μετεωρολογία, Γονιδιωματική, Βιολογία, κλπ. Για παράδειγμα, ο μεγάλος Επιταχυντής Αδρονίων του CERN θα παράγει



Σχήμα 2.1: Μια αναλογική απεικόνιση της αύξησης των δεδομένων για την τρέχουσα δεκαετία.

περίπου 15PB δεδομένων ετησίως [CER08] και το Facebook δημιουργεί καθημερινά γύρω στα 20TB συμπιεσμένων δεδομένων.

Το πρόβλημα της διαχείρισης “Μεγάλων Δεδομένων” [Jac09] αναδεικνύει την αναποτελεσματικότητα των κεντροποιημένων αρχιτεκτονικών, οι οποίες αρχίζουν πλέον να θεωρούνται παρωχημένες. Είναι αλήθεια ότι ο νόμος του Moore δεν μπορεί να ισχύει για πάντα: θεμελιώδη φυσικά εμπόδια απαγορεύουν στους κατασκευαστές να παράγουν όλο και πιο γρήγορους επεξεργαστές [Dub05]. Ως εκ τούτου, η εκθετική αύξηση των δεδομένων δεν μπορεί να αντιμετωπιστεί με την εκθετική αύξηση της υπολογιστικής ισχύος. Από την άλλη πλευρά, καταναμημένα συστήματα διαχείρισης δεδομένων που αποτελούνται από μεγάλο αριθμό “shared nothing” βασικών (commodity) υπολογιστικών και αποθηκευτικών μονάδων είναι σε θέση να παρέχουν τους απαιτούμενους πόρους με κλιμακώσιμο και αποδοτικό τρόπο.

Αυτά τα κέντρα δεδομένων, τα οποία αποτελούνται από μεγάλο αριθμό υπολογιστικών και αποθηκευτικών μονάδων που διασυνδέονται με απλό εξοπλισμό δικτύωσης, έχουν αντικαταστήσει ακριβούς υπερυπολογιστές και πλέον είναι η de-facto προσέγγιση από τη συντριπτική πλειοψηφία των εταιρειών λογισμικού, ιδιαίτερα εκείνων που ασχολούνται με εφαρμογές Διαδικτύου. Οι εφαρμογές αυτές χρησιμοποιούνται για την αξιοποίηση υπολογιστικών και άλλων πόρων του κέντρου δεδομένων με έναν ενιαίο και επεκτάσιμο τρόπο. Δεδομένου ότι επεξεργάζονται μεγάλο αριθμό ταυτόχρονων και λογικά ανεξάρτητων αιτημάτων χρηστών μαζί με τεράστιες ποσότητες ασυσχέτιστων δεδομένων, είναι σε θέση να εξισορροπήσουν αποτελεσματικά τον φόρτο μεταξύ των πόρων του κέντρου δεδομένων. Το επιτυχημένο πρωτοποριακό παράδειγμα της Google σε αυτόν τον τομέα ακολουθείται τώρα από κάθε μεγάλη υπηρεσία Internet που έχει μεγάλο όγκο κίνησης: Το YouTube [You11], το Twitter [twi11], το Skype [Sky11] και το

Facebook [Fac11] είναι ένα αντιπροσωπευτικό σύνολο υπηρεσιών των οποίων η επιτυχία και η αποτελεσματικότητα οφείλεται στη χρήση κατακεντρωμένων συστημάτων διαχείρισης δεδομένων.

Τα δίκτυα ομοτίμων (P2P) [Wik11c] είναι ένα μοντέλο υπολογισμού που παρουσιάζει καλές ιδιότητες όσον αφορά την κλιμάκωση και την αυτο-οργάνωση και εφαρμόζεται σε αρκετές εφαρμογές μεγάλης κλίμακας. Σε ένα δίκτυο P2P, οι συμμετέχοντες κόμβοι είναι ισότιμοι και δεν απαιτούν μια κεντρική αρχή συντονισμού. Το μοντέλο υπολογισμού P2P είναι το αντίθετο της αρχιτεκτονικής client-server, στον οποίο οι εξυπηρετητές προσφέρουν και οι πελάτες καταναλώνουν. Στην περίπτωση των P2P, οι κόμβοι είναι ταυτόχρονα καταναλωτές και παραγωγοί των διαθέσιμων πόρων. Τα δίκτυα P2P έγιναν δημοφιλή στην αρχή της προηγούμενης δεκαετίας από τις εφαρμογές ανταλλαγής αρχείων, όπως το Napster, Gnutella [SGG03] και Bitorrent [QS04]. Η χρήση τους είναι συχνά αμφιλεγόμενη, αφού είναι γνωστά κυρίως για την διαδικτυακή ανταλλαγή και διάδοση πολλές φορές παράνομου περιεχομένου ή υλικού του οποίου τα πνευματικά δικαιώματα προστατεύονται, όπως λογισμικό και οπτικοακουστικό περιεχόμενο. Παρόλα αυτά, τα δίκτυα P2P έχουν πολύ καλές ιδιότητες όσον αφορά την κλιμάκωση και την ανοχή σε σφάλματα. Ως εκ τούτου, έχουν υιοθετηθεί σε πολλές εφαρμογές, όπως P2PTV [HFC⁺08], κατακεντρωμένες βάσεις δεδομένων [LM10, Vol09] on-line παιχνίδια για πολλούς παίκτες [KLXH04], Content Delivery Networks (CDNs) [BCMR02], δημοσίευση και διανομή του λογισμικού [DL09], κλπ.

Η τεχνολογία των υπολογιστικών νεφών [Wik11b], με την παροχή πόρων κατά απαίτηση μέσω ενός δικτύου υπολογιστών, αποτελεί ένα πιο πρόσφατο παράδειγμα που έχει επίσης την επιθυμητή ιδιότητα για κλιμάκωση. Ως εκ τούτου, λαμβάνει ολοένα μεγαλύτερη προσοχή τόσο από τη βιομηχανία όσο και από τον ακαδημαϊκό κόσμο. Κατά απαίτηση (On-demand) και σε διανεμητική βάση (pay-as-you-go, χρέωση ανάλογα με την χρήση) πρόσβαση σε υπολογιστικούς και αποθηκευτικούς πόρους που βρίσκονται σε απομακρυσμένα κέντρα δεδομένων είναι ένα πολύ ελκυστικό επιχειρηματικό μοντέλο, ιδίως για μικρομεσαίες επιχειρήσεις (MME) ή για νεοσύστατες επιχειρήσεις καινοτομίας (startups) που χρειάζονται γρήγορη, φθηνή και κλιμακώσιμη πρόσβαση σε εξοπλισμό και υποδομή λογισμικού. Οι υπηρεσίες Amazon Compute Cloud (EC2) [Ama11a] και Amazon Simple Storage Service (S3) [Ama11c] που εισήχθησαν το 2006 ως επεκτάσεις της διαδικτυακής πλατφόρμας υπηρεσιών AWS, ήταν οι πρώτες υπηρεσίες νέφους που καθιέρωσαν την Amazon ως τον κύριο παίκτη στην βιομηχανία του υπολογιστικού νέφους. Το επιτυχημένο παράδειγμα της Amazon ακολουθήθηκε από αρκετές ακόμα εταιρίες όπως η Rackspace [Rac11], η RightScale [Rig11], η GoGrid [GoG11], κλπ. Τόσο οι προμηθευτές και οι χρήστες ωφελούνται: οι πρώτοι πετυχαίνουν την πλήρη αξιοποίηση των υποδομών τους και δεύτεροι αποκτούν άμεση πρόσβαση σε πόρους χωρίς την επιβάρυνση της διαχείρισης ή το μεγάλο αρχικό κόστος κατασκευής ενός ιδιωτικού κέντρου δεδομένων. Χρήστες όπως μικρές και μεσαίου μεγέθους επιχειρήσεις (MME) ή startups που χρειάζονται μια γρήγορη, φθηνή και κλιμακώσιμη πρόσβαση σε υλικό και λογισμικό προσφεύγουν σε τεχνολογίες υπολογιστικών νεφών: Σύμφωνα με μια πρόσφατη έρευνα [Ori10], περισσότερες από τις μισές MME πρόκειται

να χρησιμοποιήσουν υπηρεσίες cloud computing το τρέχον έτος, σε σύγκριση με μόλις 22% το περασμένο έτος. Σύμφωνα με την Gartner [Gar11], το υπολογιστικό νέφος κατέλαβε την πρώτη θέση στη λίστα των top-10 στρατηγικών τεχνολογιών για το 2011.

Τόσο τα δίκτυα ομοτίμων όσο και τα υπολογιστικά νέφη αποτελούν δομικά υλικά για την δημιουργία αποτελεσματικών συστημάτων κατανεμημένης διαχείρισης δεδομένων. Και οι δύο τεχνολογίες χρησιμοποιούνται για την συνένωση μεγάλου αριθμού γεωγραφικά απομακρυσμένων υπολογιστών. Τα δίκτυα P2P συνήθως διασυνδέουν υπολογιστές χρηστών, ενώ οι πλατφόρμες υπολογιστικών νεφών συνενώνουν μεγάλες συστοιχίες υπολογιστών που βρίσκονται σε απομακρυσμένα κέντρα δεδομένων. Πέρα από αυτό, οι χρήσεις τους είναι γενικά αλληλένδετες: τα υπολογιστικά νέφη μπορεί να χρησιμοποιούν τεχνικές P2P για την αυτο-ρύθμιση των υποδομών τους για καλύτερη παροχή πόρων, ενώ τα δίκτυα P2P μπορούν να χρησιμοποιούν υπολογιστικούς πόρους από το cloud για αποτελεσματική κλιμάκωση σε περιπτώσεις υπερφόρτωσης. Για παράδειγμα, το OpenStack [Ope11], μια πρόσφατη πλατφόρμα διαχείρισης υπολογιστικών νεφών ανοιχτού λογισμικού που υποστηρίζεται από την Rackspace και τη NASA, σαν αποθηκευτικό υπόστρωμα χρησιμοποιεί το Swift, το οποίο είναι βασισμένο στο CEPH [WBM⁺06], ένα κατανεμημένο P2P αποθηκευτικό σύστημα. Επιπλέον, το NoSQL σύστημα Cassandra που χρησιμοποιείται από το Facebook [LM10], ένα τυπικό DataStore βασισμένο σε τεχνολογίες νέφους, βασίζεται σε μεγάλο βαθμό σε P2P αλγόριθμους για την δρομολόγηση ερωτημάτων, διαχείριση πόρων και ισοκατανομή φόρτου.

Παρ'όλα αυτά, η αποδοτικότητα των δικτύων ομοτίμων και των υπολογιστικών νεφών έχει το μειονέκτημα της πολυπλοκότητας κατά τον σχεδιασμό τους, καθώς πολυάριθμα ζητήματα προκύπτουν όταν τα δεδομένα και οι πόροι είναι διαμοιρασμένα σε μεγάλο αριθμό διαφορετικών αυτόνομων κόμβων. Θέματα όπως η εξισορρόπηση φόρτου (load balancing) [Cyb89], η συνέπεια (consistency) [HM90], ο συγχρονισμός (synchronization) [Lam78], η ανοχή σε σφάλματα (fault tolerance) [Dij74], η ιδιωτικότητα (privacy) [MKG07] και η ασφάλεια (security) [And08] είναι ένα μικρό αντιπροσωπευτικό δείγμα των τυπικών προβλημάτων που προκύπτουν και χρήζουν αντιμετώπισης σε ένα κατανεμημένο περιβάλλον. Προσαρμοστικά συστήματα που μπορούν να εγκατασταθούν και να λειτουργούν σε ένα αυθαίρετα μεγάλο αριθμό υπολογιστικών κόμβων με ελάχιστο διαχειριστικό κόστος χρησιμοποιούνται ευρέως σε τέτοιες περιπτώσεις.

Το αντικείμενο της παρούσας διατριβής είναι η μελέτη τεχνικών εξισορρόπησης φόρτου σε κατανεμημένα συστήματα διαχείρισης δεδομένων. Η άνιση κατανομή του φορτίου μπορεί να προκαλέσει προβλήματα ακόμα και σε υποδομές με αρκετούς πόρους. Τα πρόσφατα παραδείγματα διακοπής λειτουργίας της FourSquare [Hor11] και της Netflix [Net11] δείχνουν ότι ακόμη και αν μια εφαρμογή έχει αναπτυχθεί σχεδόν σε απεριόριστους υπολογιστικούς πόρους του νέφους, μπορεί να υποστεί σημαντικές υποβαθμίσεις σε καταστάσεις υψηλού φορτίου, εάν έχει ρυθμιστεί με στατικό τρόπο. Απρόβλεπτα και υψηλά φορτία που εμφανίζονται λόγω κάποιων γεγονότων που ενδιαφέρουν πολύ κόσμο, όπως π.χ., ο θάνατος του Michael Jackson [PCW09]

ή του Μπιν Λάντεν, ο σεισμός στην Ιαπωνία, κλπ, μπορούν να οδηγήσουν σε φαινόμενα flash-crowd [JKR02] κατά τα οποία η υπάρχουσα υπολογιστική υποδομή αποτυγχάνει να εξυπηρετήσει το μεγάλο όγκο των εισερχομένων αιτήσεων. Εκτός από αυτό, η ασύμμετρη πρόσβαση σε δεδομένα κατά την οποία ένα μικρό και δημοφιλές αυτών μπορεί να λάβει το μεγαλύτερο μέρος του εισερχόμενου φόρτου αποτελεί έναν ακόμη λόγο για μειωμένη απόδοση. Για παράδειγμα, εξυπηρετητές που διαμοιράζουν λογαριασμούς twitter και hashtags ή ιστολόγια δημοφιλών προσώπων μπορεί να παρουσιάζουν υψηλά φορτία, σε αντίθεση με άλλους εξυπηρετητές που φιλοξενούν λιγότερο δημοφιλείς λογαριασμούς. Η ανίχνευση ασύμμετρης πρόσβασης γίνεται ακόμα πιο δύσκολη σε καταναμημένα περιβάλλοντα [KBHR10] στα οποία ο συντονισμός των πόρων δεν είναι απλή υπόθεση.

Στο πρώτο μέρος της διατριβής παρουσιάζεται ο *NIXMIG*, ένας προσαρμοστικός επιγραμμικός (online) αλγόριθμος με σκοπό την εξισορρόπηση φόρτου εργασίας σε καταναμημένες δομές δεδομένων που υποστηρίζουν την δρομολόγηση ερωτημάτων εύρους τιμών (range queries). Ο προτεινόμενος αλγόριθμος αντιμετωπίζει προβλήματα άνισων κατανομών φόρτου εργασίας που προκύπτουν όταν οι εξυπηρετητές διαμοιράζουν αντικείμενα διαφορετικής δημοτικότητας. Ο αλγόριθμος υλοποιήθηκε και εφαρμόστηκε σε έναν skip-γράφο [AS07], ένα δομημένο δίκτυο ομότιμων κόμβων ικανό να δρομολογεί ερωτήματα εύρους τιμών. Ο αλγόριθμος συγκρίθηκε πειραματικά και θεωρητικά κάτω από διαφορετικές συνθήκες κίνησης με άλλους παρόμοιους αλγόριθμους. Η ανάλυση αποδεικνύει ότι ο προτεινόμενος αλγόριθμος είναι πιο γρήγορος και καταναλώνει λιγότερους δικτυακούς πόρους κατά την διαδικασία της εξισορρόπησης. Στο δεύτερο μέρος της εργασίας παρουσιάζεται μια καταναμημένη αρχιτεκτονική δεικτοδότησης, αποθήκευσης και επερώτησης διαφορετικού τύπου δεδομένων μεγάλου όγκου. Σχεδιάζουμε και υλοποιούμε ένα κλιμακώσιμο σύστημα στο οποίο τόσο τα περιεχόμενα όσο και τα ευρετήρια αυτών δημιουργούνται και διαμοιράζονται πλήρως παράλληλα. Το σύστημα έχει σχεδιαστεί για να αξιοποιεί τεχνολογίες υπολογιστικών νεφών (cloud computing). Ο φόρτος εργασίας κατά την δημιουργία και εξυπηρέτηση τόσο των περιεχομένων όσο και του ευρετηρίου εξισορροπείται μεταξύ των κόμβων του συστήματος συνδυάζοντας καινοτόμες τεχνικές παράλληλης ανάλυσης δεδομένων [DG08] με καταναμημένες, αραιές NoSQL βάσεις [CDG⁺08]. Το πρωτότυπο σύστημα δοκιμάστηκε κάτω από μεγάλο φόρτο ερωτημάτων και η μέση απόκρισή του κρατήθηκε σε τάξη millisecond.

Παρακάτω παρουσιάζουμε εν συντομία τα κίνητρα και τις συνεισφορές των δύο παραπάνω συστημάτων.

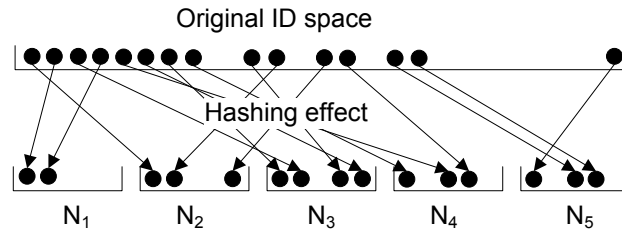
2.1 Εξισορρόπηση φόρτου σε κατανεμημένες δομές δεδομένων που υποστηρίζουν την δρομολόγηση ερωτημάτων εύρους τιμών

Οι κατακερματισμένες κατά εύρος (range partitioned) δομές δεδομένων, μια ειδική κατηγορία δικτύων ομοτίμων, χρησιμοποιούνται σε περιπτώσεις όπου σημασιολογικά “κοντινά” στοιχεία είναι ανάγκη να αποθηκεύονται με τέτοιο τρόπο έτσι ώστε να διατηρείται η σειρά τους. Η διατήρηση της τοπικότητας των δεδομένων σε τέτοιες περιπτώσεις επιτρέπει την αποτελεσματική δρομολόγηση ερωτημάτων εύρους, δηλαδή, ερωτημάτων για αντικείμενα των οποίων τα κλειδιά βρίσκονται εντός ενός συγκεκριμένου εύρους.

Τα ερωτήματα εύρους είναι πολύ σημαντικά σε πολλές εφαρμογές, όπως χωρικές [Ore86, KS03] και κατανεμημένες [VSCF09] βάσεις δεδομένων, δίκτυα αισθητήρων [LKGH03, YHP02, KVD⁺07], διαδικτυακά παιχνίδια πολλών παικτών [BPS06, KLXH04], εξυπηρετητές ιστού [LN01], αποθήκες δεδομένων [LLL00, HAMS97], κλπ.

Τα ακόλουθα υψηλότερου επιπέδου ερωτήματα μεταφράζονται σε ερωτήματα εύρους κατά τα οποία ένας αριθμός διαδοχικών αντικειμένων μπορεί να σαρωθεί με ένα μόνο αίτημα: Ερωτήματα GoogleMaps τύπου “βρες όλα τα νοσοκομεία μεταξύ Λ. Κηφισίας και Λεωφ. Μεσογείων”, ερωτήματα hotels.com όπως “βρες ξενοδοχεία στα Χανιά με διαθεσιμότητα μεταξύ 10 Ιουνίου 2011 και 13 του Ιουνίου 2011”, Condor [LLM88] ή PBS [Hen95] ερωτήματα όπως “βρες όλους τους διαθέσιμους κόμβους με μνήμη RAM μεταξύ 2GB και 8GB”, ερωτήσεις βάσεων δεδομένων όπως το “βρες όλους τους καθηγητές στην Ελλάδα, με μηνιαίο μισθό από 5K ευρώ και υψηλότερο” και ερωτήματα προθέματος όπως “βρες όλες τις λέξεις που αρχίζουν με τα γράμματα goo”.

Η χρησιμότητα των ερωτημάτων εύρους σε συνδυασμό με το πρόβλημα των “Μεγάλων Δεδομένων” οδήγησε στον σχεδιασμό και την υλοποίηση πολυάριθμων κατανεμημένων δομών ικανών για αποδοτική δρομολόγηση τέτοιου είδους ερωτημάτων. Οι Skip Graphs [AS07], τα Skip Nets [HJS⁺03], τα P-grids [APHS02], τα P-trees [CLGS04], το BATON [JOV05] και τα Prefix Hash Trees [RHRS04] είναι ένα αντιπροσωπευτικό δείγμα των εν λόγω δομών. Τα παραπάνω συστήματα οργανώνουν το ευρετήριο τους με τέτοιο τρόπο ώστε ερωτήματα για σημασιολογικά κοντινά αντικείμενα απαντώνται με την επικοινωνία ενός μικρού μέρους από ολόκληρο το δίκτυο εξυπηρετητών. Ένα βασικό πρόβλημα που αντιμετωπίζουν οι δομές αυτές είναι η λοξότητα (skew), όπου μόνο ένας μικρός αριθμός από “δημοφιλή” αντικείμενα ζητείται από το συνολικό πεδίο τιμών. Η λοξότητα έχει παρατηρηθεί σε μια ποικιλία εφαρμογών. Είναι γεγονός ότι οι περισσότερες διαδικτυακές εφαρμογές, συμπεριλαμβανομένων των εφαρμογών P2P, παρουσιάζουν μεγάλη ασυμμετρία όσον αφορά τον φόρτο εργασίας τους (πχ. [CKR⁺07, RFI02, SW02], κλπ.). Πέραν τούτου, τυχόν αστοχίες του υλικού μειώνουν επιπλέον την διαθεσιμότητα των διαφόρων πόρων. Κατά συνέπεια, οι διακομιστές υπερφορτώνονται και ο ρυθμός εξυπηρέτησης



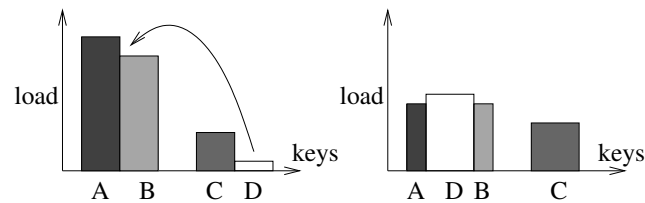
Σχήμα 2.2: Ο κατατεμαχισμός εξισορροπεί μια άνιση κατανομή αλλά καταστρέφει την τοπικότητα του περιεχομένου.

μειώνεται λόγω του υψηλού φόρτου εργασίας, κάτι που σε πολλές περιπτώσεις μπορεί να προκαλέσει από μόνο του αρνήσεις υπηρεσίας (denial of service) [JKR02].

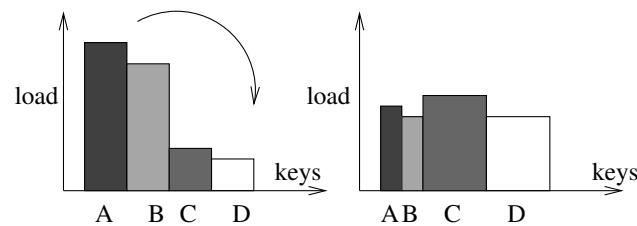
Σε αυτές τις περιπτώσεις, ένας τρόπος για να αντιμετωπιστούν υπερφορτωμένοι κόμβοι και να κατανεμηθεί ομοιόμορφα ο φόρτος εργασίας στους διαθέσιμους υπολογιστές είναι η εφαρμογή συναρτήσεων κατατεμαχισμού (hash functions) όπως η SHA-1 [Sta95] που μετατρέπουν ασύμμετρες κατανομές δεδομένων σε ομοιόμορφες. Η τεχνική αυτή χρησιμοποιείται από μια ειδική κατηγορία δικτύων ομοτίμων, τα DHTs (Κατανεμημένοι Πίνακες Κερματισμού), όπως το Chord [SMK⁺01], το Pastry [RD01], το CAN [RFH⁺01] και το Tapestry [ZHS⁺04]. Στο σχήμα 2.2 παρουσιάζουμε το αποτέλεσμα του κατατεμαχισμού σε μια άνιση κατανομή: Χωρίς τον κατατεμαχισμό, οι κόμβοι N_1 και N_2 θα ήταν υπεύθυνοι για μεγαλύτερο αριθμό αντικειμένων (πάνω μέρος του σχήματος 2.2). Ο κατατεμαχισμός εξασφαλίζει ότι, με μεγάλη πιθανότητα, κάθε κόμβος θα πάρει ίσο μερίδιο του συνολικού φορτίου, ανεξάρτητα της λοξότητας της κατανομής (κάτω μέρος του σχήματος 2.2). Παρόλα αυτά, το μειονέκτημα του μετασχηματισμού αυτού είναι η καταστροφή της τοπικότητας του περιεχομένου, καθώς συνεχόμενα αναγνωριστικά IDs επανατοποθετούνται σε εντελώς διαφορετικούς και απομακρυσμένους διακομιστές.

Ένας άλλος τρόπος για να αντιμετωπιστούν ανομοιόμορφες κατανομές είναι μέσω της αντιγραφής των δημοφιλών αντικειμένων σε πολλαπλούς κόμβους, δημιουργώντας έτσι επιπλέον αντίγραφα (replicas) τα οποία απορροφούν μέρος των ερωτημάτων. Παρόλα αυτά, ο περιορισμός που θέτει η τοπικότητα του περιεχομένου ελαχιστοποιεί τους υποψήφιους υπολογιστές, επιτρέποντας μόνο, πχ. κόμβους που απέχουν λίγα άλματα (hops) μακριά. Επιπλέον, η διαχείριση πολλαπλών αντιγράφων όχι μόνο προϋποθέτει την αλλαγή των αλγορίθμων δρομολόγησης του δικτύου κατά την εισαγωγή και αναζήτηση δεδομένων, αλλά απαιτεί και επιπλέον μέρμινα για τα θέματα συνέπειας που προκύπτουν κατά την ενημέρωση αντικειμένων (object updates).

Στις περιπτώσεις αυτές, μέθοδοι εξισορρόπησης φόρτου που ανακατανέμουν τα αντικείμενα μεταξύ των κόμβων είναι ιδιαίτερα χρήσιμες. Η δυναμική φύση και η μεγάλη κλίμακα των κατανεμημένων αυτών δομών, κατά την οποία είναι δύσκολο για έναν κόμβο να έχει μια συνολική εικόνα του φόρτου εργασίας στο δίκτυο, θέτει δύο βασικές προϋποθέσεις: επιγραμμική (online) λειτουργικότητα (η ιδιότητα να παίρνονται σωστές αποφάσεις κατά την εξισορρόπηση, έχοντας



Σχήμα 2.3: Παράδειγμα Μετανάστευσης Κόμβων. Ο κόμβος D τοποθετείται μεταξύ των κόμβων A και B και μοιράζεται μέρος του φορτίου τους.



Σχήμα 2.4: Παράδειγμα Ανταλλαγής Αντικειμένων μεταξύ Γειτόνων. Διαδοχικές ανταλλαγές κλειδιών μεταξύ των ζευγών (A,B) , (B,C) και (C,D) τελικά δημιουργούν μια εξισορροπημένη κατανομή.

μερική γνώση της κατανομής του φόρτου εργασίας) και προσαρμοστικότητα (η δυνατότητα για γρήγορη ανταπόκριση σε αλλαγές του φόρτου εργασίας).

Στην βιβλιογραφία υπάρχει μια ποικιλία μεθόδων με επίκεντρο την επίτευξη αποτελεσματικής εξισορρόπησης φόρτου είτε αυτές χρησιμοποιούν την έννοια των εικονικών κόμβων [DKK⁺01, RLS⁺03, SGL⁺06, GS05, ZH05, HLCH11, CT08, LCLC09] είτε όχι [KR06, GBGM04, AKK04, BAS04, VORT09, LCLC09, SX07, SX08, Ιου08]. Ωστόσο, μπορούν να κατηγοριοποιηθούν σε δύο γενικές στρατηγικές: *Μετανάστευση Κόμβων* (Node Migration) και *Ανταλλαγή Αντικειμένων μεταξύ Γειτόνων* (Neighbor Item Exchange). Οι τεχνικές αυτές αντιπροσωπεύουν δύο διαφορετικές προσεγγίσεις για το χειρισμό του προβλήματος: Η *Μετανάστευση Κόμβων* χρησιμοποιεί υποφορτωμένους κόμβους τους οποίους τοποθετεί σε υπερφορτωμένες περιοχές του δικτύου (βλέπε Σχήμα 2.3, όπου το ύψος των ράβδων δείχνει το φορτίο του κάθε κόμβου, ενώ το πλάτος τους αντανάκλα τον αριθμό των κλειδιών που σερβίρουν). Οι νεοαφιχθέντες κόμβοι καταλαμβάνουν μέρος του φορτίου των νέων γειτόνων τους. Από την άλλη πλευρά, η μέθοδος *Ανταλλαγής Αντικειμένων μεταξύ Γειτόνων* εξισορροπεί το φορτίο μέσω διαδοχικών ανταλλαγών αντικειμένων μεταξύ γειτονικών κόμβων (βλέπε Σχήμα 2.4). Η πλειονότητα των προτεινόμενων προσεγγίσεων χρησιμοποιεί μια μίξη των δυο αυτών μεθόδων με σκοπό την τελική εξισορρόπηση φορτίου μεταξύ κόμβων. Παρόλο που και οι δύο μέθοδοι τελικά πετυχαίνουν τον στόχο τους, η ταχύτητα και το κόστος τους διαφοροποιούνται σημαντικά, κάνοντας έναν αλγόριθμο που χρησιμοποιεί μόνο μία από τις δυο αυτές μεθόδους αναποτελεσματικό για όλες τις περιπτώσεις.

2.1.1 Συμβολή

Η συμβολή μας μπορεί να συνοψισθεί στα ακόλουθα:

- **Ανάλυση της Μετανάστευσης κόμβων (MIG) και Ανταλλαγής Αντικειμένων μέσω Γειτόνων (NIX):** Οι δύο διαφορετικές μεθοδολογίες που, όταν εφαρμοστούν διαδοχικά, εκτελούν εξισορρόπηση φορτίου σε κατακερματισμένες κατά εύρος δομές δεδομένων εντοπίζονται και προσδιορίζονται. Περιγράφουμε τους μηχανισμούς τους και μελετούμε θεωρητικά την επίδοσή τους ως προς το χρόνο ολοκλήρωσης και το κόστος επικοινωνίας. Ένα σημαντικό αποτέλεσμα του έργου μας είναι η παρατήρηση ότι, μόνο με απλές *Ανταλλαγές Αντικειμένων μεταξύ Γειτόνων* η εξισορρόπηση γίνεται ιδιαίτερα αργά και ο αριθμός των δεδομένων που ανταλλάσσονται μπορεί να είναι αρκετά μεγάλος, ενώ χρησιμοποιώντας μόνο *Μεταναστεύσεις Κόμβων* το κόστος της ενημέρωσης της δομής αυξάνεται σημαντικά.
- **Σχεδιασμός και θεωρητική ανάλυση του NIXMIG:** Με βάση αυτή την μελέτη παρουσιάζουμε τον NIXMIG, έναν υβριδικό αλγόριθμο που χρησιμοποιεί ταυτόχρονα *Ανταλλαγές Αντικειμένων μεταξύ Γειτόνων* και *Μεταναστεύσεις Κόμβων* προκειμένου να ελαχιστοποιηθούν οι υπερφορτωμένοι κόμβοι και να ισορροπηθεί η κατανομή φορτίου μεταξύ τους. Η μέθοδος αυτή καταφέρνει να προσαρμόσει τη χρήση των *Μεταναστεύσεων Κόμβων* με την *Ανταλλαγή Αντικειμένων μεταξύ Γειτόνων*: το φορτίο κινείται σαν κύμα από περισσότερο σε λιγότερο φορτωμένες περιοχές της δομής με προσαρμοστικό τρόπο, χρησιμοποιώντας την δική μας εκδοχή των διαδοχικών *Ανταλλαγών Αντικειμένων μεταξύ Γειτόνων*. Όταν εντοπίζουμε περιοχές με τοπικά μεγάλο φορτίο, τότε ενεργοποιούμε τις *Μεταναστεύσεις Κόμβων*. Επιπλέον, παρουσιάζουμε “έξυπνες” μεθόδους για τον εντοπισμό και την τοποθέτηση μακρινών κόμβων κατά την μετανάστευση, που σκοπό έχουν την περεταίρω μείωση της κατανάλωσης του εύρους ζώνης κατά την εφαρμογή του NIXMIG. Η Ύπαρξη σύγκλισης του αλγορίθμου, η ταχύτητα με την οποία η σύγκλιση αυτή επιτυγχάνεται καθώς και οι προϋποθέσεις που πρέπει να ισχύουν για να φτάσει το σύστημα σε σημείο ισορροπίας μελετήθηκαν θεωρητικά.
- **Υλοποίηση και πειραματική αποτίμηση του NIXMIG** Παρουσιάζουμε μια υλοποίηση σε skip γράφο [AS07] πάνω στην οποία εφαρμόζουμε και συγκρίνουμε τον υβριδικό NIXMIG με τις απλές *Μεταναστεύσεις Κόμβων*, τις *Ανταλλαγές Αντικειμένων μεταξύ Γειτόνων* και με έναν άλλο αλγόριθμο εξισορρόπησης φορτίου που προτάθηκε από τους Karger και Ruhl [KR06]. Μετράμε και συγκρίνουμε τη συμπεριφορά τους σε μια ποικιλία από δυναμικά και ανομοιογενή φορτία. Τα αποτελέσματά μας επικυρώνουν την προηγούμενη ανάλυση και δείχνουν ότι ο NIXMIG εξισορροπεί με χαμηλό κόστος (ανταλλάσει μόνο το ένα έκτο και το ένα τρίτο των μηνυμάτων και των αντικειμένων, αντιστοίχως, σε σύγκριση

με τον [KR06]) και με μεγάλη ταχύτητα (είναι τρεις φορές πιο γρήγορος από τον [KR06]), προσαρμόζεται στις μεταβολές του φόρτο εργασίας και είναι πολύ παραμετροποιήσιμος.

2.2 Κατανεμημένη αποθήκευση και δεικτοδότηση διαφορετικού τύπου δεδομένων μεγάλου όγκου

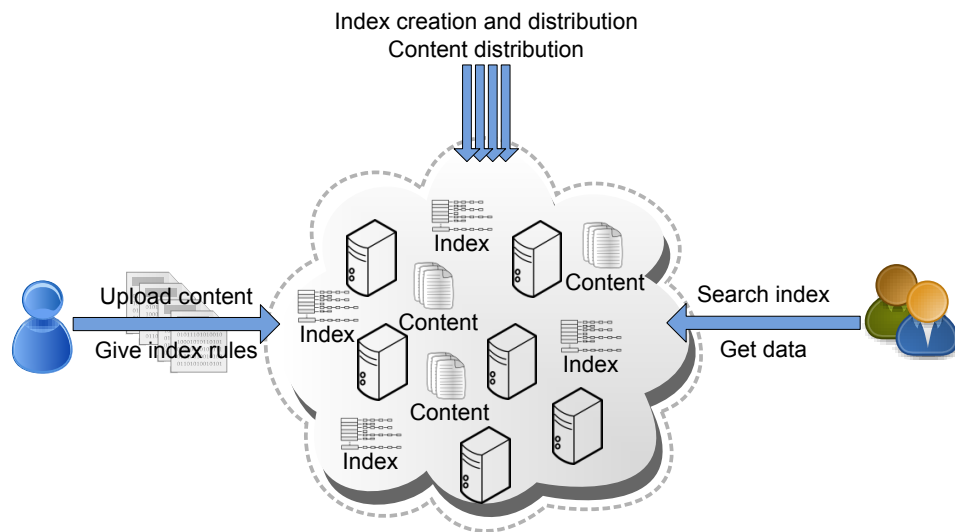
Τον τελευταίο καιρό, η ταυτόχρονη μείωση του κόστους του αποθηκευτικού χώρου και της πρόσβασης στο Internet επιτρέπει την ελεύθερη διάθεση στο κοινό μεγάλου όγκου δεδομένων. Δικτυακοί τόποι όπως το *Internet Archive* προσφέρουν πρόσβαση σε petabytes δεδομένων όπως ιστοσελίδες, βιβλία, κλπ. Ένα άλλο παράδειγμα είναι η Amazon η οποία προσφέρει δημόσια σύνολα δεδομένων (datasets) (σχεδόν) δωρεάν μέσω του αποθηκευτικού της νέφους¹. Η αποδοτική δεικτοδότηση και δημιουργία ευρετηρίου είναι πολύ σημαντική για να καταστεί δυνατή η χρησιμότητα των δεδομένων αυτών, αλλά αυτό είναι ένα πολύ δύσκολο έργο για τις εν λόγω ποσότητες δεδομένων: ακόμη και το Internet Archive, δεν επιτρέπει πλήρη αναζήτηση κειμένου (full text search) σε μια από τις πιο ενδιαφέρουσες υπηρεσίες της, το WayBackMachine² με ένα σύνολο δεδομένων της τάξης των 4,5 PB.

Αν και η δημιουργία ευρετηρίου είναι μια πολύ απαιτητική εργασία, έχει την βολική ιδιότητα της παραλληλοποίησης: ολόκληρη η εργασία μπορεί να χωριστεί σε πολλές μικρότερες υπο-εργασίες (για παράδειγμα, με το διαχωρισμό του συνόλου δεδομένων σε ισομεγέθη κομμάτια), οι οποίες μπορούν να εκτελεστούν ταυτόχρονα από διαφορετικούς υπολογιστές. Αν και ο συντονισμός των υπο-εργασιών μπορεί να γίνει και από παραδοσιακούς χρονοδρομολογητές εργασιών όπως ο Condor [LLM88] ή το PBS [Hen95], τα γενικής χρήσεως προγράμματα αυτά δεν έχουν σχεδιαστεί για υπολογισμούς μεγάλου όγκου δεδομένων. Προκειμένου να αντιμετωπιστεί το πρόβλημα αυτό, μια σειρά κατανεμημένων πλαισίων επεξεργασίας δεδομένων έχουν προταθεί για να αντικαταστήσουν αυτά τα προγράμματα σε περιπτώσεις όπου ο όγκος δεδομένων είναι πολύ μεγάλος. Τα συστήματα αυτά έχουν τα ίδια χαρακτηριστικά με τους προγόνους τους, όπως για παράδειγμα, την χρονο-δρομολόγηση εργασιών, την ανοχή σε σφάλματα και την εξισορρόπηση φόρτου, με επιπλέον λειτουργίες έχοντας κατά νου διαφορετικής κλίμακας όγκο δεδομένων: εφαρμόζουν με διάφανο τρόπο τεχνικές επεξεργασίας όπως η μετακίνηση των υπολογισμών κοντά στα δεδομένα, που επιτρέπουν την εύκολη ανάπτυξη και εκτέλεση μαζικά παράλληλων εφαρμογών. Το MapReduce της Google [DG08], το Scope της Microsoft [CJL⁺08], και το PIG της Yahoo [ORS⁺08] είναι ένα αντιπροσωπευτικό δείγμα τέτοιων πλαισίων.

Πέρα από τη δημιουργία ευρετηρίου, η εξυπηρέτηση και αποθήκευσή του είναι επίσης μια εργασία απαιτητική σε υπολογιστικούς πόρους, ειδικά όταν ο όγκος του ευρετηρίου ή ο ρυθμός αιτήσεων των χρηστών είναι πολύ μεγάλος, μια τυπική περίπτωση στις μηχανές αναζήτησης

¹<http://aws.amazon.com/publicdatasets>

²<http://www.archive.org/web/web.php>



Σχήμα 2.5: Επισκόπηση μιας κατανεμημένης πλατφόρμας δημιουργίας ευρετηρίου.

Ιστού. Το ευρετήριο της Google είχε ήδη 26 εκατομμύρια σελίδες το 1998 και μέχρι το 2008 είχε φτάσει το ένα τρισεκατομμύριο σελίδες [ΑΗ08]. Παρ'όλα αυτά, η εξυπηρέτηση του ευρετηρίου είναι επίσης μια εργασία που μπορεί να είναι κατανεμηθεί αποτελεσματικά. Για παράδειγμα, από την αρχή της λειτουργίας της Google, το ευρετήριο της κατανεμήθηκε μεταξύ πολλών κόμβων για την αποφυγή προβλημάτων κλιμάκωσης και για να υπάρχει καλύτερος έλεγχος των χαρακτηριστικών απόδοσης. Ένα τυπικό σενάριο ενός κατανεμημένου συστήματος δημιουργίας ευρετηρίου φαίνεται στο σχήμα 2.5: Οι διαχειριστές ανεβάζουν το περιεχόμενο μαζί με οδηγίες για την δημιουργία του ευρετηρίου. Το περιεχόμενο επεξεργάζεται και εξάγεται το ευρετήριο. Τέλος, οι χρήστες χρησιμοποιούν αυτό το ευρετήριο για την αναζήτηση και ανάκτηση δεδομένων. Πέρα από το ευρετήριο της Google, δεδομένα από πολλές από τις εφαρμογές της, όπως το Google Reader, Google Maps, το Google Book Search, το Google Earth, Blogger.com, το Google Code, το Orkut, το YouTube και το Gmail διατηρείται σε αυτά τα κατανεμημένα δίκτυα υπολογιστών [Wik11a]. Αυτοί οι υπολογιστές είναι οργανωμένοι σε τεράστιες συστοιχίες πολλών χιλιάδων κόμβων οι οποίοι βρίσκονται σε διαφορετικά κέντρα δεδομένων.

2.2.1 Συμβολή

Σε αυτή την ενότητα, μελετάμε την αποτελεσματικότητα του συνδυασμού ενός κατανεμημένου πλαισίου δεικτοδότησης με μια συστοιχία αποθήκευσης δεδομένων για τη δημιουργία ενός πλήρως διανεμημένου μηχανισμού για τη δημιουργία και την εξυπηρέτηση ευρετηρίων. Εστιάζουμε στην εξισορρόπηση του φορτίου τόσο κατά την δημιουργία του ευρετηρίου/περιεχομένου,

χρησιμοποιώντας ένα κατανεμημένο πλαίσιο επεξεργασίας δεδομένων όσο και για την εξυπηρέτηση του ευρετηρίου/περιεχομένου μέσω ενός καινοτόμου συστήματος αποθήκευσης δεδομένων. Ο μηχανισμός δημιουργίας και διαμοιρασμού πρέπει να είναι σε θέση να χειρίζεται μεγάλο όγκο δεδομένων και ταυτόχρονων αιτημάτων χρηστών κατά τρόπο έγκαιρο και αποτελεσματικό. Ο σχεδιασμός πρέπει να είναι αρκετά γενικός ώστε να επιτρέπει τον χειρισμό δεδομένων διαφόρων τύπων, από εντελώς αδόμητα μέχρι δομημένα δεδομένα. Επιπλέον, το σύστημα πρέπει να επιτρέπει στον διαχειριστή να καθορίζει εύκολα αποτελεσματικούς κανόνες με τους οποίους να επιλέγει ακριβώς την εμβέλεια και τον τύπο του ευρετηρίου που θέλει να δημιουργήσει. Με λίγα λόγια, πετυχαίνουμε τον εξής διπλό στόχο: πρώτον, υλοποιούμε και εφαρμόζουμε ένα πρωτότυπο πλαίσιο δεικτοδότησης και διαμοιρασμού περιεχομένου που επιτρέπει την εύκολη και προσαρμόσιμη διαχείριση δεδομένων και ευρετηρίου. Δεύτερον, χρησιμοποιούμε το πρωτότυπο σύστημα για τη μέτρηση των επιδόσεων ενός ευρέως διαδεδομένου συστήματος αποθήκευσης δεδομένων υπό ρεαλιστικές συνθήκες κίνησης μεγάλου φόρτου εργασίας για διαφορετικούς τύπους και μεγέθη δεδομένων.

Παρουσιάζουμε ένα κατανεμημένο σύστημα κατάλληλο για δεικτοδότηση, αποθήκευση και διαμοιρασμό μεγάλου όγκου δεδομένων (σε μεγέθη TB και περισσότερο) κάτω από υψηλό φόρτο εργασίας. Οι χρήστες τροφοδοτούν το σύστημα με τα πρωτογενή δεδομένα καθώς και με κανόνες δεικτοδότησης που έχουν να κάνουν με τον τύπο των δεδομένων αυτών, και το σύστημα τα επεξεργάζεται ανάλογα. Η επαυξημένη πληροφορία που προέρχεται από την επεξεργασία των δεδομένων εξάγεται με τη μορφή ενός κατανεμημένου ευρετηρίου και διαμοιράζεται σε ένα μεγάλο αριθμό ταυτόχρονων χρηστών. Για την επιτάχυνση της διαδικασίας, η υπολογιστικά και αποθηκευτικά απαιτητική δεικτοδότηση χρησιμοποιεί το καινοτόμο σύστημα *MapReduce* [DG08]. Για την επίτευξη χαμηλών χρόνων απόκρισης κάτω από υψηλό φορτίο ταυτόχρονων ερωτημάτων, τα αιτήματα των χρηστών εξυπηρετούνται μέσω της *HBase*, μια υλοποίηση ανοιχτού κώδικα του *BigTable* της Google [CDG⁺08].

Παρατηρούμε ότι το πλαίσιο διανέμει τόσο το περιεχόμενο όσο και το ευρετήριο. Το σύστημα *MapReduce* του πλαισίου *Hadoop* [Ara11b] χρησιμοποιείται τόσο κατά τη διάρκεια αποθήκευσης του περιεχομένου όσο και κατά τη δημιουργία ευρετηρίου, έτσι ώστε να αξιοποιηθούν αποτελεσματικά οι υπολογιστικοί πόροι της συστοιχίας. Για την ανάκτηση του περιεχομένου και του ευρετηρίου χρησιμοποιούμε τη *NoSQL* β'άση *HBase* [Ara11d], δεδομένου ότι είναι στενά συνδεδεμένη με το *Hadoop*, έχει μία από τις μεγαλύτερες κοινότητες ανοικτού κώδικα στον χώρο των *NoSQL* και χρησιμοποιείται από ένα μεγάλο αριθμό επιχειρήσεων και οργανισμών παγκοσμίως. Η αποθήκευση του περιεχομένου και του ευρετηρίου γίνεται μέσω του *HDFS*, του κατανεμημένου αποθηκευτικού συστήματος του *Hadoop*.

Το σύστημα πετυχαίνει τους ακόλουθους στόχους:

- **Real-time χρόνοι απόκρισης ερωτημάτων:** Οι χρόνοι εκτέλεσης ερωτημάτων είναι στην τάξη των *millisecond*, για να επιτρέπουν στους χρήστες να κάνουν γρήγορες αναζητήσεις στο περιεχόμενο. Δεν επιτρέπουμε την “on the fly” εκτέλεση βαριών αναλυτικών εργασιών σαν ερωτήματα των χρηστών (όπως γίνεται στα συστήματα Pig [ORS⁺08] και Hive [TSJ⁺09]), καθώς τέτοιες εργασίες δεν είναι κατάλληλες για “live” αναζητήσεις, εξαιτίας του μεγάλου χρόνου εκτέλεσης (από λεπτά μέχρι ώρες).
- **Κλιμακωσιμότητα:** Το σύστημα έχει την δυνατότητα να κλιμακώνεται με ελαστικό τρόπο με την απλή προσαρμογή του αριθμού των συμμετεχόντων κόμβων. Η κλιμάκωση αυτή αντιμετωπίζει αυξημένο όγκο δεδομένων και όγκο ταυτόχρονων ερωτημάτων.
- **Ευκολία στην χρήση:** Οι χρήστες έχουν την ευχέρεια να καθορίζουν απλούς κανόνες για να δηλώνουν το εύρος και το τύπο του ευρητηρίου, και μπορούν να κάνουν ερωτήσεις με “νόημα” πάνω στα δεδομένα (πχ αναζήτηση για συνέδρια που περιέχουν την λέξη `cloud` και πραγματοποιήθηκαν στην `California`).
- **Υποστήριξη ποικίλων τύπων δεδομένων:** Το σύστημα θα πρέπει να υποστηρίζει διάφορους τύπους δεδομένων, συμπεριλαμβανομένων αδόμητων (πχ. `log entries`, αρχεία `HTML`, κλπ.), ημι-δομημένων (αρχεία `XML`) και δομημένων (βάσεις `SQL`).

Το υπόλοιπο της παρούσας διατριβής είναι οργανωμένο ως εξής: στο κεφάλαιο 4 παρουσιάζεται αναλυτικά ο αλγόριθμος *NIXMIG*, το κεφάλαιο 5 παρουσιάζει το κατανεμημένο πλαίσιο επεξεργασίας ευρητηρίων και στο κεφάλαιο 6 παρουσιάζουμε τα γενικά συμπεράσματα καθώς και ορισμένες μελλοντικές ερευνητικές κατευθύνσεις. Τέλος, στο παράρτημα παρουσιάζουμε κάποια αρχικά συμπεράσματα της υπό εξέλιξη εργασίας μας που ασχολείται με την μέτρηση της ελαστικότητας διάφορων `NoSQL` συστημάτων σε υπολογιστικά νέφη.

Introduction

In the past few years, we are witnessing a tremendous increase in the volume of digital information that is created and consumed worldwide. Our era is marked by what is referred to as the “data explosion”. According to Google’s CEO [Kir10], 5 hexabytes of information that were created between the dawn of civilization until 2003 are now created every 2 days. The IDC Digital Universe survey released in May 2010 [GR10] reports that in 2009, despite the global recession, the digital world grew by 62% to nearly 800,000 petabytes. In 2010 it was expected to grow up to 1.2 million petabytes, or 1.2 zettabytes, and by the end of the decade it will be 44 times the size of 2009. A schematic representation of this growth can be seen in Figure 3.1. The small dot represents 1 Exabyte of data which corresponds to the entire US mobile data traffic for 2010, whereas the whole grey box represents 35 Zettabytes, the estimation of the entire mass of digital data in 2020. This data includes web logs and click streams from social networking sites such as Facebook [Fac11] and Twitter [twi11] and large scale e-commerce sites such as Amazon [Ama11e] and eBay [eBa11], raw data produced by RFID tags and sensor networks, call detail records, data from astronomy, atmospheric science, genomics, biogeochemical, biological and other scientific research, military surveillance, medical records, audiovisual archives and financial transactions. For example, CERN’s Large Hadron Collider will produce roughly 15PB of data annually [CER08] and Facebook creates around 20TB of compressed data daily.

When it comes to deal with the “Big-Data” [Jac09] problem, centralized single-server architectures turn out to be inefficient and are now becoming obsolete. It is true that Moore’s law cannot hold for ever: fundamental physical barriers prohibit manufacturers to produce faster

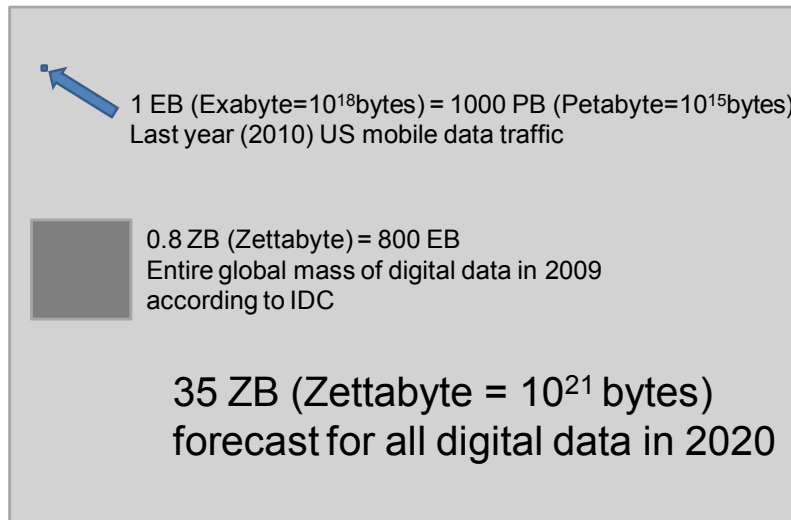


Figure 3.1: A proportional representation of data growth estimation for the current decade.

CPUs [Dub05]. Therefore, exponential data growth cannot be dealt with exponential resource growth. On the other hand, distributed data management systems that are organized in large datacenters are able to provide the required storage and computational resources in a horizontally scalable and cost efficient way compared to monolithic single-server approaches.

These datacenters, which comprise of numerous shared-nothing commodity computing and storage units interconnected with simple networking hardware, have replaced expensive supercomputers and now are the de-facto approach by the vast majority of software companies, especially those that deal with Internet applications. These applications are used to harness the datacenter's computational and storage resources in a unified and scalable manner. Since they process both a large number of concurrent and logically independent user requests along with vast amounts of uncorrelated data, they are able to efficiently balance the imposed load between the datacenter's resources. The successful pioneering example of Google in this domain is now followed by every large Internet service that deals with vast amounts of user requests: YouTube [You11], Twitter [twi11], Skype [Sky11] and Facebook [Fac11] are a representative set of services whose success and efficiency is also attributed to the use of distributed data management systems.

Peer to peer systems (P2P) [Wik11c] are a computing paradigm that exhibits good properties in terms of scalability and self-organization and it is employed in various large scale applications. In a P2P computing system, participating peers are equally privileged and do not need a central coordination authority. The P2P computing model is the opposite of the client-server architecture, in which servers supply and clients consume. In the P2P case, peers are equal resource consumers and producers. Peer to peer systems (P2P) have become popular in the start

of the previous decade from file exchange applications such as Napster, Gnutella [SGG03] and BitTorrent [QS04]. Their applicability is often confused, since they are mostly known for the exchange and dissemination of many times illegal or copyrighted material such as software and audiovisual content between internet users. Despite this, they have been found to have good properties in terms of scalability and fault tolerance. Therefore, they are adopted in numerous applications such as P2PTV [HFC⁺08], distributed databases [LM10, Vol09] on-line multi-player games [KLXH04], Content Delivery Networks (CDNs) [BCMR02], software publication and distribution [DL09], etc.

Cloud computing [Wik11b], the provision of resources on demand via a computer network, is a more recent paradigm that also shows desired scalability properties. As of this, it has been receiving an increasing amount of attention from both the industry and academia. On-demand and pay-as-you-go access (i.e., resource renting and usage for as long as they are needed) to both infrastructure such as CPU cycles, storage and network bandwidth and software such as developing platforms and applications that reside in distant datacenters is a very attractive business model. Amazon's Elastic Compute Cloud (EC2) [Ama11a] and Simple Storage Service (S3) [Ama11c] which were introduced in 2006 as extensions of AWS [Ama11d], its web services platform, were the first cloud services that established Amazon as the leading player in the cloud computing industry. Its successful paradigm was followed by many other companies such as Rackspace [Rac11], RightScale [Rig11], GoGrid [GoG11], etc. Both vendors and users are benefited: the former achieve full utilization of their infrastructure and the latter gain instant resource access without the administration burden or the large upfront costs of building a private datacenter. Users such as small to medium sized enterprises (SMEs) or start ups that need a quick, cheap and scalable access to hardware and software infrastructure are embracing the cloud technology: according to a recent survey [Opi10], more than half of SMEs were going to use cloud computing services in 2010, compared to a mere 22% the previous year. According to [Gar11], cloud computing is ranked first in the list of top-10 strategic technologies for 2011.

Both peer to peer and cloud computing are used to build efficient distributed data management platforms. They both resemble in the fact that they involve large numbers of computing nodes. P2P systems usually interconnect user computers whereas cloud platforms interconnect large computer clusters that reside in distant datacenters. Despite this difference, their use is intertwined: a cloud may utilize P2P methods for self-organizing its physical infrastructure for better resource provisioning, whereas P2P networks may utilize cloud resources for efficient scaling-out in cases of high and short-term loads. For instance, OpenStack [Ope11], a recent open source cloud management platform supported by RackSpace and NASA, builds Swift, its storage platform, on top of Ceph [WBM⁺06], a peer to peer based distributed file system. Moreover, Facebook's Cassandra [LM10], a typical NoSQL cloud datastore, is heavily based in peer to peer algorithms for routing, resource discovery and load balancing.

The efficiency of both cloud and peer to peer computing comes at the price of design complexity, as numerous issues arise when content and resources are dispersed. Load balancing [Cyb89], consistency [HM90], synchronization [Lam78], fault tolerance [Dij74], privacy [MKG07] and security [And08] are representative samples of typical problems that need to be tackled in a distributed environment. Efficient handling of these problems requires approaches that can harness large amounts of resources in a fully decentralized and scalable manner with the least possible human intervention. Self-organizing adaptive systems that can be deployed and operate in an arbitrarily large number of participating nodes with minimal effort are widely employed in these situations.

In this thesis, the problem of load balancing in distributed data management systems is studied. It is true that uneven load distribution can cause serious problems even in over provisioned infrastructures. The recent examples of the FourSquare [Hor11] outage and netflix [Net11] depict that even if an application is deployed in the cloud's virtually unlimited resources, it can suffer significant degradations in high load situations if it is configured in a static way. Unanticipated high loads that occur due to a popular event, e.g., the death of Michael Jackson [PCW09] or Bin Laden, the earthquake in Japan, etc, lead to flash crowd effects [JKR02] in which the computing infrastructure fails to accommodate the high volume of incoming requests. Apart from this, skewed data access in which a small portion of popular data may get the majority of the applied load is another reason for degraded performance. For instance, twitter accounts and hashtags or blogposts of popular persons and the servers that host them can experience high loads, in contrast to servers that host less popular accounts. Skew detection and handling is more difficult in distributed environments [KBHR10] in which resource coordination is not straight-forward.

In the first part of this thesis, *NIXMIG*, an adaptive online algorithm that balances load in distributed range partitioned data structures (i.e., structures that are able to answer range queries) is presented. The problem of uneven load distribution that occurs when peers serve objects of varying popularity is addressed. *NIXMIG* is implemented on top of a Skip Graph [AS07], a structured peer to peer system capable of answering range queries. *NIXMIG* is experimentally and theoretically compared to other load balancing algorithms and the analysis shows that it is faster and consumes less bandwidth during the balancing procedure. In the second part, the problem of indexing and serving large and diverse (unstructured, semi-structured and fully structured) data sets is addressed. A scalable system in which both the index and the content is created and served in a fully distributed way is presented and implemented. The workload of both content and index creation and serving is balanced among cluster peers by combining the power of the latest distributed data analysis frameworks based on MapReduce [DG08] with an open source implementation of Google's BigTable [CDG⁺08]. The system is tested under heavy query load and its mean query response time was kept in the order of milliseconds.

In the following sections, the motivation and the contributions of the aforementioned two systems are briefly presented.

3.1 Load Balancing In Distributed Range-Partitioned Data Structures

Distributed range partitioned data structures, which are a special case of structured peer to peer systems, are widely used in situations where semantically close items need to be stored near each other in an order-preserving way. This data locality property enables the efficient handling of range queries, i.e., queries for resources whose keys lie within a specified range.

Range queries are very important in numerous applications such as OLAP cubes [HAMS97], spatial [Ore86, KS03] and distributed [VSCF09] databases, wireless sensor networks [LKGH03, YHP02, KVD⁺07], online multiplayer games [BPS06, KLXH04], web servers [LN01], data warehousing [LLL00] etc. The following higher level application requests are usually translated into range queries in which portions of consecutive items may be scanned with a single request: GoogleMaps queries like “find all hospitals between Kifissias Avenue and Mesogeion Avenue”, hotels.com queries like “find hotels in Chania with available rooms between 10 June 2011 and 13 June 2011”, Condor [LLM88] or PBS [Hen95] queries like “find all available nodes with RAM between 2GB and 8GB”, database queries like “find all academic professors in Greece with a monthly salary from 5K euro and higher” and prefix queries like “find all words that begin with the letters goo”.

The need for efficient resolution of range queries along with the Big-Data problem has lead to the design and implementation of numerous distributed range-partitioned data structures. Skip Graphs [AS07], Skip Nets [HJS⁺03], P-grids [APHS02], P-trees [CLGS04], BATON [JOV05] and Prefix Hash Trees [RHRS04] are a representative set of such structures. The aforementioned systems organize their index in a way so that queries for semantically close items (for instance, range scans) are answered by contacting only a small subset of the entire overlay. Nevertheless, one of the main drawbacks of these structures is skew, where only a small number of popular items from the ID space are requested. Data skew is a well-documented concern for a variety of applications. It has been widely observed that most Internet-scale applications, including P2P ones, exhibit highly skewed workloads (e.g., [CKR⁺07, RFI02, SW02], etc). Failing or departing nodes further reduce the availability of various content. Consequently, resources become scarce, servers get overloaded and throughput can diminish due to high workloads that, in many cases, can by themselves cause denial of service [JKR02].

One way to handle hotspots and balance load in such cases is by applying typical hash functions such as SHA-1 [Sta95] that transform skewed data access patterns to uniform distributions. This approach is adopted by a special category of peer to peer systems which are called structured P2P systems or Distributed Hash Tables (DHTs) [BKK⁺03] such as Chord [SMK⁺01],

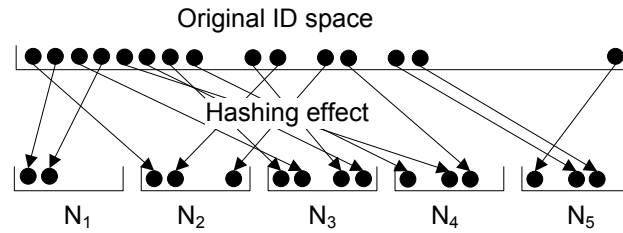


Figure 3.2: Hashing may balance a skewed distribution, but destroys content locality.

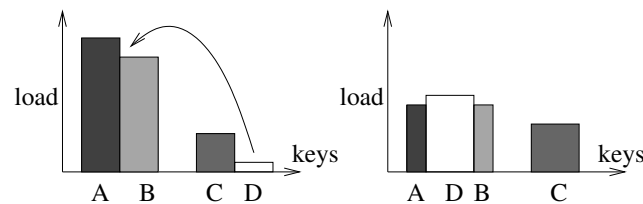


Figure 3.3: Node Migration example. Node D is placed between nodes A and B and shares part of their load.

Pastry [RD01], CAN [RFH⁺01], Tapestry [ZHS⁺04], etc. In Figure 3.2 we present the hashing effect in a skewed ID space: without hashing, nodes N_1 and N_2 would be assigned more items than the others (top of Figure 3.2). Hashing makes sure that, with high probability, each node will get an equal share of the total load regardless the distribution skew (bottom of Figure 3.2). Nevertheless, this transformation comes at the cost of destroying content locality, as consecutive IDs get reassigned in a number of completely different and distant servers.

Another orthogonal way to deal with data skew is by replicating popular items in numerous nodes. However, the content locality constraint of structured P2P minimizes available replica candidates allowing, for instance, only few-hop away neighbors, something that makes balancing even more difficult. What is more, replication not only needs to change the underlying routing protocol to handle multiple replica locations during item searches and insertions, but it must also deal with consistency issues during object updates.

In such cases, load balancing methods [DBK06] that re-partition and re-distribute items between nodes are an appealing solution. The highly dynamic and large scale nature of these distributed data structures, where it is difficult for a single node to have a total network workload overview, poses two basic requirements: online functionality, i.e., the property to make correct decisions only with partial, local workload knowledge, and workload adaptivity, i.e., the ability to quickly respond to workload changes.

Currently, a variety of methods exists in the literature focusing on achieving efficient load balancing for such structures, whether they utilize the notion of “virtual servers” [DKK⁺01, RLS⁺03, SGL⁺06, GS05, ZH05, HLCH11, CT08, LCLC09] or not [KR06, GBGM04, AKK04, BAS04, VORT09,

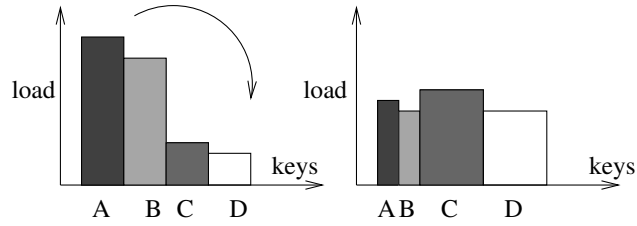


Figure 3.4: *Neighbor Item Exchange example. Iterative key exchanges between (A,B), (B,C) and (C,D) node pairs produce a balanced load.*

LCLC09, SX07, SX08, Jou08]. Yet, they can be categorized in two general strategies: *Node Migration* (hence *MIG*) and *Neighbor Item Exchange* (hence *NIX*). These techniques represent two different approaches to handling the problem: *MIG* utilizes underloaded peers by placing them in overloaded areas of the network (see Figure 3.3, where the height of the bars shows the load of each node, while their width reflects the number of keys served). The newly arriving peer takes up part of the load of its new neighbors. *NIX* balances load through iterative item exchanges between neighboring nodes (see Figure 3.4). The majority of proposed approaches utilize a version of these two schemes in order to finally balance load among peers each responsible for a given range of the data. While they both achieve their goal, their speed and cost greatly vary, making a method that utilizes only one of them inefficient for all cases.

Contribution

NIXMIG's contribution can be summarized in the following:

- ***NIX and MIG analysis:*** These two different methodologies that, iteratively applied, perform load balancing on distributed range-partitioned data structures are formally identified. Their mechanisms are described and their performance in terms of completion time and communication cost is analyzed. An important result of this work is the observation that, through mere key exchanges the achieved result can be highly delayed and the number of exchanged items can be very large, whereas using only node migrations the cost of updating the structure is considerably increased.
- ***NIXMIG design and analysis:*** Based on this analysis, a hybrid method that utilizes both item exchange and node migration in order to minimize overloaded peers and balance the load distribution among them is described. This method manages to adjust the use of migrating nodes with the neighbor item exchange operations: Load moves in a “wave-like” fashion from more to less loaded regions of the structure adaptively, using *NIXMIG*'s version of the Neighbor Item Exchange mechanism. When highly overloaded regions are locally identified, Node Migration is activated. Smart, “skew aware” remote underloaded

node location and placement mechanisms that further decrease *NIXMIG*'s bandwidth consumption are also presented. The algorithm's convergence existence and speed along with the preconditions that need to hold for the system to reach an equilibrium are theoretically studied.

- **Skip Graph implementation and experimental evaluation:** A Skip Graph [AS07] implementation is presented, on top of which *NIXMIG* versus simple *MIG*, *NIX* and another load balancing algorithm proposed by Karger and Ruhl [KR06] are applied and compared. Their behavior is measured and compared in a variety of skewed, dynamic and realistic workloads. The results validate the analysis of the previous sections and show that *NIXMIG* method balances at low cost requiring only one sixth and one third of message and item exchanges respectively compared to [KR06] and high convergence rate being three times faster than [KR06], adapts to changing workloads and is highly customizable.

3.2 A Distributed Framework for Indexing Webscale Datasets

Lately, cheaper storage and bandwidth enables the growth of publicly available datasets: sites like the *Internet Archive* offer access to petabytes of content such as web pages, books, etc. Another example is Amazon offering public datasets (almost) for free through its cloud infrastructure [Ama11b]. Effective indexing is very important to enable the dataset usability, but this is a very difficult task for such data volumes: even Internet Archive [Int11a] does not allow full text search in one of its most interesting services, the WayBackMachine [Int11b] with a dataset of 4.5 PB.

Although index creation is a very demanding task, it has the convenient property of parallelism: the whole task can be split into numerous small sub-tasks (for instance, by splitting the dataset into equally sized partitions) which can be executed by a number of different nodes. Although the task coordination can be performed by legacy schedulers such as Condor [LLM88] or PBS [Hen95], these general-purpose programs are not designed with data-intensive calculations in mind. In order to tackle this problem, a number of data-aware distributed processing frameworks have been proposed as a natural evolution of these old-fashioned generic job schedulers. These systems have the same characteristics with their ancestors, e.g., job scheduling, fault tolerance and load balancing, but they are designed with Big-Data in mind: they seamlessly employ processing techniques such as moving computation near to the data, which enable the easy development and deployment of massively parallel applications. Google's MapReduce [DG08], Microsoft's Scope [CJL⁺08], and Yahoo's PIG [ORS⁺08] are a representative set of such frameworks.

Apart from index creation, index serving and storing is also a resource intensive task, especially when the index size or the user request rate is very large, a typical case in web search

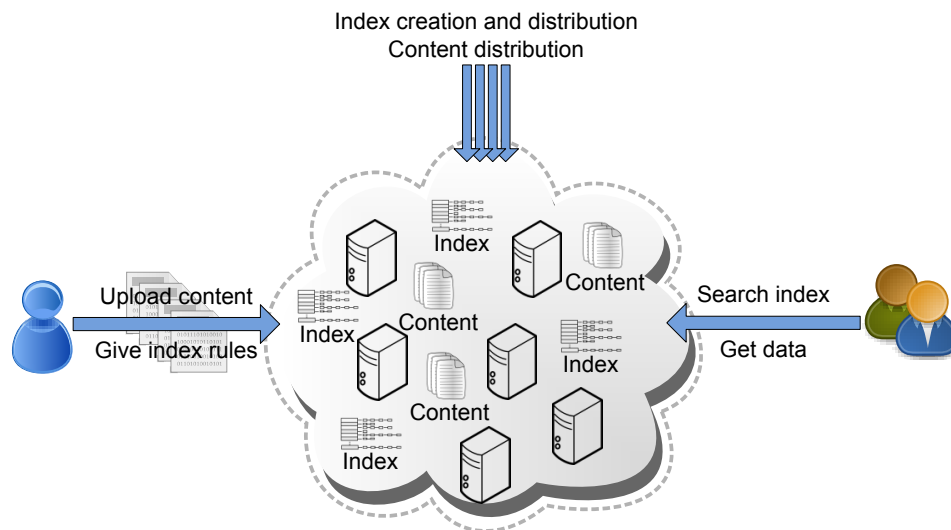


Figure 3.5: Overview of a distributed indexing platform.

engines. Google’s index already had 26 million pages back at 1998 and by 2008 it grew up to 1 trillion pages [AH08]. Nevertheless, index serving is also a task that can be efficiently distributed. For instance, from the early beginnings of Google’s history, its index was distributed among many commodity nodes to avoid scalability issues and to gain better control of performance characteristics. A typical setup of a distributed index creation can be seen in Figure 3.5: administrators upload content along with instructions of what to index. The content is processed and the index is extracted. Finally, users utilize this index to query and retrieve data. Apart from Google’s index, data from many of its applications such as Google Reader, Google Maps, Google Book Search, Google Earth, Blogger.com, Google Code hosting, Orkut, YouTube and Gmail is kept in these distributed networks of commodity nodes [Wik11a]. These computers are organized in cluster-like data stores that span between multiple datacenters and their size scales to thousands of nodes.

Contribution

In this section, we study the efficacy of combining a distributed indexing framework with a cluster-like data store to create a fully distributed mechanism for index creation and serving. We are interested in balancing the load of both the index/content creation by utilizing a distributed data processing framework and the index/content serving by utilizing a scalable data-store. The creation and serving mechanism must be able to handle big amounts of data and concurrent user requests in a timely manner. The design must be generic enough to allow the handling of data of various degrees of structure spanning from completely unstructured to structured data. Moreover, our system must enable the content administrator to easily define efficient rules so as

to choose exactly the scope and type of the index she wants to create. In a nutshell, our achievements are twofold: we first implement and deploy a content indexing and serving framework prototype that allows for easy and customizable data and index manipulation. Secondly, we utilize this prototype to measure the performance of a widely adopted data-store under realistic and heavy workloads for different types and amounts of data.

A distributed processing platform suitable for indexing, storing and serving large amounts (in the orders of TB and more) of content data under heavy request loads is presented. Indexing rules of variable granularity, relative to both type of data and user-input, along with the raw content are processed by the framework. The augmented information is extracted in the form of a distributed index served to an arbitrarily large number of concurrent users. In order to speed up indexing, the compute and storage-intensive index creation and maintenance leverages Hadoop [Apa11b] the open source version of the innovative *MapReduce* [DG08] framework. To achieve low response times in high query load using commodity-node clusters, user requests are served through an *HBase* database, the open source alternative of Google's Bigtable [CDG⁺08].

We notice that our framework distributes both the content and the index. Hadoop's MapReduce [Apa11b] framework is employed both during content upload and index creation so as to efficiently harness the cluster's computational resources. For content retrieval and index querying we utilize HBase [Apa11d] since it is tightly coupled with Hadoop, it has one of the biggest open-source communities in the area of NoSQL and it is used by a big number of companies and organizations worldwide. As our back-end storage for both the Index and the Content we utilize Hadoop's Distributed File System (HDFS) that provides a unified view of the cluster's local storage.

The implemented platform achieves the following :

- **Near real-time query response times:** Query execution times are kept in the order of milliseconds under reasonable amount of load allowing users to perform fast content searches. We are not considering “on the fly” heavy analytical tasks to execute as user queries, an approach followed by Pig [ORS⁺08] and Hive [TSJ⁺09], since these tasks are not suitable for live searches, due to their long execution time which can span from minutes to hours.
- **Scalability:** The system is able to scale in an elastic way by simply adjusting the number of participating server nodes. The system handles both increased storage requirements and concurrent user requests.
- **Ease of use:** Users define simple rules to declare the scope and type of the required index and perform meaningful searches over the data (e.g., find conferences whose title contains `cloud` and were held in `California`).

- **Support of almost any type of data:** The system supports various types of content, including unstructured, semi-structured and structured data such as log entries, HTML files, XML files, relational tuples, etc.

The remainder of this thesis is organized as follows: Chapter 4 presents *NIXMIG* in detail, Chapter 5 presents the distributed indexing framework for webscale datasets and Chapter 6 concludes our work and discusses about possible research directions. Finally, in the appendix we present some initial findings of our on-going work in measuring the elasticity of various NoSQL systems in cloud environments.

Load Balancing In Distributed Range-Partitioned Data Structures

Distributed systems such as peer to peer overlays have been shown to efficiently support the processing of range queries over large numbers of participating hosts. In such systems, uneven load allocation has to be effectively tackled in order to minimize overloaded peers and optimize their performance. In this chapter, the two basic methodologies used to achieve load-balancing are detected: Iterative key re-distribution between neighbors and node migration. These two key mechanisms are identified and their relative advantages and disadvantages are described. Based on this analysis, *NIXMIG* [KTK11,KTK09,KTK08], a hybrid method that adaptively utilizes these two extremes to achieve both fast and cost-effective load-balancing in distributed systems that support range queries is proposed. Its convergence is theoretically proved and as a case study, an implementation on top of a Skip Graph is offered, where *NIXMIG*'s findings are thoroughly validated in a variety of static, dynamic and realistic workloads. *NIXMIG* is compared with an existing load balancing algorithm proposed by Karger and Ruhl [KR06] and the experimental analysis shows that, *NIXMIG* can be as much as three times faster, requiring only one sixth and one third of message and item exchanges respectively to bring the system to a balanced state.

The design rationale behind *NIXMIG* is motivated by the following goals:

Abstraction: To enable *NIXMIG*'s re-usability, our system must be independent of the underlying network overlay. *NIXMIG* should be unaware of the overlay on top of which it is deployed.

Online functionality: In order for the system to be able to scale to a large number of peers, balancing decisions must be made in a distributed way, avoiding the expensive maintenance of centralized load directories that can also act as a single point of failure. As of this, *NIXMIG* must be able to both locate suitable underloaded peers and apply the appropriate balancing action using local per-server measurements.

Fast and cheap Balancing : The algorithm must be able to balance load quickly and in a bandwidth efficient way (i.e., by optimizing both the probing messages needed for underloaded node location and the number of transferred items during balancing actions).

Highly configurable: The system administrator should have the freedom to fine tune *NIXMIG*'s behaviour by adjusting its variables according to, for instance, the average experienced system workload.

The remainder of this chapter is organized as follows: Section 4.1 gives the reader the basic notation and formulation of the problem. Section 4.2 describes the two different primitive mechanisms for load balancing, while in Section 4.3 the hybrid method is presented. In Section 4.4 some enhancements in the Skip Graph structure for more efficient load movement are presented. The experimental results are detailed in Section 4.5, while Related Work and the Conclusions Section conclude the chapter.

4.1 Notation and Problem Setup

This section presents the formulation of the addressed problem. In the following, participating nodes are assumed to be fully cooperative in following the load balancing protocol: we consider *NIXMIG* to be applied in a controlled environment such as, a datacenter, where no malicious or selfish peers can interfere with its operation. Security or incentive mechanisms are orthogonal issues that can be separately addressed.

4.1.1 Notation

Throughout the thesis indexing and storing of M keys with IDs $(0, 1, \dots, M)$ in N nodes, where $N \ll M$ is considered. A key represents an object or item, hence these terms shall be used interchangeably. M keys are divided along N partitions (ranges) with boundaries $r_0 \leq r_1 \leq \dots \leq r_N$ (obviously, $r_i \in [0, M], \forall i \in [0, N]$). Each node N_i stores and indexes keys for the partition $[r_i, r_{i+1})$. Nodes that manage adjacent ranges are said to be neighbors. Two different directions are considered: *forward*, towards which indexed values are increasing and *backward*, where values are decreasing. Node N_i 's forward and backward neighbors are Node N_{i+1} that is responsible for the adjacent range $[r_{i+1}, r_{i+2})$ and Node N_{i-1} that is responsible for the adjacent range $[r_{i-1}, r_i)$ respectively.

As *item load* $l_j(t)$, $j \in [0, M]$ at time t , the number of user requests for this specific item over a specific time interval (for instance, *reqs/sec*) is defined. Item load can be viewed as a portion of bandwidth (kb/sec) consumed on queries for this key. The *server load* $L_i(t)$ of node N_i at time t is the sum of the loads of the items that it stores: $L_i(t) = \sum_{j=r_i}^{r_{i+1}} l_j(t)$. It is noted here that the server load can be arbitrarily chosen according to the application or the resource that needs to be adjusted. For instance, in situations where storage space is more important, the server load can be defined as the number of stored items per server. Nevertheless, the bandwidth consumption caused by query answering is more important and expensive than storage space.

The natural ordering of the indexed keys is kept, so as to facilitate the routing and answering of range queries. Each stored item has a different popularity that is assumed not to be known beforehand and to change over time. Users perform both exact match and range queries. In the case of range queries, more than one node may be contacted in order for the correct answer to be computed.

Each node N_i , according to its capabilities sets a local load threshold, $thres_i$. When the load exceeds this value $L_i(t) > thres_i$, the node wishes to shed some of its load according to the load balancing algorithm that is implemented.

The goal is to transform the set of partition boundaries through consecutive item exchanges or node migrations after some time so that $L_i(t') < thres_i, \forall i \in [0, N]$. In addition, a balanced load distribution is also desirable.

4.1.2 Problem statement – Metrics

Given the previous setup, balancing the server load imposed by skewed query distributions in distributed range-partitioned data structures is requested.

More formally: a set of N partitions $\{r_1(t), r_2(t), \dots, r_N(t)\}$ containing ordered items that are distributed over N separate nodes, with loads $S_{N_1(t)}, \dots, S_{N_N(t)}$ is considered. *NIXMIG*'s goal is to transform the set of partition boundaries through consecutive item exchanges between neighboring nodes after some time into $\{r'_1(t'), r'_2(t'), \dots, r'_N(t')\}$ so that $S'_i < thres_i, \forall i$. Hence, the primary goal is to alleviate overloaded peers from the excessive burden and allow them to shed their load to less loaded servers. Moreover, *NIXMIG* performs a form of *local* load balancing on the overloaded areas of the network. The metrics that are going to be used in order to evaluate the scheme's performance are the number of overloaded peers at various times, the total number of balancing operations, the time-span that the method needs in order to minimize the overloaded servers, as well as some of the standard statistic measures that characterize the network as time advances: Average load, the Gini coefficient etc.

The Gini coefficient (or Gini ratio) G is a summary statistic that serves as a measure of inequality in a population [DW00]. The Gini coefficient is calculated as the sum of the differences

between every possible pair of individuals, divided by the mean size:

$$G = \frac{\sum_{i=1}^n \sum_{j=1}^n |x_i - x_j|}{2n^2 \mu}$$

where n is the number of observations whose values are given by x_i , and $\mu = \frac{\sum_{i=1}^n x_i}{n}$ is their mean. The Gini coefficient has been used as a measure of inequality in size and fecundity in plant populations in numerous studies, e.g. [Wei85, Geb89, KPC89] and as a measure of inequality of income or wealth in economic studies, e.g., [Jov82, Mor62]. Its value ranges between 0 and 1, where 0 corresponds to perfect equality and 1 corresponds to the theoretic case of an infinite population with only one individual having a non-zero value. Recent work [PT09, PT07] proposed its use as a load-balancing metric. Assuming the population comprises of the number of received requests by each node, the value of G is calculated as an index of load distribution among servers. Note here that a low value of G is a strong indication that load is equally distributed among them, but does not necessarily imply that this load is low.

4.2 Load Balancing Using Neighbor Item Exchange and Node Migration

Balancing is performed by transferring keys from overloaded peers to less loaded ones. The necessity for preserving order in a range-queriable data structure requires that any item exchange must be performed only between neighboring nodes in the structure. Nevertheless, there are situations where several neighboring nodes experience similar load stress. In that case, distant underloaded peers can gracefully depart from their place, join in the overloaded area and take a portion of its keys. While this operation seems more efficient, a large number of message exchanges is required for the remote node location and the overlay structure maintenance.

Distributed structures that support range queries perform routing in logarithmic time by maintaining a routing table list of $\log N$ increasingly distant nodes (for an overlay of size N). Without loss of generality, we consider these nodes to be placed in L_{max} levels (at the lowest level, L_0 , each node holds the IDs of its immediate neighbors, etc). The maintenance cost of this overlay is a costly procedure in terms of communication exchange between the participating nodes: Figure 4.1 depicts the message exchanges that occur when node N_m leaves its place at time t_a (left part of Figure 4.1) and re-joins next to node N_p at time t_b (right part of Figure 4.1). Solid lines represent node routing links, whereas dotted ones represent the messages required for overlay maintenance. In the described structure, every node contains $2L_{max}$ routing entries (backward and forward for every level). For simplicity, we describe the procedure for a random level, L_c : Before node N_m leaves its place, it removes every forward and backward link (stored

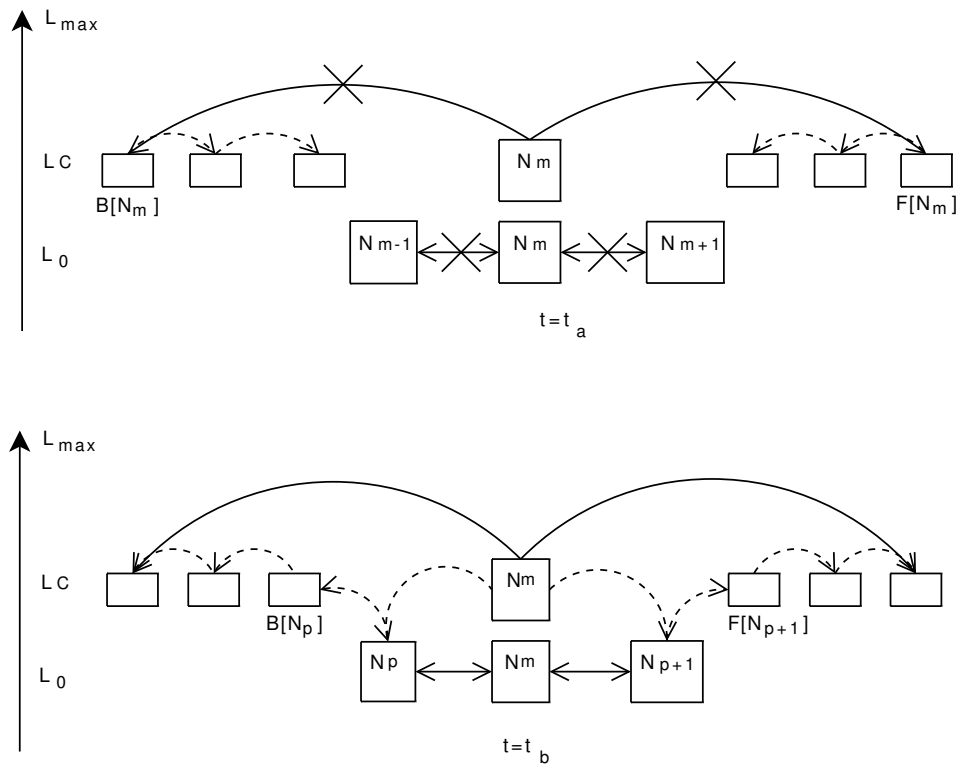


Figure 4.1: Overlay maintenance communication cost for migration of node N_m next to node N_p

in tables $F[N_m]$ and $B[N_m]$ respectively (lines marked with an X). This triggers a number of message exchanges where nodes that were in $F[N_m]$ and $B[N_m]$ (i.e. N_m 's old neighbors) contact a number of distant nodes in order to fill their routing table "hole" (dotted lines on the left side of Figure 4.1). This operation is carried out for every old neighbor in $2L_{max}$ levels. When node N_m re-enters between nodes N_p and N_{p+1} (right side of Figure 4.1) at time t_b , it uses $F[N_{p+1}]$ as forward and $B[N_p]$ as backward links, scans the structure (dotted lines) and creates its own routing table (solid lines).

Two different load balancing algorithms are now described: *NIX* (Neighbor Item Exchange), that transfers only keys between neighboring nodes and *MIG* (Node Migration) that transfers both keys and nodes from remote arbitrary locations.

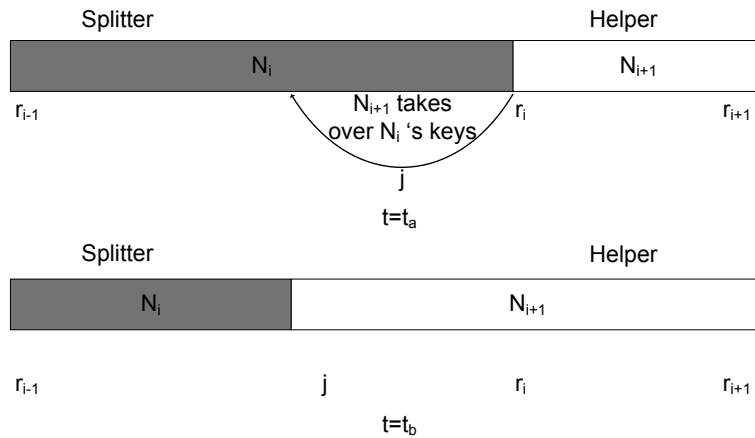


Figure 4.2: Key exchange between two nodes

Algorithm 1: $\text{NIX}(N_i \rightarrow N_{i+1}, load)$

- 1: $\{N_i \text{ calculates key range to pass to } N_{i+1}\}$
 - 2: $j \leftarrow r_{i+1}$
 - 3: **while** $j \geq r_i$ **do**
 - 4: **if** $\sum_{m=j}^{r_{i+1}} l_m \geq load$ **then**
 - 5: key range is $[j, r_{i+1}]$
 - 6: **break**
 - 7: **else**
 - 8: $j \leftarrow j - 1$
 - 9: **end if**
 - 10: **end while**
 - 11: N_i transfers $[j, r_{i+1}]$ to N_{i+1}
 - 12: New N_i partition : $[i, j]$
 - 13: New N_{i+1} partition : $[j, r_{i+2}]$
-

Algorithm 2: $\text{MIG}(\text{Node } N_m \rightarrow \text{Node } N_p, load)$

- 1: $\text{NIX}(N_m \rightarrow N_{m-1}, L_m)$
 - 2: **for all** N_i in N_m 's routing table **do**
 - 3: N_m removes link to N_i
 - 4: N_i searches for new routing entry
 - 5: **end for**
 - 6: $\text{NIX}(N_p \rightarrow N_m, load)$
 - 7: N_m creates new routing table
-

4.2.1 Neighbor Item Exchange

The load exchange between neighboring nodes is described in Algorithm 1. For simplicity, in Figure 4.2 the situation where keys are transferred from Node N_i to its forward neighbor N_{i+1} is described. The transferring node (which will be refer to as the *splitter* peer) sets a pointer $j = r_{i+1}$ and scans its range backwards. The procedure stops when sufficient number of items have been found so as to fulfill its request. This scheme is on-line, because servers can continue to answer queries during that process. What is more, helper nodes can alleviate their neighbors immediately: the helper can answer queries on behalf of the neighbor while the process is not completed, as they are both aware of the location of the pointer j . Queries that exist in the splitter's message queue and request items that have been copied to the helper (that is, they contain items that exist on the right side of the pointer j), are answered by the helper. We note here that the splitter-helper node ID change caused by the range adjustment does not have to be reflected immediately to their remote neighbor's routing tables, as the overlay consistency is preserved (the node ordering remains unaltered). Therefore, the new IDs can be disseminated lazily with the first routing maintenance message exchange. Nevertheless, it is obvious that a major disadvantage of *NIX* is that possibly many iterative such operations may be needed in order to balance load inside large regions of loaded peers.

4.2.2 Node Migration

In Algorithm 2 the situation where Node N_m leaves its place to join next to overloaded Node N_p and take a portion of its keys is described. N_p locates N_m by issuing probing messages to its routing table neighbors until it locates an idle and underloaded peer that could migrate next to it. *MIG* is performed in two phases: In the first phase, Node N_m transfers its partition to its neighboring node N_{m-1} , clears its routing links, and informs them to search for a new entry (leave phase, lines 1-5 of Algorithm 2). In the second step of the procedure (the join phase), Node N_m places itself next to the overloaded peer, accepts a portion of its load and creates its new routing table (lines 6-7 of Algorithm 2). This process was thoroughly described in Figure 4.1.

4.2.3 Analysis

In the following, an analysis to calculate the theoretical worst upper bounds for the completion time and amortized balancing costs (i.e., costs per balancing operation) of *NIX* and *MIG* is presented. Three types of amortized balancing costs are considered, with respect to bandwidth consumption for: item exchanges between nodes (C_{itx}), overlay maintenance during migrations (C_{ovm}) and locating underloaded peers during probing (C_{prb}).

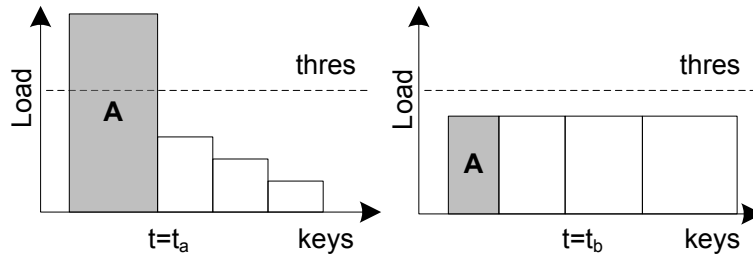


Figure 4.3: Balancing effect of a chain of NIX operations

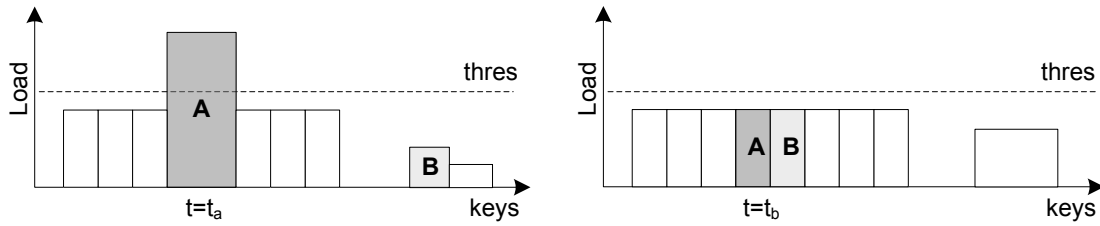


Figure 4.4: Balancing effect of a single MIG operation

In Theorems 1 and 2, the aggregate method of amortized analysis [CLRS01] is used to calculate the average cost of each balancing operation and completion time of *NIX* and *MIG* in the worst case of an initial setup (i.e., worst upper bound of amortized cost). The notations of Section 4.1 are utilized.

Theorem 1. *In the worst case, the running time of NIX is $O(N)$ and the amortized cost per balancing operation is $O(M)$.*

Proof. In the first picture of Figure 4.5 an initial setup of node and item combination that leads *NIX* to its worst behavior in terms of completion time and item exchanges is presented. Buckets represent nodes and balls depict items. Bars above items represent the unit item load $l_j = 1$. For simplicity, each node sets $thres_i = 1$. At the beginning, N_1 contains all M objects, of which only the leftmost N are requested (i.e., they have $l_j = 1, j \in (0, N]$) and the rest $M-N$ are not queried (i.e., $l_j = 0, j \in (M - N, M]$). All other nodes are empty. Since $L_1 = N > thres_1$, N_1 will perform a *NIX* operation with its neighbor N_2 at t_0 and it will transfer to it a total of $M - 1$ keys, keeping only the leftmost key, so that $L_1 = l_1 = 1 \leq thres_1$. Likewise, at t_1 node N_2 will transfer $M - 2$ keys to its right neighbor N_3 keeping only its leftmost key. Finally, after $N-1$ steps, in the second picture of Figure 4.5 all nodes are balanced, since they will be responsible for a single item whose load is 1. N_N will also contain the remaining $M-N$ zero load keys. Given that $N-1$ steps are needed, the running time of *NIX* is $O(N - 1) = O(N)$. By summing all moved items in every step, the total cost is

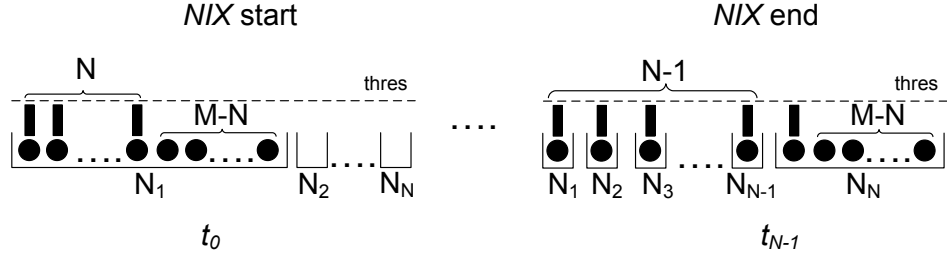


Figure 4.5: Worst case of initial setup and converged balanced network for the NIX case

$$\begin{aligned}
 C_{itxTotal} &= \sum_{i=1}^{N-1} (M - i) \\
 &= \sum_{i=1}^{N-1} M - \sum_{i=1}^{N-1} i \\
 &= M(N - 1) - \frac{(N - 1)N}{2}
 \end{aligned}$$

As no probing and overlay maintenance is necessary, the cost per operation (i.e., amortized cost) then is

$$\begin{aligned}
 C_{itx} &= \frac{C_{itxTotal}}{N - 1} \\
 &= \frac{M(N - 1) - \frac{(N-1)N}{2}}{N - 1} \\
 &= M - \frac{N}{2} \\
 &= O(M - N) \\
 &= O(M)
 \end{aligned}$$

Since $N \ll M$.

□

Theorem 2. *In the worst case, the running time of MIG is constant $O(1)$ and the amortized cost per balancing operation is $O(\frac{M}{N} + \log N)$.*

Proof. In the first picture of Figure 4.6 a worst initial network setup for the MIG case is depicted. Similar to NIX, N_1 contains all M objects, of which only N are requested and every node sets $thres_i = 1$. All other $N-1$ nodes are empty at first. Requested items are evenly distributed in

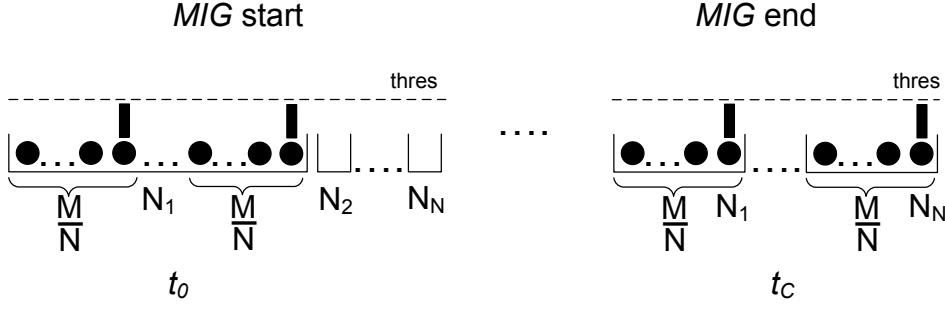


Figure 4.6: Worst case of initial setup and converged balanced network for the MIG case

the ID space: for every $\frac{M}{N}$ objects, there is one with $l_j = 1$ (for instance, $l_j = 1$ if $j \bmod N = 0$, and 0 otherwise). In this setup, N_1 will initiate $N-1$ migrations with the rest of the nodes, where in each migration $\frac{M}{N}$ keys are offloaded from N_1 to the helper. Finally (second picture of Figure 4.6), a total of

$$\begin{aligned} C_{itxTotal} &= \sum_{i=1}^{N-1} \frac{M}{N} \\ &= (N-1) \frac{M}{N} \end{aligned}$$

keys are transferred. The cost for item exchanges then is

$$\begin{aligned} C_{itx} &= \frac{C_{itxTotal}}{N-1} \\ &= \frac{(N-1) \frac{M}{N}}{N-1} \\ &= O\left(\frac{M}{N}\right) \end{aligned}$$

which is basically the cost for a node insertion or deletion (see Theorem 3 of Karger's work [KR06]). The probing cost C_{prb} is $O(\log N)$ since it involves contacting $\log N$ neighbors. Moreover, in most DHT-like networks, overlay maintenance costs $C_{ovm} = O(\log N)$ messages. Therefore, the total MIG cost is $C_{itx} + C_{ovm} + C_{prb} = O\left(\frac{M}{N} + \log N\right)$. Migrations take a constant number of steps as, unlike NIX operations, they are executed in parallel: therefore we consider MIG running time to be $O(1)$ (although overlay maintenance usually takes $O(\log N)$ time, this can happen lazily after the key transfer phase). \square

To gain insight into the behavior of the two algorithms in a more general case, let us consider a typical “balls into bins” setup [MU05], with N items being uniformly distributed among N nodes (we only consider N out of M items, since these items affect node loads). The fraction of underloaded nodes, i.e., nodes with a load less or equal to 1, is calculated by estimating the probability of a node to hold either one or no popular item. Utilizing the equation that calculates the probability of a particular bin to have exactly k balls

$$\begin{aligned} P[N_j \text{ has exactly } k \text{ balls}] &= P[L_j = k] \\ &= \sum_{k=0}^1 \binom{N}{k} \left(\frac{1}{N}\right)^k \left(1 - \frac{1}{N}\right)^{N-k} \end{aligned}$$

we have:

$$\begin{aligned} P[N_j \text{ is underloaded}] &= P[L_j = 0] + P[L_j = 1] \\ &= \binom{N}{0} \left(\frac{1}{N}\right)^0 \left(1 - \frac{1}{N}\right)^N + \binom{N}{1} \left(\frac{1}{N}\right)^1 \left(1 - \frac{1}{N}\right)^{N-1} \\ &= \left(1 - \frac{1}{N}\right)^N + \left(1 - \frac{1}{N}\right)^{N-1} \end{aligned}$$

For large N , this is equal to $\frac{1}{e} + \frac{1}{e} = 0.74 = 74\%$. Moreover, with high probability, the maximum load of a node is $\frac{\log N}{\log \log N}$. Thus, only $100 - 74\% = 26\%$ of the nodes is overloaded and the most loaded node(s) are well under the maximum initial load of N of the most overloaded node in the worst case of *NIX* and *MIG* ($\frac{\log N}{\log \log N} < N$). Both algorithms benefit in this case: *NIX* will initiate small concurrent waves of item exchanges, finishing faster than $O(N)$ (as more waves are done in parallel) and less costly than $O(M)$ (as waves involve a smaller number of nodes and transfer a smaller amount of the id space). Similarly, *MIG* will transfer less items than $O(\frac{M}{N})$, since a fraction ($P[L_j = 1] = \frac{1}{e} = 37\%$) of the nodes will not participate in the balancing procedure, as their load is equal to their *thres* value.

Although *MIG* performs better in terms of completion time and exchanged messages for a large number of N , it needs extra messages for overlay re-organization and probing. This can be avoided with the selective use of *NIX* operations.

In Figure 4.3, a situation where a wave of *NIX* operations is more favorable compared to *MIG* is presented: Node A can shed its load towards its underloaded neighbors without the need for extra remote nodes, leading the neighborhood in a balanced state (right side of Figure 4.3). In Figure 4.4 we describe a situation where a *MIG* operation is more cost-effective than a number of *NIX* operations. Node A is located between nodes that their load is near their *thres* value (left

side of Figure 4.4). In this situation, a chain of *NIX* operations would simply forward the load from one node to another, as there is no nearby underloaded neighbor that could absorb it. On the other hand, the migration of a remote node B next to A (right side of Figure 4.4) solves the problem in one step, justifying the extra number of required probing and maintenance messages needed to locate the underloaded peer and fix the topology respectively. In any case, in order for A to decide the appropriate balancing action, a clear view of the neighborhood's load is required.

From this analysis, it is obvious that fewer *MIG* operations can produce the same result to considerably more *NIX* ones, as helping nodes can be placed anywhere. Nevertheless, it is also evident that each node migration is very costly, while the location of possible helpers and their exact new location has to be optimized. On the other hand, *NIX* avoids probing and routing maintenance messages, but it requires a large number of item exchanges between successive nodes, especially when it is applied in the “middle” of an overloaded neighborhood.

Algorithm 3: REMOTEWAVE(Node N_{p+lc+1} , $exNodes_{lc+1}$ hops)

```

1: Find  $N_m$  such that
    $L_m < thres_m$  and  $N_m$  is idle
2: if such  $N_m$  exists then
3:    $tmpL_m = L_m, j = 0$ 
4:   while  $tmpL_m \leq tmpL_{p+lc+1}$ 
     and  $j \leq exNodes_{lc+1}$  do
5:     if  $N_{m+j+1}$  is idle then
6:        $tmpL_m + = L_{m+j+1}$ 
7:       Node  $N_{m+j+1}$  sends a LockRequest to  $N_{m+j+2}$ 
8:     else { $N_{m+j+1}$  is locked}
9:        $N_{m+j}$  aborts lock
10:    end if
11:     $j = j + 1$ 
12:  end while
13:   $rNodes = j$ 
14: end if
15: return  $rNodes$ 

```

4.3 The NIXMIG Algorithm

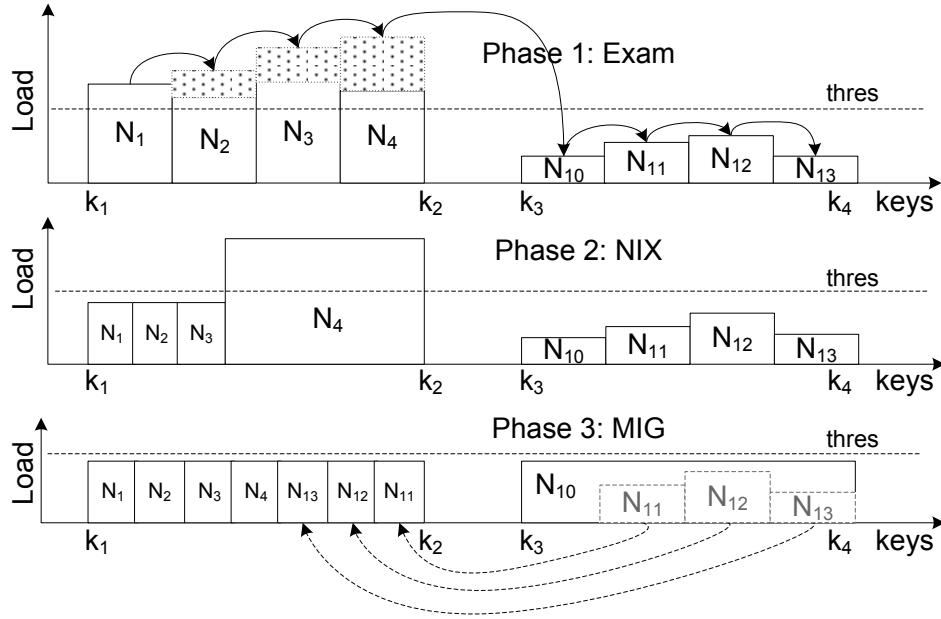


Figure 4.7: A successful NIXMIG operation

Algorithm 5: LOCALWAVE(Node N_p , ttl hops)

- 1: $lc = 0, tmpL_p = L_p, N_p$ sends a *LockRequest* to N_{p+1}
- 2: **while** $lc \leq ttl$ and $exNodes_{p+lc+1} \leq ttl$ **do**
- 3: **if** N_{p+lc+1} is idle **then**
- 4: **if** $tmpL_{p+lc} > overThres$ **then**
- 5: $movedLoad_{lc} = a(tmpL_{p+lc} - thres_{p+lc})$
- 6: **else** $\{0 < tmpL_{p+lc} < thres\}$
- 7: $movedLoad_{lc} = tmpL_{p+lc} - thres_{p+lc}$
- 8: **end if**
- 9: $tmpL_{p+lc} = L_{p+lc} - movedLoad_{lc}$
- 10: $tmpL_{p+lc+1} = L_{p+lc+1} + movedLoad_{lc}$
- 11: $exNodes_{lc+1} = \lfloor \frac{tmpL_{p+lc+1}}{thres_{p+lc+1}} - 1 \rfloor$
- 12: N_{p+lc} locks N_{p+lc+1}
Node N_{p+lc+1} sends a *LockRequest* to N_{p+lc+2}
- 13: **else** $\{N_{p+lc+1}$ is locked $\}$
- 14: N_{p+lc} aborts lock
- 15: **end if**
- 16: $lc = lc + 1$
- 17: **end while**
- 18: **return** $\langle lc, exNodes_{lc+1}, tmpL_{p+lc+1} \rangle$

In this section we describe *NIXMIG*, the proposed hybrid approach. The goal of *NIXMIG* is to balance load by adaptively choosing to utilize either *NIX* or *MIG*. The rationale behind our

Algorithm 4: NIXMIG(Node N_p , *tll hops*)

```

1: LOCALWAVE(Node  $N_p$ , tll hops)
2: if  $exNodes_{lc+1} > 0$  then
3:   REMOTEWAVE(Node  $N_{p+lc+1}$ ,  $exNodes_{lc+1}$  hops)
4: end if
5: for  $i = 0$  to  $lc$  do
6:   if  $L_{p+i} > overThres_{p+i}$  then
7:      $load = a(L_{p+i} - thres_{p+i})$ 
8:   else if  $L_{p+i} > thres_{p+i}$  then
9:      $load = L_{p+i} - thres_{p+i}$ 
10:  end if
11:  NIX( $N_{p+i} \rightarrow N_{p+i+1}$ ,  $load$ ), unlock  $N_{p+i}$ 
12: end for
13: if  $rNodes > 0$  then
14:   for  $i = 0$  to  $rNodes$  do
15:     MIG( $N_{m+i+1} \rightarrow N_{p+lc}$ ,  $\frac{tmpL_{p+lc}}{rNodes}$ )
16:     unlock  $N_{m+i+1}$ 
17:   end for
18: end if

```

method is that *MIG* is fast but costly, whereas *NIX* is slow but cost-effective. Hence, a scheme is devised that, using only local knowledge, identifies conditions where *MIG* is necessary to speed up the balancing process but is not excessively utilized. In short, when *NIX* operations cannot alleviate an overloaded neighborhood, this method employs node migrations for faster load relief in that area.

4.3.1 Algorithm

NIXMIG (Algorithm 4) is initiated when the load of a node N_p passes its self-imposed $thres_p$ value and it is performed in three phases: In the first phase (Exam phase), the overloaded node examines the load status of a number of neighboring nodes (Procedure 5) and, if necessary, an additional number of distant nodes is contacted (Procedure 3). In Table 4.1, we explain the variables used by the aforementioned methods. The node examination is performed in a wave-like manner towards one direction of the structure, where each node contacts its successor. When the first phase is successful, then the algorithm proceeds to the *NIX* phase (lines 5-12 of Algorithm 4) and portions of keys are iteratively transferred from one neighbor to another. Finally, the algorithm proceeds to the *MIG* phase (lines 14-17 of Algorithm 4), where the reserved underloaded nodes of the remote wave offload their keys to their neighbor and take a portion of the range of the final node of the *NIX* wave. We note here that the *MIG* phase is optional: it is triggered only if extra remote nodes are needed to absorb a neighborhood's load.

Table 4.1: NIXMIG variables

variable	definition
tll	Maximum number of contacted nodes per wave
lc	Number of nodes reserved for the NIX wave
$rNodes$	Number of nodes reserved for the MIG wave
$tmpL_p$	Load of node N_p if balancing is performed
$movedLoad_{lc}$	Load that will be moved from N_{p+lc} to N_{p+lc+1} if balancing is performed
$exNodes_{lc}$	Number of extra remote nodes needed at step lc of Procedure 5
N_m	Remote node that will accept migration load
a	Load fraction accepted by the helper if the splitter's load is more than $overThres$

In Figure 4.7 we depict the phases of a successful NIXMIG operation initiated by node N_1 . For clearer presentation, we assume that all nodes have equally set their $thres$ value (dotted horizontal line). In the Exam phase, N_1 issues a Lock Request that eventually reaches N_4 through N_2 and N_3 . N_4 calculates the number of extra nodes that are needed to migrate to the neighborhood to absorb its load, and issues a new request for remote node reservation to node N_{10} . When N_{10} reserves nodes N_{11} , N_{12} and N_{13} , the NIX Phase begins. In the NIX phase of Figure 4.7, nodes N_1 to N_3 iteratively shrink their responsible range by adjusting their boundaries and drop their load under their required $thres$ value. At the end of Phase 2, most of the neighborhood's load ends up to N_4 , but this will happen for a very small period of time, as N_4 has already reserved the requested number of remote nodes to share this load. Finally, in the MIG Phase, the remote underloaded reserved nodes N_{11} , N_{12} and N_{13} sequentially offload their keys to N_{10} , place themselves next to N_4 and take a portion of its range. We notice that at the end of Phase 3 all participating nodes' loads are below their $thres$ value. We now give a more detailed presentation of the algorithm phases.

Exam phase: The Exam phase of NIXMIG serves a dual purpose: it examines the load status of the contacting nodes to decide the appropriate balancing actions, while reserving them to participate in the balancing procedure. The load examination begins with the node's neighborhood (Procedure 5): after each node is successfully reserved (line 3 of Procedure 5) a NIX operation between Node N_{p+lc+1} (that acts as a helper) and its predecessor N_{p+lc} (that acts as a splitter) is simulated by N_{p+lc+1} . The splitter's load in this calculation is assumed to be $tmpL_{p+lc}$ and is equal to the load that would end up to it if a chain of lc NIX operations was initiated by N_p towards N_{p+lc} . Using this variable, node N_{p+lc+1} calculates the load that will be transferred towards it ($movedLoad_{lc}$ variable). This recursive calculation can be seen in Phase 1 of Figure 4.7: The movedLoad variable in steps 1,2 and 3 is depicted with the dotted rectangle

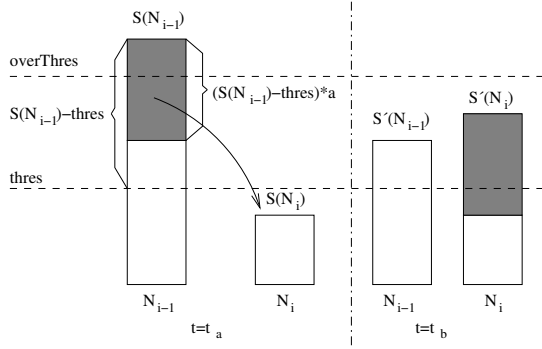


Figure 4.8: Load exchange between splitter N_{i-1} and helper N_i , splitter node in critical condition

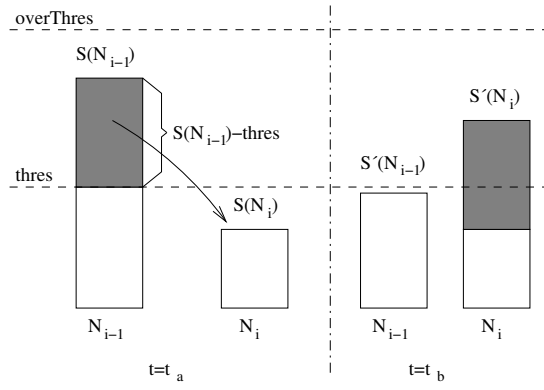


Figure 4.9: Load exchange between splitter N_{i-1} and helper N_i , splitter node not in critical condition

above nodes N_2, N_3 and N_4 respectively. In each step, the $tmpL$ variable is calculated by adding $movedLoad$ to the nodes' current load. $tmpL_{p+nc+1}$ is used by N_{p+lc+1} to estimate the number of extra remote nodes that are required to migrate next to it to absorb the neighborhood's load ($exNodes_{lc+1}$ variable in line 11). The examination of a node's neighborhood finishes when a number of ttl nodes have been successfully reserved, or when it is estimated that more than ttl remote nodes are needed to absorb the calculated extra load (line 2).

When the previous phase finishes, N_{p+lc+1} uses the $exNodes_{lc+1}$ variable to decide whether extra nodes are needed (line 2 of Algorithm 4). If this is the case, it uses the previously described underloaded node location mechanism to locate a remote peer N_m (line 1 of Procedure 3). N_m then tries to reserve $exNodes_{lc+1}$ adjacent nodes that are able to leave their place and help N_p 's overloaded neighborhood. These nodes will offload their keys to N_m before they migrate. The reservation is performed in a similar wave-like manner for at most $exNodes_{lc+1}$ hops. During reservations, each contacted node estimates $tmpL_m$, and if this exceeds $tmpL_{p+lc+1}$, the algorithm moves on to the next phase (line 6 of Procedure 3), with only the so far reserved nodes participating in the migration procedure. Therefore, the goal of the remote locking procedure is to reserve the required $exNodes_{lc+1}$ without overloading N_m that will accept their load when they migrate.

When this phase completes, the locked nodes are ready to begin balancing actions. We note here that locked nodes continue to answer user queries but they do not participate in or initiate other balancing actions until they are unlocked (lines 11 and 16 of Algorithm 4) or a timeout has occurred. Moreover, during the exam phase no item exchanges are performed. If the exam phase is not successful (e.g. not enough underloaded nodes are found, or a contacted node participates in another balancing procedure), nodes are unlocked and an exponential back-off mechanism is applied to the time N_p will wait before it initiates another *NIXMIG* operation.

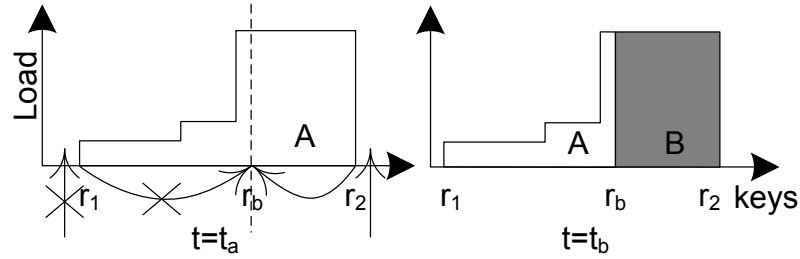


Figure 4.10: Smart remote node placement: A scans its range and decides to place B on its forward direction, minimizing the number of transferred items $|r_2 - r_b|$.

NIX phase: When the locking phase succeeds, the algorithm proceeds to the second Phase and the initiator starts an iterative procedure where portions of ranges are transferred from one locked node to its neighbor for all the lc reserved nodes (lines 5-12 of Algorithm 4). In order to calculate the portion of load that a *splitter* will shed, we introduce the *overThres* threshold, where $overThres > thres$. If the *splitter*'s load is above the *overThres*, then only a fraction a of the extra load is accepted. In Figure 4.8 we present such an example: overloaded node N_{i-1} 's load is above its *overThres* value at $t = t_a$ and offloads only a portion of its extra load to helper node N_i at $t = t_b$. On the other hand, if the *splitter*'s load is below its *overThres* setting, then the *splitter*'s excessive load is fully accepted by the helper. This situation is presented in Figure 4.9: splitter node N_{i-1} offloads all of its extra load to its helper N_i . The purpose of *overThres* is to smooth out the key/load exchanges between sequential *NIX* executions. When *NIX* is performed consequently in a number of overloaded nodes, some nodes may end up with a large portion of the load that was shifted to them during recursive *NIX* procedures from *all* the nodes in the forwarding path. For this exact case, the helper peer does not alleviate the splitter from all of its excessive load, instead, it only accepts a portion of it.

MIG phase: The final step of the algorithm is the *MIG* wave, where a number of $rNodes$ remote locked nodes offload their keys to node N_m , leave their place and join next to N_{p+lc+1} (lines 13-18). Placing remote nodes next to N_{p+lc+1} and not between nodes N_p and N_{p+lc+1} minimizes intra-node communication, as nodes N_p to N_{p+lc} are unlocked after the *NIX* wave (line 11), and their routing tables are not significantly altered.

To sum up, *NIXMIG* first examines the neighborhood of an overloaded node: if its extra load can be absorbed by its neighbors it performs a cost-effective “wave-like” set of successive item exchanges. If this is not the case because, for instance, the entire neighborhood is overloaded, it selectively initiates a more expensive migration request to speed up the process.

4.3.2 Enhancements

In this section, we present enhancements to the original *NIXMIG* algorithm that further decrease the bandwidth utilization of the balancing procedure. More specific, we present remote underloaded node location and placement mechanisms, direction selection heuristics and collision avoidance methods which aim at minimizing the created traffic during balancing operations.

Remote underloaded node location: *NIXMIG*'s performance depends on its ability to easily locate an underloaded node. To avoid random probing or the maintenance of a centralized registry we utilize the query-induced traffic to piggyback information about underloaded nodes. As packets are routed, underloaded nodes add their ids and all participating nodes extract from the incoming packets this information to a local cache. Overloaded nodes use this cache to contact underloaded ones and if they fail to do so, then they resort to random probing.

Remote underloaded node placement: When a remote underloaded node has been successfully located and reserved, the splitter must decide which range to offload to it. In situations where the load is uniformly distributed in the key space, the same load movement results in the same (in terms of transferred items) key movement. Nevertheless, in skewed distributions, this property does not hold (e.g. a small range of items may experience the same load with a larger one), and the smallest possible range must be detected and transferred during load movement. In Figure 4.10 we present this detection mechanism: Overloaded node A is responsible for the key range $[r_1, r_2]$ in which the load is not uniformly placed. In the left side of Figure 4.10 at $t = t_a$ node A simulates two *NIX* operations by scanning its range in the forward direction starting from r_1 (arrows marked with an X) and in the backward direction starting from r_2 . Finally, in the right side of Figure 4.10 at $t = t_b$ node B is placed in the forward direction of A, as this minimizes the number of transferred keys ($|r_2 - r_b| < |r_1 - r_b|$).

Direction selection: Nodes use their local knowledge in order to decide to which direction they should try to perform *NIXMIG* operations. This is achieved by collecting statistics from previous lock attempts or wave operations they had initiated or participated in. The wave direction for both the *LOCALWAVE* and the *REMOTEWAVE* is decided by the wave initiator in the following way: Each peer keeps moving averages with the number of the incoming *LOCKREQUEST* messages it has received from every direction. Then, when it issues a new *LOCKREQUEST* it forwards it to the direction of the neighbor where it has received the *fewer* number of requests. The rationale behind this is that since fewer requests came from this direction, then it must be less overloaded than the opposite direction. Therefore, *NIXMIG* prefers underloaded areas of the structure, where the probability of faster load shedding through them is bigger. Finally, in the absence of previous statistics, the direction decision is a coin toss.

Collision avoidance: *NIXMIG* is equipped with a collision avoidance mechanism during Lock Requests. To avoid unnecessary key exchange, a node can abort a *NIXMIG* attempt during

locking phase. When a *LockRequest* to one direction fails, the *wave initiator* releases the lock, and performs truncated binary exponential back-off to the time it will wait before it tries to lock again towards this direction.

4.3.3 Theoretical Analysis

Load balancing between neighboring nodes can be classified in two general categories [BFH09]: *diffusion* and *dimension exchange* methods. In the case of diffusion [Cyb89], every node balances its load concurrently with every other partner, whereas in the dimension exchange approach [GM96] every node is allowed to balance load only with one of its neighbors at a time (*NIXMIG* falls into this category). Diffusion methods are analyzed using linear algebra, whereas the analysis of dimension exchange methods is performed using a potential function argument. Potential functions map the load distribution vector at time t $\vec{w}(t) = (L_1(t), \dots, L_N(t))^T$ into a single value that shows how “far” the system is from the balanced state. In the case of homogeneous peers, the balanced state is represented by the vector $\vec{w}_{bl} = (\bar{w}, \dots, \bar{w})^T$ where $\bar{w} = \frac{\sum_{i=1}^N L_i(t)}{N}$ (every node gets an equal portion of the total load).

The goal of a balancing algorithm is to ensure that every load exchange between nodes will eventually decrease an initially large potential value and will lead the system to a more balanced (ideal) state. If this drop is ensured, the algorithm converges to an equilibrium. In the case of *NIXMIG*, we define the potential function of an arbitrary load distribution as $\phi(t) = \sum_{i=1}^N (L_i(t) - thres_i)^2$, where $\phi(t)$ is the square of the Euclidean distance between \vec{w} and the vector $\vec{w}_{thres} = (thres_1, \dots, thres_N)^T$ in which every node’s load is equal to its self-imposed *thres* value (ideal balanced state). Note that *NIXMIG* takes into account node heterogeneity and its balanced state is different from \vec{w}_{bl} . What is more, recall from Section 4.1 that *NIXMIG* terminates when $L_i(t) < thres_i \forall i \in [0, N]$, which means that a balanced state is every load distribution vector that satisfies this constraint. In Theorem 3 we prove the convergence of *NIXMIG* algorithm, along with the preconditions that need to hold for the system to reach an equilibrium.

Theorem 3. *Any load balancing action using NIXMIG between a splitter node N_i and a helper node N_j leads the system to a balanced state, as long as the difference of the splitter’s load from its *thres* value is by a constant of $1 - a$ bigger than the difference of the helper’s load from its *thres* value, that is, $(L_i - thres_i)(1 - a) > L_j - thres_j$.*

Proof. In an atomic item exchange between two neighboring nodes, the load that will be moved from the splitter to the helper is $l = a(L_i - thres_i)$, $0 < a \leq 1$ (the case where $thres_i < L_i < overThres_i$ is covered by the general case for $a = 1$). The new loads are $L'_i = L_i - l$, $L'_j = L_j + l$. Now, we have to show that the drop in the potential $\Delta\phi = \phi - \phi'$ caused by this load exchange is positive:

$$\begin{aligned}
\Delta\phi &= (L_i - \text{thres}_i)^2 + (L_j - \text{thres}_j)^2 \\
&\quad - [(L_i - l) - \text{thres}_i]^2 - [(L_j + l) - \text{thres}_j]^2 \\
&= 2a(L_i - \text{thres}_i)[(L_i - \text{thres}_i)(1 - a) + (\text{thres}_j - L_j)]
\end{aligned}$$

$\Delta\phi$ is the product of three terms. The first two are positive because $a \in (0, 1]$ (1) and L_i is overloaded ($L_i > \text{thres}_i$ (2)). So, the potential drop is positive if the third term is positive which happens if $(L_i - \text{thres}_i)(1 - a) > L_j - \text{thres}_j$. \square

Corollary 1. *Any load balancing action using NIXMIG between a splitter node N_i and a helper node N_j leads the system faster to an equilibrium as long as the helper is underloaded, that is, $L_j < \text{thres}_j$.*

Proof. The algorithm's convergence rate is faster as long as the selection of balancing partners ensures a larger drop in the $\Delta\phi$ value. If L_j is underloaded, then the third term of $\Delta\phi$ is larger (as a sum of two positive terms) compared to the case when L_j is overloaded. \square

Corollary 1 is a special case of Theorem 3 that shows the importance for the algorithm's convergence of easily locating underloaded peers. In Corollary 2 we identify the moved load value l_{opt} that maximizes the algorithm's convergence rate leading the system quicker to an equilibrium. We define as $thdif_i = L_i - \text{thres}_i$ the difference of N_i 's load from its thres_i value.

Corollary 2. *NIXMIG's optimum convergence rate is obtained when half of the difference of $thdif_i$ from $thdif_j$ is transferred from splitter Node N_i to helper Node N_j , that is, $l_{opt} = \frac{1}{2}(thdif_i - thdif_j)$*

Proof. $\Delta\phi$ as a function of the moved load l is

$$\begin{aligned}
\Delta\phi(l) &= (L_i - \text{thres}_i)^2 + (L_j - \text{thres}_j)^2 \\
&\quad - [(L_i - l) - \text{thres}_i]^2 - [(L_j + l) - \text{thres}_j]^2 \\
&= -2l^2 + 2(L_i - L_j + \text{thres}_j - \text{thres}_i)l
\end{aligned}$$

We notice that $\Delta\phi(l)$ is a quadratic function of l ($al^2 + bl + c$) with coefficients $a = -2$, $b = 2(L_i - L_j + \text{thres}_j - \text{thres}_i)$ and $c = 0$. Because $a = -2 < 0$, $\Delta\phi(l)$ has a maximum point for

$$\begin{aligned}
l_{opt} &= -\frac{b}{2a} = -\frac{-2(L_i - \text{thres}_i + \text{thres}_j - L_j)}{-4} \\
&= \frac{1}{2}[(L_i - \text{thres}_i) - (L_j - \text{thres}_j)] \\
&= \frac{1}{2}(thdif_i - thdif_j)
\end{aligned}$$

□

In the case of a homogeneous splitter-helper pair ($thres_i = thres_j$) from Corollary 2 we notice that $l_{opt} = \frac{1}{2}(L_i - L_j)$, and thus $a_{opt} = \frac{1}{2}$.

4.4 Case study: NIXMIG over Skip Graphs

As a case study, we have implemented the Skip Graph distributed data structure and we have applied *NIXMIG* on top of it. Skip Graphs are a distributed version of skip lists [Pug90]. Each node in a Skip Graph participates in a number of increasingly sparse doubly linked lists. The lowest linked list at level 0 contains all the nodes, ordered by their identifiers. In a Skip Graph of N nodes, $\log(N)$ levels are required. At each level, a node holds pointers to increasingly “distant” nodes in both directions (forward and backward). A node’s pointers at each level constitute its routing table: during routing, a node examines its neighbors starting from the higher level of the Skip Graph until it finds the one that is closer to the requested value. The routing is performed in logarithmic time, with the difference that no hashing is used, preserving the locality of indexed items. In Figure 4.11, a simple Skip Graph containing five nodes is depicted. Simple arrows represent routing table links between neighboring nodes, whereas arrows with small rectangles represent links to “distant” nodes: we can notice the link between N_3 and N_5 at level 1 of the graph. The range that each node is responsible is depicted under each node (for instance, node N_1 is responsible for range $[8, 15]$).

Despite the locality preserving property of a Skip Graph, there is a difference compared to other DHT structures: Edge nodes (N_1 and N_5) in Figure 4.11 are not connected. While for routing purposes the higher-order lists can be used to make jumps to distant nodes quickly, the use of L_0 (i.e. the lowest order list which connects all the nodes) cannot be avoided when answering a range query that involves more than a single peer. With that in mind, there is an asymmetry created by the fact that the edge nodes are not connected: A peer close to one end of the list has significantly fewer messages arriving to it from one direction of the structure. In the case of *NIXMIG*, this is a reason to prevent the system from equally diffusing load to every direction.

We propose an enhancement on the Skip Graph data structure, where we create links between the edge nodes, thus transforming the structure into a circular one (see Figure 4.11). We notice that the ordering, hence the locality preserving nature of the graph is not affected. In fact, no other node needs be aware of this property, except from the edge nodes. With this modification, edge nodes can now perform key exchange operations like any other node in the system. As an example, let us consider a key exchange between overloaded node N_1 and underloaded N_5 . After a successful *NIXMIG* run, N_1 will be responsible, for example, for partition $[8,15]$ and

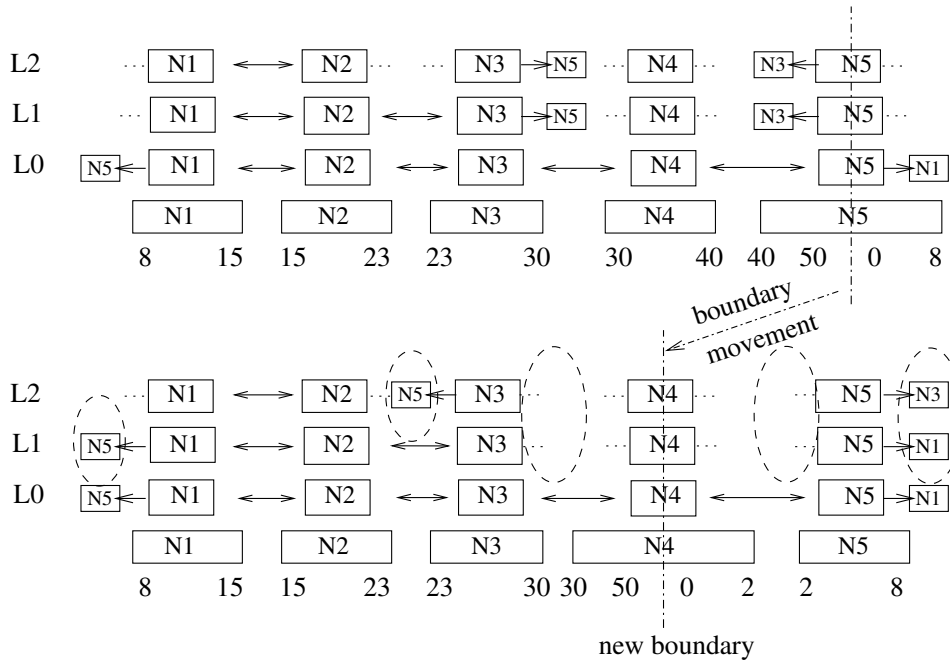


Figure 4.11: Pointers before and after boundary exchange

N_5 for the “boundary” partitions $[40, 50] \cup [0, 8]$ (i.e., the partitions that contain the initially defined k_1, k_M) (see the upper image of Figure 4.11). We note here that this procedure does not affect the routing of the Skip Graph: Queries for the exchanged partition $[0, 8]$ that reach N_1 are forwarded to N_5 . The routing tables of nodes connected to edge nodes remain unaltered.

Routing tables need to change only when the boundary partitions change “ownership”: This happens only when the splitter node is the owner of the “boundary” partitions and during a *NIX* operation it passes a complete partition to its helper node. Figure 4.11 describes this situation, where the splitter node N_5 is the owner of the boundary partitions (upper image) and after the transfer of the range $[40, 50] \cup [0, 2]$, the helper node N_1 is now the new owner of the boundary partitions $[30, 50] \cup [0, 2]$ (lower image). After this operation, the splitter N_5 scans every level of his routing table, and changes his neighbors according to his new range. We describe this procedure in Algorithm 6 for the *forward* direction. The procedure for the opposite direction is similar. New neighboring nodes are selected using the *membership vector* functionality as described in [AS07]. We notice in Figure 4.11 that after the execution of Algorithm 6, node N_5 removes his backward pointer to N_3 and creates a forward pointer to N_3 at level 2, and at level 1 it removes his backward pointer to N_3 and creates a forward pointer to N_1 .

Algorithm 6: FIXROUTINGTABLE(Node N_p)

```

1: for  $level = 0$  to  $MaxLevel$  do
2:    $cursor \leftarrow N_p.next[level]$ 
3:   repeat
4:     if  $cursor$  is suitable for neighbor then
5:        $N_p.next[level + 1] \leftarrow cursor$ 
6:       break
7:     else {remove previous neighbor}
8:        $N_p.next[level + 1] \leftarrow null$ 
9:     end if
10:     $cursor \leftarrow cursor.next[level]$ 
11:  until  $cursor$  is null OR  $cursor$  holds boundary partitions
12: end for

```

4.5 Experimental Results

We now present a comprehensive simulation-based evaluation of our method on our own discrete event simulator written in Java. The implementation of the simulator along with the load balancing protocols, the Skip Graph data structure and the different types of workloads resulted in 6.5K lines of pure Java code. The time unit in our simulation is assumed to be equal to the time needed by a node to perform an operation with another node. Such operations include atomic item exchanges, lock requests, one-hop query routing messages, etc. For instance, a LOCALWAVE wave of $tll = 5$ takes five time units to complete. For the remaining of the experimental section we consider the time unit to be equal to one second. Starting off from a pure Skip Graph implementation, we incorporate our online balancing algorithms on top. By default, we assume a network size of 500 nodes, all of which are randomly chosen to initiate queries at any given time.

During the start-up phase, each node stores and indexes an equal portion of the data, $\frac{M}{N}$ keys. By default, we assume 50K keys exist in the system, thus each node is initially responsible for 100 keys.

Queries occur at rate $\lambda_r = 250 \text{ queries/sec}$ with exponentially distributed inter-arrival times in a 4000 sec total simulation time. Each requester creates a range by choosing a starting value according to some distribution. The range of each created query is constant, and for the 50K setting it is equal to 100 keys (i.e., every range query requests 100 consecutive keys). The total network workload is a product of the query range with the query arrival rate, i.e., $w_{tot} = 250 \text{ queries/sec} \cdot 100 \text{ keys/query} = 25.000 \text{ keys/sec}$ (in every second, around 25K keys are requested in total). Recall from section 4.3.3 that in the ideal balanced state of an homogeneous network, each node should get an equal load portion of $\bar{w} = \frac{w_{tot}}{N} = \frac{25.000}{500} = 50 \text{ keys/sec}$.

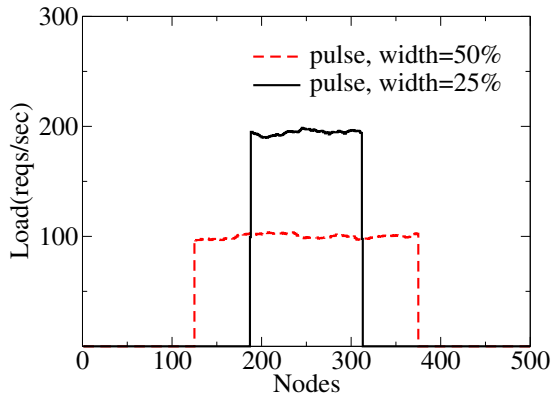


Figure 4.12: Pulse workloads of various width. The width is used to create more or less skewed distributions

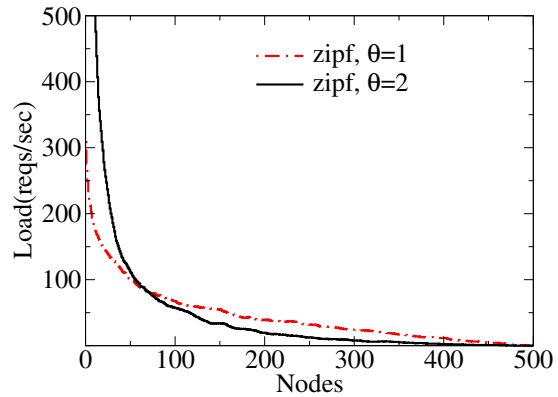


Figure 4.13: Zipfian workloads of various θ . The θ regulates the distribution's skewness

In our experiments, we utilize several different distributions to simulate skew: A *zipfian* distribution, where the probability of a key i being asked is analogous to $i^{-\theta}$ and a *pulse* distribution, where a range of keys has a constant load and the rest of the keys are not requested. By altering the parameters of each distribution (e.g., the θ parameter, the width of the pulse, etc), we manage to create more or less skewed workloads to test our algorithms. In Figure 4.12 we show various pulse workloads with different width. Note here that the total system workload is the area under the pulse. Since throughout the experiments we utilize a stable reference total workload (i.e., 25,000 keys/sec), a pulse with a smaller width has a higher constant load and vice versa. In the pulse case, the width sets the pulse skew: a small width (solid line of figure 4.12) results to a more skewed distribution than a large width (dotted line of figure 4.12). In Figure 4.13 we show two zipfian workloads with various θ values. In the zipfian case, the θ parameter is used to regulate workload skew: a large θ (solid line of figure 4.13) results to a more skewed distribution than a small θ (dotted line of figure 4.13).

A node calculates its load using a simple moving average variable that stores the number of the keys it has served over a predefined time window. To minimize fluctuation caused by inadequate sampling, this time window is set to around 700 seconds. Since nodes in the beginning of the simulation do not have enough samples to estimate their load, we let the system stabilize on the input workload for 700 seconds without performing any balancing operation.

In the following, we plan to demonstrate the effectiveness of our protocol to minimize overloaded peers and create a load-balanced image of the system. As we mentioned before, we are interested in the resulting load distribution (in terms of overloaded servers, load balancing), the rate at which this is achieved (measured in seconds), as well as the cost measured in the number of exchanged messages and items.

During the experiments, *NIXMIG*'s parameters were set to the following values: $thres = 60keys/sec$, $\alpha = \frac{1}{2}$, $tll = 5 nodes$ and $overThres = 400keys/sec$. The idea behind these parameters is the following: the $thres$ value is near the minimum theoretical value of $\bar{w} = 50keys/sec$ for which most of nodes eventually participate in the balancing procedure: the larger the $thres$ value, the easier (i.e., using less operations and bandwidth consumption) it is for *NIXMIG* to bring the system to its equilibrium making its comparison to other algorithms not fair. Furthermore, for homogeneous splitter-helper pairs, we have shown in Corollary 2 that $\alpha_{opt} = \frac{1}{2}$. With respect to tll and $overThres$, in Table 4.2 we experimentally study *NIXMIG*'s behaviour where in each column we vary the tll from 1 to 10 nodes and in each line we vary the $overThres$ from 160keys/sec to 500keys/sec. Table cells show the aggregated performance results for each tll and $overThres$ combination. We notice that a combination of a tll value of 5 nodes (third column) and an $overThres$ value of 400keys/sec (third line) balances load quicker and cheaper compared to other $tll - overThres$ combinations: smaller tll values prohibit *NIXMIG* to examine a sufficient number of nodes, whereas a larger tll value slows down the process due to more inter-node communication during locking procedures. The selected $overThres$ value enables *NIXMIG* to move the optimal amount of load during neighbor item exchanges: larger values lead to unnecessary load movement, whereas smaller values require more balancing operations.

Table 4.2: Exchanged items and messages and completion time for various $overThres$ and tll values

overThres \ tll	1			3			5			10		
	keys	msgs	time	keys	msgs	time	keys	msgs	time	keys	msgs	time
160	62K	21K	92	74K	21K	93	85K	22K	170	134K	39K	554
280	68K	23K	114	79K	22K	120	80K	20K	117	80K	18K	147
400	66K	22K	116	71K	21K	108	70K	19K	77	80K	20K	173
500	69K	23K	147	71K	20K	105	71K	20K	105	80K	20K	166

4.5.1 Measuring the effectiveness of *NIXMIG*

In the first set of experiments, we compare *NIXMIG*'s performance in a number of different input workloads against simple *MIG*, simple *NIX* and the *Item Balancing* protocol (hence *IB*) proposed by Karger and Ruhl in [KR06]. *IB* was chosen as, in contrast with other systems, it applies the same minimal set of operations compared to *NIXMIG*: they both avoid the use of centralized load directories, item replication and node virtualization (for a brief description of *IB* and a survey of similar systems refer to Section 4.6).

As input workload we utilize *pulses* of variable width from 3% to 50% while keeping a constant surface (the pulse height is inversely proportional to its width) and constant surface zipfian workloads of variable θ from 1 to 4.5. In every case, nodes set their $thres$ value to 60reqs/sec.

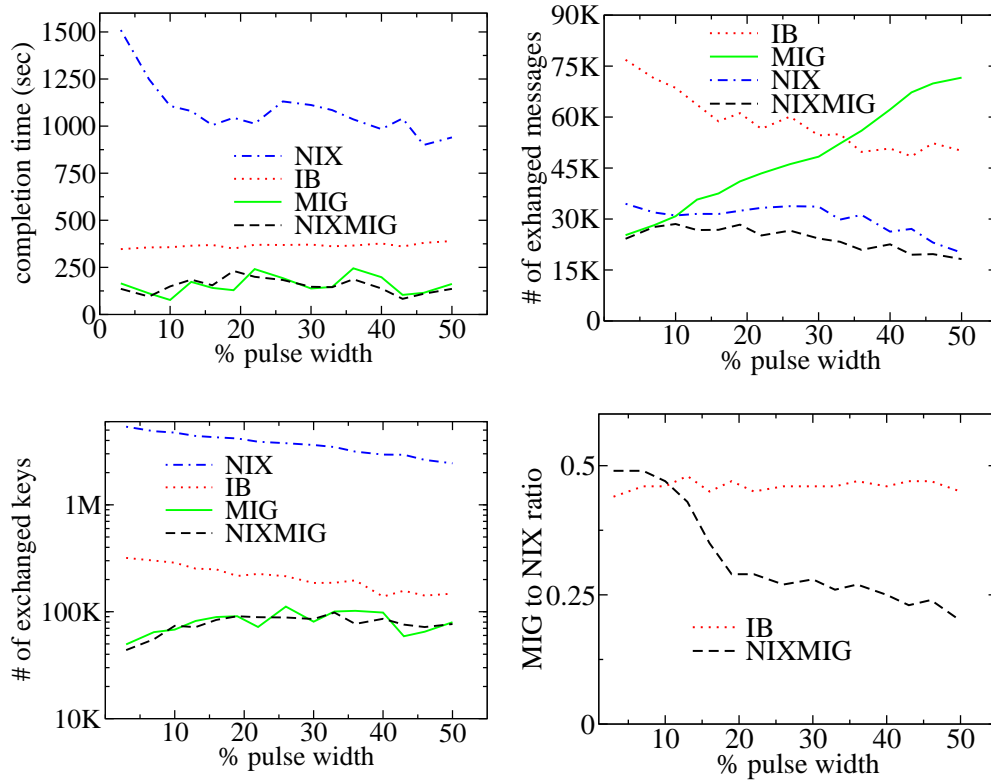


Figure 4.14: Completion time, exchanged messages and items and MIG to NIX ratio of NIXMIG, plain NIX, plain MIG and Item Balancing for various pulse widths.

This *thres* value can also be seen as corresponding to 60kb/sec bandwidth allocation, assuming that, for each request, 1kb of data is transmitted. The simulation terminates when every node has dropped its load under its *thres* value.

We have implemented the *IB* protocol setting $\varepsilon = \frac{1}{4}$ which provides the best balancing result. Moreover, probing messages occur with a rate of $0.1msg/sec$ to keep the probing traffic low. In any case, we terminate the execution of *IB* when 50 seconds of simulation time have passed and no balancing action has occurred.

To apply *NIX*, we use Algorithm 4 and omit the *REMOTEWAVE* procedure: each overloaded node performs only a *LOCALWAVE* followed by a chain of *NIX* operations. For the wave direction selection, nodes use the following simple heuristic: new lock requests are sent towards the direction from which less lock requests were encountered. For *MIG*, nodes omit the *LOCALWAVE* of Algorithm 4 and directly proceed to the *REMOTEWAVE* procedure followed by a chain of *MIG* operations. In every situation, the load is balanced by moving most of the nodes inside the “hot” pulse area that is initially handled by a small number of overloaded nodes. In the *NIX* case overloaded nodes iteratively shrink their range by offloading keys to their immediate neighbors, in

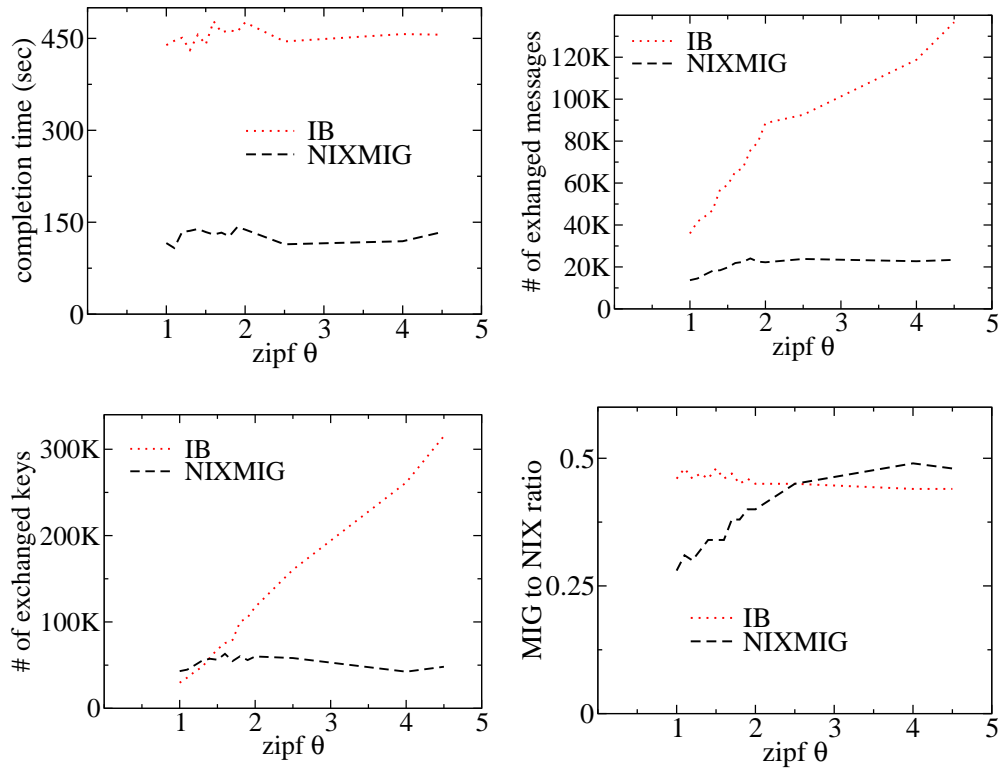


Figure 4.15: Completion time, exchanged messages and items and MIG to NIX ratio of NIXMIG and Item Balancing for various zipfian θ values.

the *MIG* case remote nodes leave their place and rejoin inside the overloaded area and in the *NIXMIG* case a combination of both these methods is adaptively utilized.

In Figure 4.14 we compare *NIXMIG* against simple *NIX* simple *MIG* and the *IB* protocol. In the first graph, we present the completion time of each algorithm for the applied workloads. We notice that both *MIG* and *NIXMIG* balance load 4-8 times faster than *NIX*: for *NIX*, every node must accept and offload a large number of items for the balancing to succeed, whereas in the other two algorithms this is done in a more efficient way. Moreover, we notice that *NIXMIG* converges in almost half the time than *IB*.

Nevertheless, in the second graph we notice that *MIG* is costly in terms of message exchanges, as it carelessly employs a large number of unnecessary node migrations. On the other hand, *NIXMIG* utilizes node migrations only when the load cannot be absorbed locally, thus keeping the number of required messages low compared to both *NIX* and *MIG*. In addition, *NIXMIG* requires less than half the messages compared to *IB*: *IB* requires a large number of probing messages, whereas *NIXMIG* uses the underloaded node location mechanism described in Section 4.3. Furthermore, the number of required messages in the *IB* algorithm increases more due to

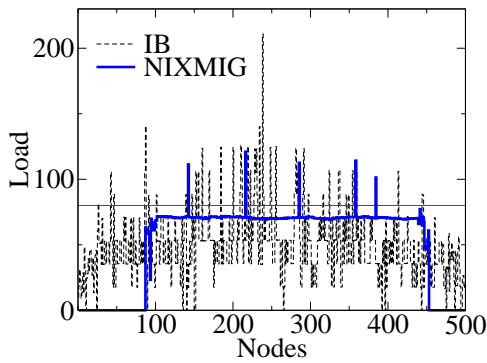


Figure 4.16: Load snapshot at $t=800$ sec for a 3% pulse

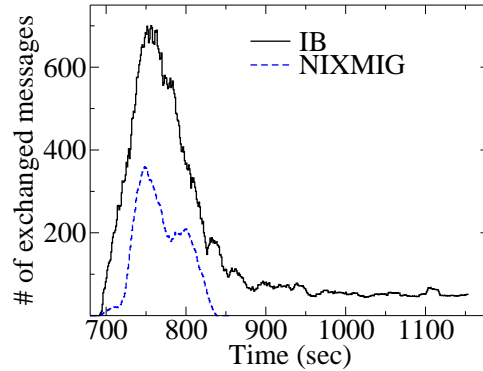


Figure 4.17: Number of exchanged messages during time for a 3% pulse

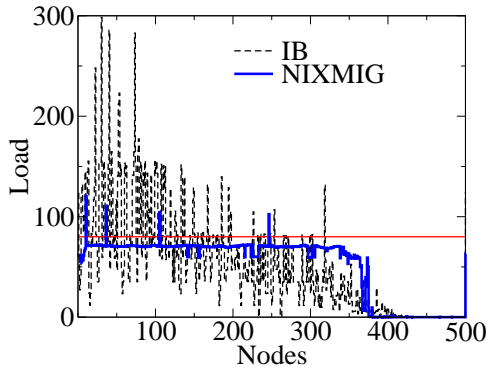


Figure 4.18: Load snapshot at $t=800$ sec for a zipfian of $\theta = 4.5$

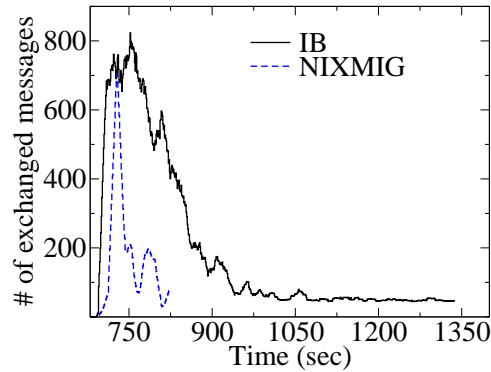


Figure 4.19: Number of exchanged messages during time for a zipfian of $\theta = 4.5$

the fact that mostly node migrations are performed, as its *MIG* to *NIX* ratio is near 0.5 (see the fourth graph).

In the third graph we notice that *NIX* requires two orders of magnitude more item exchanges than *MIG* and *NIXMIG* due to the iterative key transfer procedure. What is more, *NIXMIG* requires roughly the same number of item exchanges compared to *MIG*. *NIXMIG* outperforms *IB* whereas in skewed workloads *NIXMIG* exchanges one third of the items compared to *IB*: the cooperative nature of *NIXMIG* minimizes unnecessary load movement (thus item exchanges) back and forth, unlike *IB* where each node acts on its own. We observe that the *IB*'s number of exchanged messages and items drops when the workload is less skewed: *IB* performs less balancing actions, as it cannot easily locate nodes that their load differs by a fraction of ε .

Finally, in the fourth graph we present *NIXMIG*'s and *IB*'s ratio of migrations to simple neighboring item exchange operations for various pulse widths. Here we notice *NIXMIG*'s workload adaptivity: in extremely skewed workloads of 3-5% pulse widths mostly node migrations are used (recall from Algorithm 2 that each migration requires two neighboring item exchanges, thus the

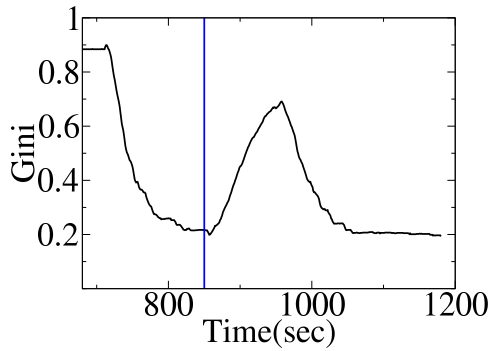


Figure 4.20: The Gini variation for the dynamic setting

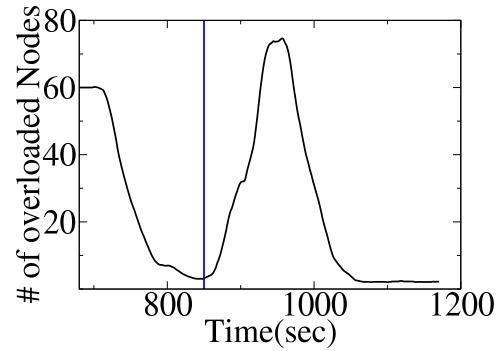


Figure 4.21: Number of overloaded peers over time, dynamic setting

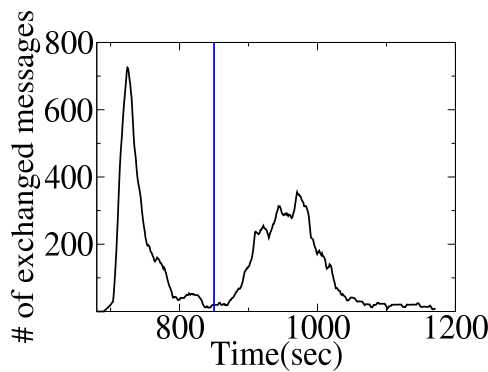


Figure 4.22: Number of exchanged messages over time, dynamic setting

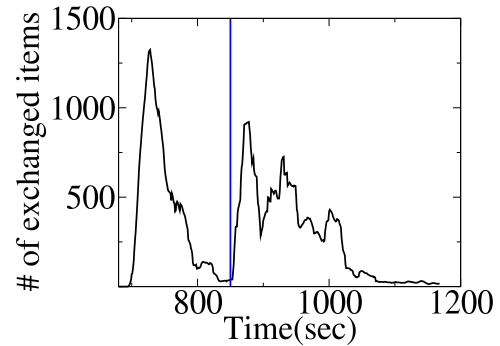


Figure 4.23: Number of exchanged keys over time, dynamic setting

ratio in plain migrations is 0.5). When the pulse's width is increased, the ratio drops as load is absorbed using more neighboring item exchanges and costly remote migrations are avoided. On the contrary, *IB* most of the times carelessly employs node migrations. For an explanation of *IB*'s behavior see Section 4.6.2.

These experiments confirm *NIXMIG*'s adaptivity to an arbitrary workload, as it identifies the most effective balancing action, combining the advantages and avoiding the disadvantages of both plain remote migrations and plain neighboring item exchanges. We continue our experimental analysis with a more thorough comparison of *NIXMIG* against *IB*.

In Figure 4.16 we present a system's load snapshot after 100 seconds for the two algorithms for a 3% pulse. We notice that, unlike *IB* (dotted line), *NIXMIG* (solid line) has successfully dropped almost every node's load under its *thres* value (horizontal red line). Moreover, in Figure 4.17 we present the variation of exchanged messages during time for the *NIXMIG* and the *IB* algorithm. We notice that *NIXMIG* constantly performs less message exchanges than *IB*. What is more, in the *IB* algorithm we notice the constant traffic posed by the random probing messages.

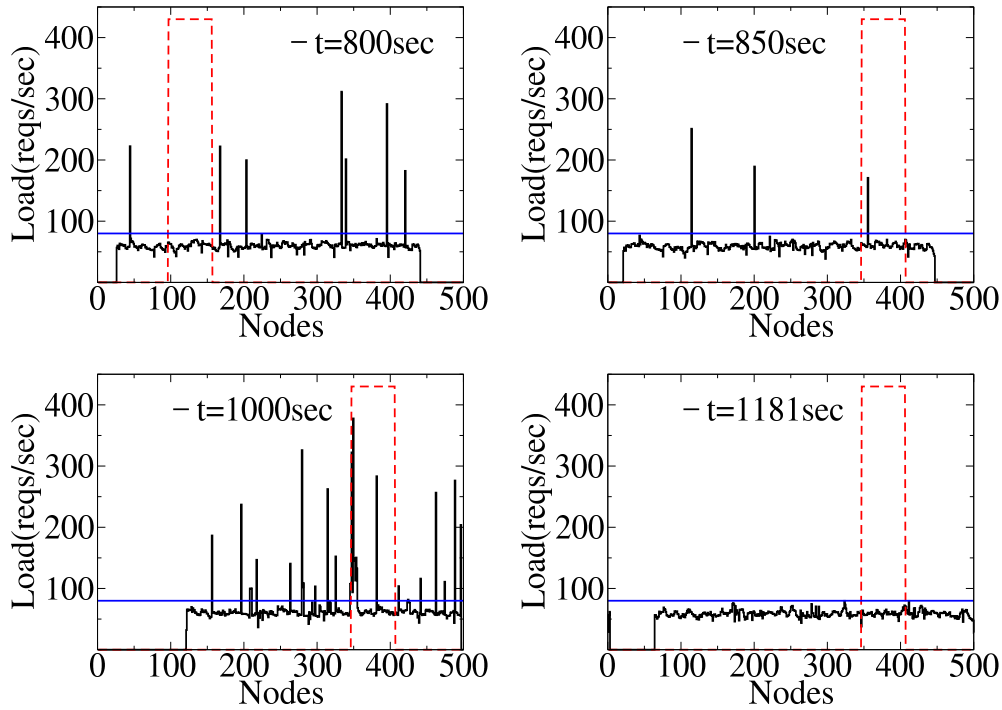


Figure 4.24: Load distribution as it progresses in time: Snapshot taken at times $t=\{800, 850, 1000, 1181\}$ sec

In Figure 4.15 we present the performance results of *NIXMIG* against *IB* for the zipfian setting. In this situation, the workload's skew increases as the θ parameter increases unlike the pulse setting where the skew decreases as the pulse width increases. In the first graph, we notice that *NIXMIG*'s completion time is similar to the one in the pulse setting. On the other hand, *IB*'s completion time increases compared to the respective completion time for the pulse setting: in the zipfian case, the load is spread more uniformly compared to the pulse setting, making it harder for *IB* to identify load imbalances. In any case, *NIXMIG* is three times faster than *IB*. In the second graph, we notice that *NIXMIG* requires a constant number of messages with a slight drop in the less skewed workload area, as more neighboring item exchanges are performed. On the other hand, *IB* requires constantly more messages due to the reasons mentioned in the previous paragraph. In the workloads with $\theta > 3$ *NIXMIG* requires one sixth of the messages that *IB* requires. In the third graph we observe that *NIXMIG*'s and *IB*'s behaviour in item exchanges is similar as in the pulse setting. *NIXMIG* performs more item exchanges than *IB* in the less skewed workloads of $\theta < 1.6$, as it performs more neighboring item exchanges. In more skewed situations, *NIXMIG* performs one third less item exchanges compared to *IB*. The last graph shows the adaptivity of *NIXMIG* where more migrations are employed in more skewed workloads, whereas *IB*

performs mostly migrations in any case. Finally, in Figures 4.18 and 4.19 we present a load snapshot after 100 seconds and the variation of the message exchanges during time respectively for a zipfian workload of $\theta = 4.5$. The same behaviour as in the pulse setting is observed: *NIXMIG* balances load faster and uses constantly less messages than *IB*.

4.5.2 *NIXMIG* scalability

In the following experiment we study *NIXMIG*'s behaviour when the number of participating peers increases. Nodes share a total of 5 million keys and the applied workload is a 10% pulse. We vary the network size from 500 to 50,000 nodes. Table 4.3 presents our findings compared to the 500 node setting (i.e., we register the increase in measurements compared to the 500 node setting result). We notice that the number of messages increases linearly compared to the number of nodes. Moreover, we also notice a slow linear increase of the completion time: even for a 100 times larger network the algorithm terminates only 3 times slower. This happens because multiple *NIXMIG* executions are performed in parallel. Finally, the number of exchanged items remains constant: this shows that *NIXMIG* does not perform unnecessary item transfers when the network size increases.

Table 4.3: Ratio of exchanged messages, completion time and transferred items for various network sizes compared to a 500 node setting.

Nodes ratio	Messages ratio	Completion time ratio	Items ratio
10	13	1.2	1.05
50	102	2.2	0.96
100	208	2.8	0.98

4.5.3 *NIXMIG* performance under dynamic workload

We now present results showing the performance of our *NIXMIG* method when the workload suddenly changes its skew. We assume an initial pulse load of width 12% and height $430req/sec$ where items [10000,16000] are requested. This pulse suddenly moves at time $t=850sec$ to items [34000, 40000]. Note that this is an extreme scenario, since the skew changes completely and abruptly at this time.

Figure 4.20 shows the variation of the Gini [DW00] coefficient over time respectively. Gini values range between 0 and 1, where 0 corresponds to perfect equality and 1 corresponds to the theoretic case of an infinite population with only one individual having a non-zero value. Recent work [PT09,PT07] proposed its use as a load-balancing metric. Assuming our population comprises of the number of received requests by each node, we calculate the value of G as an index of load distribution among servers. Note here that a low value of G is a strong indication that load is equally distributed among them, but does not necessarily imply that this load is low.

Figure 4.21 shows the number of overloaded peers during time. We notice that both metrics are affected immediately after the change in load occurs, nevertheless, *NIXMIG* works over this new situation and manages to reduce both quantities: The Gini coefficient increases when the pulse changes, but *NIXMIG* manages to keep it well under 0.9 (which is its initial value) until it is dropped near 0.2 in the balanced state. Moreover, the number of overloaded nodes slightly passes the initial value of 60 until it is minimized by *NIXMIG*. In Figures 4.22 and 4.23 we present the number of exchanged messages and items during the simulation time respectively where we notice *NIXMIG*'s cost-effective balancing: the number messages does not exceed 700msg/sec (in a 500 node setting) whereas the number of exchanged keys stays under 1200keys/sec (in a 50K setting). Finally, the reason that the convergence time is documented to be larger than that of handling a single pulse is obvious: The very sudden change in skew forces the invalidation of many already performed balance operations and nodes with no load problem suddenly become very overloaded.

Figure 4.24 shows the progress of the balancing process in time: First, at time $t=800$ sec, after 100 sec of balancing time (recall that *NIXMIG* started at $t=700$ sec), just before the query load changes, we show that *NIXMIG* is very close to balancing the load. This is obvious from the improvement shown at $t=850$ sec, where the old pulse diminishes and the new one appears. After this point, the newly overloaded nodes start shedding load to their neighbors (hence the snapshot picture for time $t=1000$ sec). Finally, *NIXMIG* totally balances load (last image).

In the following experiment, we utilize the previously described 12% pulses and we modify the position in the ID-space of the second pulse along with the time we trigger the sudden pulse move. At every execution, the initial pulse is applied over items [10K,16K]. We present our results in Table 4.4. Each column represents an increase in the second pulse's distance from the initial one using a 10% step of 5K keys and each line an increase in the time we trigger the sudden change using a step of 20 seconds. We measure the total number of exchanged items along with the time it took for *NIXMIG* to balance both workloads. We notice that as the new pulse's distance increases in each column, *NIXMIG* performs more key exchanges and takes more time to complete. The same increase in both metrics is noticeable when, in each row we increase the time we trigger the second pulse. Nevertheless, even in the worst case where the second pulse is triggered at $t=820$ sec (60 sec later compared to the 760 sec case) and in the [30K-36K] position (40% further than in the [15-21K] case), *NIXMIG*'s performance is not significantly degraded: only 30% more items are transferred and balancing is 2.2 times slower compared to the 760 sec and [15-21K] combination.

Table 4.4: Exchanged items and completion time for the dynamic setting for various trigger times and new pulse positions

Time \ Position	[15K-21K]	[20K-26K]	[25K-31K]	[30K-36K]
	keys time	keys time	keys time	keys time
760	107K 183	120K 238	125K 303	130K 308
780	109K 201	124K 258	129K 374	132K 315
800	128K 265	129K 268	132K 291	135K 321
820	130K 334	132K 352	138K 380	140K 406

Table 4.5: Number of probing messages and success rate of SmartUNL vs NaiveUNL for various pulse widths

Pulse width	SmartUNL		NaiveUNL	
	# of msgs	%succ	# of msgs	%succ
5	156	64	3.2K	5
10	207	60	2.9K	5
15	168	63	2.8K	5
20	239	70	3.2K	6
25	185	71	3.7K	5

4.5.4 Smart underloaded node location mechanism

In this section, we present the gains of our proposed smart underloaded node location mechanism (hence *SmartUNL*) compared to random probing (hence *NaiveUNL*). In the *SmartUNL* case each query message can piggyback at most five underloaded node ids. We utilize pulses of variable width from 5% to 25%. In Table 4.5 we present the total number of probing messages used in both methods, along with their success rate (defined as the ratio of the successful to total sent messages). We notice that *SmartUNL* sends a small number of messages (around 200) with a high success rate between 60% and 70% whereas *NaiveUNL* utilizes a large number of messages (around 3K) with a very low success rate of 5% to 6%: in the case of *SmartUNL* overloaded nodes utilize their local underloaded node cache to perform “targeted” probing messages, whereas in the *NaiveUNL* setting the probing is done in a completely random way.

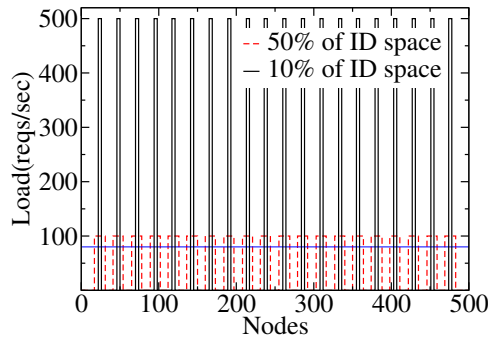
4.5.5 Smart remote node placement mechanism

Next, we study the effect of minimizing the number of exchanged items during load transfers caused by migrations by taking into account load skew, as presented in Section 4.3.2.

More specific, we compare our smart, skew-aware, remote node placement mechanism (hence *SmartRNP*) to the random case (hence *RandomRNP*) where nodes are randomly placed and to

Table 4.6: Ratio of transferred items using *SmartRNP* vs *RandomRNP* and *AdversarialRNP* for various “hot” range percentages

ID space % of “hot” range	<i>SmartRNP</i> # of transf. items	Ratio of transf. items compared to:	
		<i>RandomRNP</i>	<i>AdversarialRNP</i>
10	5.3K	0.47	0.16
20	6.8K	0.52	0.24
30	7.0K	0.59	0.36
40	7.2K	0.66	0.49
50	6.7K	0.71	0.53

**Figure 4.25:** A number of popular areas of 10% width and 50% width in the ID space.

the situation where an “adversary” places remote nodes so as to maximize the number of transferred items (hence *AdversarialRNP*). As our input workload, we consider a number of (around twenty) popular ranges in the ID space for which all keys are requested, whereas all other keys are not queried at all. We vary the width of each popular range so that all “hot” ranges occupy from 10% to 50% of the ID space. In Figure 4.25 we present the initial load distributions of popular ranges occupying 50% (dotted line) and 10% (straight line) of the id space. In Table 4.6 we present the effect of *SmartRNP* for various workloads (first column): in the second column we depict the number of exchanged keys due to a *MIG* operation effectively minimized by *SmartRNP*, in the third column we present the ratio of *SmartRNP* to *RandomRNP* key movement and in the fourth column the ratio of *SmartRNP* to *AdversarialRNP* key movement. We notice that for highly skewed distributions of 10%, *SmartRNP* exchanges only 47% items compared to *RandomRNP* and 16% compared to the adversarial case, while this ratio increases (i.e., *SmartRNP* number of transferred items gets closer to the number of *RandomRNP* and *AdversarialRNP*) for less skewed distributions. This is explained by Figure 4.10: the larger the skew, the larger the difference of $|r_2 - r_b|$ from $|r_1 - r_b|$ making node A ’s decision more critical for the algorithm’s performance. Although the transfer of either $|r_2 - r_b|$ or $|r_1 - r_b|$ results in the same load alleviation of the overloaded node, in highly skewed distributions one range may contain significantly less keys than the other. What is more, we notice that *RandomRNP* performs constantly better

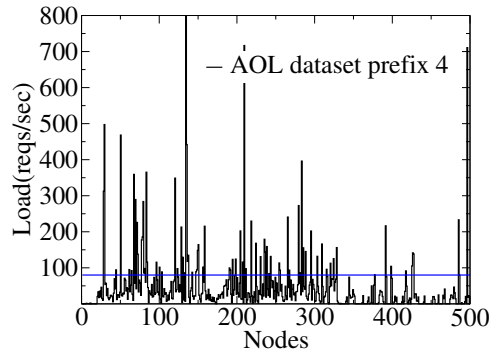


Figure 4.26: Load distribution of the AOL dataset with a query prefix size of 4 characters.

than *AdversarialRNP* (worst case scenario) in terms of transferred items, as with high probability half of its decisions are “correct” (i.e., they minimize key transfer).

4.5.6 NIXMIG in realistic workloads.

Table 4.7: Completion time, number of exchanged messages and MIG to NIX ratio of NIXMIG for various prefix lengths.

Prefix length	time(sec)	messages	MIG to NIX ratio
4	134	16.4K	0.30
5	140	17.9K	0.32
10	150	18.5K	0.34

In the following experiment we utilize a publicly available dataset from AOL that contains twenty million queries from over 650,000 users over a 3-month period [Arr06]. The dataset comprises around 3.7 million distinct keywords of varying popularity which are initially equally divided among 500 nodes. By measuring the number of occurrences of each keyword in the searches, we calculated the query frequency distribution: Clients are generating prefix queries of variable length (e.g., “goo*”, “googl*”, etc) based on the calculated frequencies. Prefix queries, typically used by search engines to provide the autocomplete feature among others, are translated to range queries in our setup. Compared to our previous reported experiments, nodes now store almost 100 times more objects, while the query workload follows a realistic distribution, with the selectivity of the range queries taking many possible values. In figure 4.26 we present the initial load distribution of the AOL dataset with a query prefix size of 4 characters. We notice the load skew: some “lucky” nodes have zero load (e.g., the first 10 nodes that are responsible for the arithmetic values) whereas some other ones are way over their thres value (for instance, a node near the end of the X axis which is responsible for the letter w is overloaded, as a large portion of the queries are for various websites and thus begin with the prefix “www”).

Table 4.7 presents the results for variable prefix lengths. In all cases, *NIXMIG* balances load fast and under 150 sec, a result that is well inline with our previous findings (see Section 4.5.1 – first graphs of Figures 4.14 and 4.15). *NIXMIG* adapts its operations to the applied workload: When the prefix length increases, *NIXMIG* applies more migrations, increasing the number of exchanged messages and the *MIG* to *NIX* ratio. This happens because the prefix length affects the number of matched objects and thus the range query size (“goog*” returns more results than “googl*”). Queries of larger prefix lengths are served by a smaller number of nodes. Consequently, these nodes are excessively overloaded and request more migrations for a faster load alleviation.

4.6 Related Work

Load-balancing is an essential operation for all distributed data structures. The balancing of uneven load distributions has been extensively studied in the case of local indexing structures [SKS92, BAC⁺90] in parallel databases. Nevertheless, the proposed algorithms cannot be applied in the case of distributed data structures, as they require global knowledge of the system load. DHTs such as Chord and Pastry [SMK⁺01, RD01] tackle load-balancing issues by applying hash functions to the data, thus destroying their locality and allowing only exact match queries, as explained in Chapter 3. These systems implement a distributed hash table like data structure, i.e., they provide a put(key,value) and get(key) primitives. The need to preserve the order of the indexed keys to achieve efficient lookups in range-queriable structures prevents us from using simple hash functions to uniformly balance load among nodes. Therefore, we can categorize the available options into two broad orthogonal groups: data replication and data migration approaches.

Data replication alleviates the overloaded nodes by providing extra targets for the incoming requests. Data migration requires actual data transfers between nodes in order to balance load. *NIXMIG* falls into the data migration category. Data replication, with its relative advantages and disadvantages, is applied in conjunction with data migration to further improve performance and fault tolerance. For instance, one sole replica of an overloaded node’s items can effectively drop by half its load (provided that the routing protocol redirects half of the requests to the replica node) but on the other hand, both updates and query routing are more difficult to handle.

4.6.1 Data Replication

In *HotRoD* [PNT06], the authors deal with skewed load distributions by replicating overloaded *arcs* of items (consecutive item ranges). To perform this, they propose a randomized order-preserving hash function, that is used to replicate popular areas into less loaded peers. The same function is used for object location: Hence, a random node containing a replica of the requested

item is contacted. Apart from the obvious consistency issues that arise during replica creations, HotRoD does not deal with replica maintenance: when the popularity of an item changes, the replication factor remains the same. In [PNT10] the authors of *HotRoD* present *Saturn*, an extended version of *HotRoD*, which replaces static replication with dynamic threshold-based replication. In [GSBK04, TR06], the *LAR* and *APRE* protocols for load balancing are presented. *LAR* is evaluated in a structured overlay (Chord [SMK⁺01]) whereas *APRE* is evaluated in an unstructured peer to peer network. Both protocols proactively replicate overloaded nodes and augment the underlying routing table with hints towards the created replicas. Their work is similar to ours in that they both use local load measurements, and they perform balancing actions when the load exceeds a self-imposed threshold. Nevertheless, their work does not deal with range queries. In [LCC⁺02] the authors present and evaluate a number of proactive replication strategies in an unstructured peer to peer system. They study the uniform replication strategy, in which the replication factor of each object is static and the proportional replication strategy, in which the replication factor is proportional to the querying rate of each object. They conclude that both strategies have the same average search size, i.e., the average number of query hops until an object is found, and they propose an optimized replication strategy called square-root replication. In [NW07] the authors also utilize replicas to alleviate overloaded nodes, by employing dynamic caching techniques. Replica holders are organized in a tree-like fashion similar to the consistent hashing approach [KLL⁺97], in which the root node is the node responsible for the total “hot” region. Each child node is responsible for a subset of its parent region. Nodes maintain the number of items they served during a time period (epoch) and if they are below or above a predefined threshold they either create more children, or destroy their children and become themselves leaf nodes. This procedure is performed from bottom to top in the tree and it is called Continuous Hot Spots Protocol.

4.6.2 Data Migration

In the case of peer to peer systems, data migration can be further classified in the node virtualization [DKK⁺01, RLS⁺03, SGL⁺06, GS05, ZH05, HLCH11, CT08, LCLC09] and one ID per server [KR06, GBGM04, AKK04, BAS04, VORT09, LCLC09, SX07, SX08, Jou08] strategies. In the former, every actual server can host a number of virtual servers which are responsible for small disjoint ranges of its items, and balancing is performed by moving virtual servers between actual ones. It has been widely used because of its ease of use (virtual servers can be concurrent threads of a DHT implementation running on the same machine), but its main drawback is the increased bandwidth and memory consumption caused by the maintenance of numerous routing tables (the number of open network connections gets multiplied by a factor of $\Omega(\log n)$ [Man04]).

One ID per server approaches

In the work of Karger and Ruhl [KR06] (Item Balancing, hence *IB*) a *work-stealing* technique is applied: peers randomly probe distant nodes and compare their loads. If the load of the less loaded node is smaller than a fraction of $0 < \varepsilon < \frac{1}{4}$ of the more loaded node's load (i.e., $l_j \leq \varepsilon \leq l_i$ where l_j is the less loaded node and l_i the more loaded node) then a migration (*MIG*) or an neighboring item exchange (*NIX*) is performed. In a nutshell, *IB* is executed in two phases. In the first phase, *IB* tries to perform a *NIX* operation, and if this is not possible, in the second phase it also tries for a *NIX* operation and if this is still not possible, only then it resorts to a *MIG* operation. In the first phase the algorithm distinguishes two cases, according to whether nodes are immediate neighbors (i.e., forward or backward nodes) or distant nodes. If nodes are immediate neighbors (i.e., $i = j + 1$), node N_j transfers $\frac{l_i - l_j}{2}$ items to N_i so that both nodes end up with the same load $\frac{l_i + l_j}{2}$. If nodes are not immediate neighbors (i.e. $i \neq j + 1$), then the algorithm moves to the second phase and the underloaded node N_j checks its immediate neighbor N_{j+1} to see if it is more loaded than the distant node N_i (i.e., if $l_{j+1} > l_i$). If this is the case, it performs a *NIX* operation between N_j and N_{j+1} . Otherwise, N_j migrates next to N_i and takes a part of its load. Although from the algorithm description we notice that *IB* prefers *NIX* operations and resorts to *MIG* only in one sub-case of the second phase (when the other two *NIX* attempts have failed), the majority of its operations are *MIG* operations (see paragraph 4.5.1 in the experimental section). This behavior is explained in the following way: Since a node randomly contacts one out of its $O(\log N)$ neighbors and only $O(1)$ are immediate neighbors, then the first phase will be selected with a low probability. For instance, in a network with 500 nodes where each node maintains a routing table with $\log 500 \approx 10$ peers, a peer will randomly select its immediate neighbors and execute phase one with a probability of $\frac{1}{10} \approx 10\%$, whereas phase two is executed with a probability of $\approx 90\%$. Now, in the second phase, since the remote peer checks its load with its immediate neighbor to initiate a cheap *NIX* and avoid an expensive *MIG*, their loads must be significantly different. Nevertheless, since range queries mostly affect a number of continuous servers due to their locality property, these nodes most of the times experience similar loads. Therefore, *IB* mostly selects the second sub-case of the second phase which is a *MIG* operation.

Ganesan et al [GBGM04] propose a balancing mechanism that works on top of a Skip Graph system [AS07]. In their work they also distinguish two different load balancing operations, *NBRADJUST* which is a *NIX* operation and *REORDER* which is a *MIG* operation. Nevertheless their use is not regulated, although, similar to Karger's *IB* at first they try to employ *NIX* and finally they resort to *MIG*. In their work, the authors introduce the *Imbalance Ratio* σ which is defined as the asymptotic ratio between the largest and the smallest load in the system. Their

algorithms try to ensure the smallest σ value at the end of the balancing procedure. Ganesan performs balancing by utilizing an approach called the *Threshold Algorithm*. In this algorithm, each node is aware of an ordered set of load thresholds and is responsible for periodically updating a shared directory with its current load. The ordered set consists of an infinite, increasing geometric sequence of thresholds $T_i = \lfloor c\delta^i \rfloor$ for all $i \geq 1$ and some constant c . When a node's load crosses a threshold T_i , then a balancing operation is initiated, using a procedure called *ADJUST-LOAD*. Similar to Karger's approach, the algorithm in the first phase tries for a *NIX* operation: it checks its immediate neighbors, and if they are underloaded the initiator averages its load with its less loaded one. If both neighbors are overloaded (i.e., they have more than $\frac{1}{\delta}$ times the initiator's load), then the algorithm moves to the second phase. In the second phase, the initiator contacts the globally least-loaded node to perform a *MIG* operation. If the least-loaded node's load is small enough (less than $\frac{1}{\delta^2}$ times the initiator's load) then a *MIG* operation is performed, otherwise the system is considered balanced. In the case where $\delta = 2$ the authors present and analyze the *Doubling Algorithm*. They prove that the *Doubling Algorithm* ensures $\sigma = 8$. What is more, they search for the δ value for which they can guarantee the smallest imbalance ratio. They conclude that this happens when δ is larger or equal to the golden ratio $\phi = \frac{\sqrt{5}+1}{2}$. In the extreme case where $\delta = \phi$, they introduce the *Fibbing algorithm*, in which the ordered Threshold set consists of the Fibonacci numbers (with $T_1 = 1$ and $T_2 = 2$). They prove that the *Fibbing algorithm* guarantees an imbalance ratio of $\phi^3 = 4.24$. Compared to *NIXMIG*, the main drawbacks of Ganesan's approach are the following:

- The algorithm needs to maintain a centralized directory where nodes periodically report their load. This is a costly procedure in terms of bandwidth, since each node needs to periodically report its load, so that the load information is up-to-date.
- Their definition of load is the number of stored items per node, whereas in our case we are interested in the bandwidth consumption during query answering. We argue that bandwidth is more "expensive" and valuable resource than storage space.
- They do not take into account node heterogeneity. *NIXMIG*'s nodes can set different thresholds according to their processing power or available bandwidth. This is not possible in the Ganesan's case: The Threshold algorithm tries to equally balance load among all peers, irrespective of their capabilities.

In [AKK04], the authors of the Skip Graph structure propose a second layer on top of a simple Skip Graph, the buckets layer. Each bucket contains a number of ordered items and each server may have several buckets. Buckets can be classified as active (i.e., contain a number of items) or free (i.e., they are empty). Moreover, buckets are further classified as open or closed, according to whether they can store more items or not. Next, the list of active buckets is partitioned

into groups of two or three, maintaining the invariant that every closed bucket is adjacent (i.e., is neighboring) to an open bucket: this invariant is used to make sure that the initial structure is quite scarce (i.e., it has “holes” that are able to store possible newly arrived items) which favor simple *NIX* operations instead of *MIG* ones. This happens as during item insertions, the node firstly checks to see if it can fill up an adjacent open bucket with a *NIX* operation. If the item is to be stored by a closed bucket, the bucket tries to offload some of its keys to its neighbor, which, is an open bucket. If its neighbor also is full, then a remote request is done for a *MIG* operation is done and a free bucket joins next to the structure and takes part of the new keys. The main drawback of this scheme is the requirement of a list of free nodes: This luxury cannot be considered trivial in actual deployments. Moreover, it complicates the design scheme by adding complexity to the routing creation and maintenance and by fragmenting the ID space in numerous small buckets.

Similar to [AKK04] is the P-ring [CLM⁺11, CLM⁺07] load balancing methods, since the authors also employ helper peers in their approach. Nodes find helper peers by utilizing the P-ring structure and search for a “special” reserved key whose value contains the list of the available helpers. They argue that without free-helper peers the use of a node leave and join operation is very expensive, because of the creation of new routing table links and item transfers. Nevertheless, as previously discussed, free helper peers mean that these peers will remain idle. To tackle this, the authors propose a more “active” participation of the helper peers: they assign helper peers to actual ones (called masters or owners) and they are made responsible for a fraction of their master’s id space. Load is split in half, without considering node heterogeneity as in *NIX-MIG*’s threshold based load exchange. Instead of thresholds, they utilize the *sf* variable (storage factor). Nodes try to keep the number of items they own during insertions/deletions between *sf* and $2*sf$. When they have more than $2*sf$, they request a helper peer to join next to them and take part of their load (split algorithm, which is similar to [AKK04]). On the other hand, when they have less than *sf*, they acquire items from their immediate neighbors (merge algorithm). Their neighbors either pass a portion of their values to them (i.e., they perform a *NIX* operation) or they transfer all their items and become helper peers. As of this, P-ring tries to equally distribute load among peers, while *NIXMIG* is interested in keeping every peer’s load under its *thres* value.

In Mercury [BAS04], the authors also use probing and node migration to solve load balancing problems. Mercury overlay construction is inspired by Kleinber’s small world phenomenon [Kle00] something that bounds the expected number of hops between neighbors to $O(\log n^2)$, where *n* is the number of participating nodes. As of this, each node *A* selects its routing table links with the following procedure: *A* draws an integer $x \in (0, n)$ using the harmonic distribution $h_n(x) = \frac{1}{n \log x}$ and links with node *B* which is *x* hops away from *A*. Nodes use a novel random sampling procedure with probe messages to calculate the average network load. The main idea is to sample the load distribution locally and then exchange this information with the

other network in an epidemic style. In the local probing case, each node contacts d of its immediate neighbors (called the d -neighborhood) and calculates an average load for this neighborhood. Load estimation for remote ID-space areas is performed by issuing k_1 probe messages with a small TTL value of $\log n$. Each node in the path of the probing message selects randomly one of its routing links and forwards the probe until $TTL = 0$. Finally, nodes maintain a fresh list of the k_2 most recent load estimates. These estimates are used to calculate a value average or a histogram that depicts the value's distribution in the ID space. In the load balancing case, nodes utilize their local load histogram to initiate balancing actions if their load is above the histogram's average. In that case, they contact a random node from a lightly loaded area of the ID space and request from it to leave its place and join next to them, which is effectively a simple *MIG* operation. A node is considered overloaded when its load to the average load \bar{L} ratio is bigger than α and underloaded when this ratio is smaller than $\frac{1}{\alpha}$, where $\alpha \geq \sqrt{2}$. Therefore, Mercury bounds the node loads between $\frac{1}{\alpha}$ and α . We notice that Mercury, in contrast to *NIXMIG*, neither does it take into account node heterogeneity, nor does it regulate the usage of *NIX* and *MIG* actions.

Similar to Mercury is the Oscar system [GDA07, GDA10]. Oscar peers utilize random sampling to figure out the network load distribution (i.e., average network load) and proceed to balancing actions by exchanging keys between participating peers. Oscar is also based on Kleinber's small world phenomenon [Kle00] during probing and overlay construction. Nevertheless, instead of utilizing the simple harmonic distribution for long link (i.e., routing table entries) construction, Oscar is based on the histogram estimation method of Mercury. As of this, instead of picking $\log n$ integers $x \in (0, n)$ using the harmonic distribution $h_n(x) = \frac{1}{n \log x}$ to split the ID space in logarithmic-sized partitions (i.e., each partition size is, for instance double the size of the previous one, etc), like Mercury does, Oscar splits the ID space in $\log n$ partitions so that the number of stored items per partition is logarithmic in size (i.e., each partition contains, for instance, twice the objects compared to the previous one), according to the item distribution. The distribution is estimated using the histogram creation mechanism of Mercury. Balancing is performed in the same way with Mercury, upon node joining and leaving, where arrived or departed nodes interact with the most loaded or the least loaded node in the overlay accordingly. As of this, Oscar has the same disadvantages with Mercury with respect to *NIXMIG*.

Giakkoupis et al [GH05] propose the S&M protocol, a load balancing method that is based in the multiple random choices scheme, which is derived from the power of two choices of Mitzenmacher [Mit01]. The protocol is initiated during node joins and leaves, and tries to balance load equally among peers. They also consider heterogeneous nodes by using weights that are powers of two: for nodes n_1, n_2 with weights w_1, w_2 respectively, n_1 can manage a ID space that is $\frac{w_1}{w_2}$ times larger than n_2 can handle. In the case of a node join, the new node selects at random a logarithmic number of points in the ID space, splits in half the largest node segment (i.e., the ID space for which a node is responsible) containing at least one of the selected keys and gets

the other half. During a leave, the node again samples a logarithmic number of nodes, selects the node with the smallest segment (i.e., the least loaded node), the least loaded node offloads its keys to its neighbor and takes over the keys of the departed node. In essence, the S&M protocol performs a random sampling to identify the most/least loaded node in the case of node joins/leaves respectively. As in most cases, balancing is performed by splitting the ID space in half. We also notice that S&M performs plain *MIG* operations after a random probing of logn messages. Compared to *NIXMIG*, S&M does not regulate the use of *MIG* operations, each node acts selfishly on its own (whereas in the *NIXMIG* case waves drastically reduce load oscillations), and probing messages add an extra cost to the balancing procedure.

In [GS04], the authors present the Range Search Tree (RST) a structure similar to the segment tree data structure used in spatial databases and computational geometry [BCKO08] for efficient resolution of range queries. RST is basically a distributed segment tree data structure, where each peer is mapped to one or numerous nodes in the segment tree data structure. To tackle load balancing issues, they propose and utilize an organization of nodes into a logical matrix called Load Balancing Matrix (LBM). LBMs organize nodes to answer queries for a specific “hot” range by utilizing both node migrations (i.e., the range splits and a new node takes half of the keys) and replication (i.e., more than one nodes are responsible for the range subsets). LBMs automatically shrink and expand according to query load. The main drawback of RST is that it keeps the DHT hashing function that destroys locality, something that renders range queries (especially ones with high selectivity) quite ineffective. Although they employ segment tree algorithms to decompose a range query to smaller ones, they add unnecessary computational and algorithmic complexity to the query resolution procedure.

Similar to Mercury is the HiGLOB framework [VORT09]. In HiGLOB, each node maintains a list of load information about non-overlapping regions of the key space, and if it detects imbalances, it performs load exchanges. Non-overlapping regions of the key space constitute the load histogram, which is used by the node to perform load exchanges. HiGLOB is more interested in efficient (in terms of execution time and bandwidth consumption) histogram construction and maintenance. The authors of HiGLOB apply their approach in three different underlying overlays: the Skip Graph [AS07], the BATON [JOV05] and the Chord [SMK⁺01] overlay. Nevertheless, they also deal with load balancing: they make the distinction between static and dynamic load balancing. The former is triggered only with the insertion or deletion of new nodes, whereas the latter is triggered whenever a node is considered overloaded or underloaded. Nodes utilize their histograms to classify themselves as overloaded when their load is greater than twice of the average load of any group in their histogram or underloaded when their load is smaller than half of the average load of any group in their histogram. Dynamic load balancing is further classified in local (i.e., a NIX operation between neighboring nodes) or network (i.e., a MIG operation with one remote node) load balancing. HiGLOB also favors local to network balancing

actions, as during balancing, it first tries to perform local load balancing, and when this is not possible it resorts to network balancing. Nevertheless, the HiGLOB algorithms only examine the immediate neighbors of a node, something that resembles a *NIXMIG* wave of a TTL=1, whereas *NIXMIG* can examine a larger portion of both the local and the remote node neighborhood by setting a larger TTL.

The LIGHT [TZX10] system builds a binary tree called Space Partition Tree on top of a DHT so as to support complex queries such as ranges, min-max and k-nearest neighbor queries. The ID space is recursively partitioned in two equal-sized subspaces until each subspace contains fewer than θ_{split} items. In each partition split, the height of the created binary tree is increased. Actual data is stored only in leaf nodes in a structure called leaf bucket. Apart from the leaf bucket, each node is aware of a limited local view of the global partition tree called local tree. The local tree is actually the routing table of the leaf node and it is used to route queries. In terms of load balancing, the LIGHT system employs an approach inspired by the power of two choices of Mitzenmacher [Mit01] and its application to DHTs [BCM03], something that drops peer imbalance ratio from $O(\log n)$ to $O(\log(\log n))$. Their approach is called double-naming strategy: each leaf bucket can have two different names, which leads to two different possible physical node locations. The protocol then chooses from the two possible physical nodes the one which is less loaded to store the leaf bucket (hence the power of two choices). The problem with this approach is that in each key lookup two different queries are needed, as nodes are unaware of the final storage location of the leaf bucket.

In Armada [LCLC09], a general range query scheme on top of a DHT is presented. Armada splits the ID space into partition trees in a similar way than [TZX10]. Armada's load balancing is performed with a hash function responsible for placing items into nodes that knows in advance the distribution of items in the ID space. This method is called ObjectID balancing. In this method the partition tree is built taking into account the distribution of indexed items. As of this, the distribution is considered static, and each node needs to know in advance the item distribution of the total network. Nevertheless, this distribution is difficult to be calculated in a distributed way, since a single node cannot have a full and fresh picture of the network load, which may vary over time.

Shen and Xu [SX07, SX08] distinguish peers between simple ones and super-nodes. Super peers are dynamically elected according to their capacity and connection bandwidth. Peers form two different types of overlays: physical clusters (called pClusters) in which nodes are connected to physically close peers and logical clusters (called lClusters) in which nodes are connected to logically (i.e., DHT-ID wise) close peers in the overlay. Physical proximity is calculated by comparing distances to some special predefined landmark nodes. These auxiliary overlays are used to summarize load information between physical or logical clusters (i.e., neighborhoods). Each cluster elects a super peer. Super peers are used for routing messages and queries on behalf

of the cluster's nodes, whereas simple nodes utilize their super peer as a proxy to the entire network. Each super peer has a pair of donating sorted list (DSL) and starving sorted list (SSL) which store load information about peers in its cluster. The SSL contains a sorted list of the most loaded objects stored by overloaded servers, and the DSL contains the least loaded servers. The balancing is then performed in three steps: First, local nodes periodically update their load information to the super peer, so that DSL and SSL are up-to-date. Second, a super node with a non-empty SSL tries to match local overloaded peers with local underloaded ones (i.e., top SSL with top DSL entries). This procedure is called local load balancing. If the load cannot be absorbed locally, then the super peer moves to the third phase, the global load balancing. In this phase, it randomly contacts other super peers to find non-empty DSLs until it empties its own SSL. In general, balancing is performed by moving "hot" items and placing doubly-linked pointers both to the source (overloaded peer) and the destination (underloaded peer) of the moved item. The drawback of this method is that during lookups the overloaded peer will still be contacted, as it is still responsible for this "hot" item. Therefore, their approach is useful only when nodes are overloaded in terms of storage space, since the moving of items release used storage space. Nevertheless, when there is a data access imbalance problem, their approach not only does it fix the problem, but it makes it even worse: overloaded nodes are still contacted and an extra hop is added to the query resolution so as to locate the final object holders.

Finally, in chordal graphs [Jou08] balancing is performed by a process called "free drifting" which is actually a *NIX* operation. Nodes periodically compare their load with their immediate neighbors in the overlay (i.e., their forward and backward nodes) and if they detect imbalances they perform load exchange with them by issuing a simple *NIX* operation. The problem with this approach is the difficulty to detect global imbalances (as for instance in the histogram method of Mercury [BAS04] and Oscar [GDA07, GDA10]) since only two nodes are used for load comparison. Moreover, the use of only *NIX* operations has the disadvantages experimentally observed in Section 4.5.1 and theoretically studied in Section 4.2.3

Node virtualization approaches

The idea of virtual servers for load balancing in peer to peer systems was initially proposed in CFS [DKK⁺01]. CFS is a distributed file system based on the Chord DHT, implemented by the same team that developed Chord. CFS tackles inherent load imbalances caused by the DHT assignment of items to nodes which, according to the consistent hashing algorithm [KLL⁺97] is $O(\log n)$, where n is the number of participating nodes. What is more, CFS balancing also takes into account node heterogeneity caused by different hardware and network infrastructures. CFS addresses these issues by assigning virtual servers to an actual one according to its capacities and its current load status. When an actual node is overloaded or underloaded, virtual nodes are

removed or added accordingly. A virtual node's address is, for instance the hashed value of the concatenation of the actual server's IP along with the virtual node's index number. Messages are routed in the virtual node layer.

Based on the idea of CFS, Rao et al [RLS⁺03] proposed three load balancing algorithms (One to One, One to Many, and Many to Many) which were extended by Surana et al [SGL⁺06] for heterogeneous peer to peer systems with churn. In the first case, an overloaded node contacts one node at random (as in the work of Karger and Ruhl [KR06]) while in the second case it contacts numerous nodes before it takes a balancing decision. The third case is similar to the approach used by Ganesan et al [GBGM04]: a distributed directory with load information is maintained and contacted by overloaded peers before any balancing decision is taken.

In Y_0 [GS05], Godfrey and Stoica tackled the problem of multiplication of open routing links per server by placing virtual servers owned by an actual one "near" themselves in the ID space. As of this, distant routing links are kept only from the "first" (i.e., the node with the smallest id) and the "last" (i.e., the node with the largest id) virtual node. Virtual nodes between the first and the last do not keep routing tables, as they can be reached through the first and the last virtual node. They make the assumption that the load is uniformly distributed in the identifier space and each one from the total n actual servers that hosts k virtual servers gets a random continuous fraction of $\Theta(\frac{k}{n})$. Nevertheless, as we discussed earlier, uniform load distributions are not frequently encountered in order preserving structures. What is more, it has been shown that with only one ID per actual server balancing results are the same as in the case of node virtualization (see related work of [Man04]).

Zhu and Hu [ZH05] also build and maintain a distributed load directory in the form of a k -ary tree structure that is stored in the overlay. This directory is used by nodes to detect load imbalances and to find suitable overloaded-underloaded node pairs. The directory is basically a tree in which each tree node is responsible for a specific region of the ID space. Children are responsible for load estimation and dissemination of small disjoint and consecutive areas of the ID space, whereas their parent is responsible for the total area, and so forth. Tree parents request load information from their children: requests traverse the tree from top to bottom, leaf nodes request load information from their virtual nodes, and information is then pushed back from bottom to top. This information is used by each node to identify whether it is overloaded or underloaded, and to find suitable helpers in the case of high load. The algorithm then decides peer matchings so that the minimum amount of item transfer is performed. The authors also consider proximity during assignments with the use of landmark nodes as in the work of [SX07, SX08].

In [HLCH11] the authors propose a novel balancing scheme by utilizing virtual nodes. Their method is novel, since no auxiliary network or an hierarchical scheme for virtual to actual server assignment is used. Instead of this, actual nodes obtain a partial view of the system's load state, and based in this, they perform node reassignment in parallel. Each peer gathers a partial view

of the system state by randomly contacting a logarithmic number of nodes with the use of random walkers. The samples are used to calculate an approximation for both the nodes loads and capacities probability distributions. The authors calculate the optimal number of samples so as the approximation error is bounded between a predefined value. What is more, they experimentally compare their algorithm with the many to many approach with a single directory proposed in [SGL⁺06] and with [ZH05] where they identify weaknesses and strengths of each system.

Chen and Tsai [CT08] use the general assignment problem (a particular case of a linear programming problem) to assign virtual to actual nodes: they make an initial estimation using the ant system heuristic which afterwards is refined using the descent local search algorithm. This procedure is iteratively applied until a solution is reached.

In Armada [LCLC09], the authors use virtual servers for balancing purposes but they do not provide details about their specific implementation.

4.7 Conclusions

In this chapter, the performance in terms of bandwidth cost and convergence speed of balancing range queriable data structures using successive item exchanges or node migrations was evaluated. Extensive experimental results showed that none of these methods by itself is capable of efficiently balancing arbitrary workloads: Neighbor item exchanges are expensive in terms of item transfers and slow in terms of convergence speed, whereas node migrations are fast but costly in terms of message exchange. Based on these findings, *NIXMIG*, a hybrid approach that adaptively decides the appropriate balancing action was proposed and evaluated. Load moves in a “wave-like” fashion until it is absorbed by underloaded nodes, and node migration is triggered only when it is necessary. Furthermore, several improvements to the original *NIXMIG* algorithm that further decrease its bandwidth utilization during balancing operations were proposed. The developed simulation was used to experimentally compare its performance with other algorithms. Results show that *NIXMIG* can be three times faster, while requiring only one sixth and one third of message and item exchanges respectively to bring the system in a balanced state under a variety of skewed, dynamic and realistic workloads.

A Distributed Framework for Indexing and Serving Large and Diverse Datasets

In this chapter, a distributed architecture for indexing and serving large and diverse datasets [KATK10] is presented. It incorporates and extends the functionality of Hadoop, the open source MapReduce [DG08] framework, and of HBase [Apa11d], a distributed, sparse, NoSQL database, to create a fully parallel indexing system. Experiments with structured, semi-structured and unstructured data of various sizes demonstrate the flexibility, speed and robustness of this implementation and contrast it with similarly oriented projects. The 11 node cluster prototype managed to keep full-text indexing time of 150GB raw content in less than 3 hours, whereas the system's response time under sustained query load of more than 1000 queries/sec was kept in the order of milliseconds.

The remainder of this chapter is organized as follows: the basic system architecture is presented in Section 5.1, experimental results are detailed in Section 5.2, in Section 5.4 we discuss some design decisions, while Related Work and the Discussion Section conclude the chapter.

5.1 Architecture

A distributed processing platform suitable for indexing, storing and serving large amounts (in the orders of TB and more) of content data under heavy request loads is presented. Indexing rules of variable granularity, relative to both type of data and user-input, along with the raw

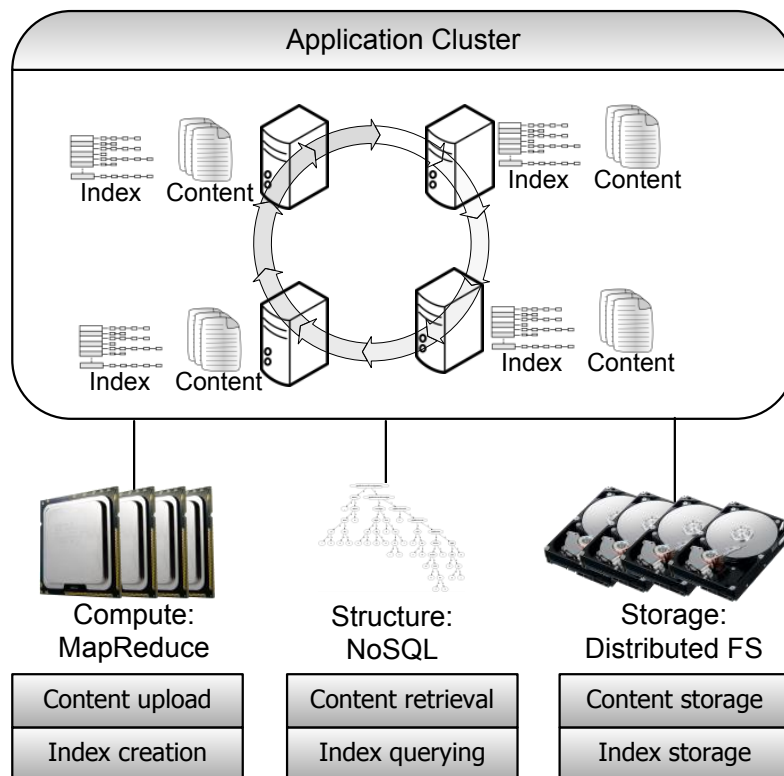


Figure 5.1: Combining *MapReduce* with *NoSQL* to create a fully distributed indexing system.

content are processed by the framework. The augmented information is extracted in the form of a distributed index served to an arbitrarily large number of concurrent users. In order to speed up indexing, the compute and storage-intensive index creation and maintenance leverages Hadoop [Apa11b] the open source version of the innovative *MapReduce* [DG08] framework. To achieve low response times in high query load using commodity-node clusters, user requests are served through an *HBase* database, the open source alternative of Google's Bigtable [CDG⁺08].

In Figure 5.1 we depict a platform overview. We notice that our framework distributes both the content and the index. Hadoop's *MapReduce* framework is employed both during content upload and index creation so as to efficiently harness the cluster's computational resources. For content retrieval and index querying we utilize *HBase* since it is tightly coupled with Hadoop, it has one of the biggest open-source communities in the area of *NoSQL* and it is used by a big number of companies and organizations worldwide. As our back-end storage for both the Index and the Content we utilize Hadoop's Distributed File System (*HDFS*) that provides a unified view of the cluster's local storage.

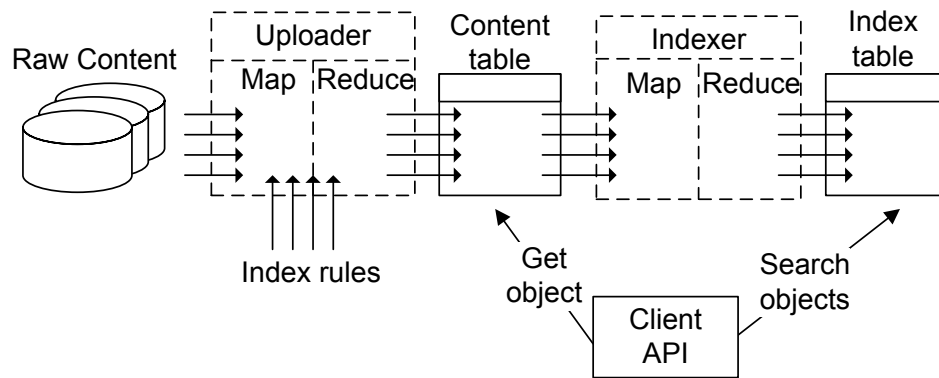


Figure 5.2: System Architecture

The system is built on top of *Hadoop* [Apa11b], an open source java implementation of the MapReduce paradigm that has been widely adopted by a large number of organizations worldwide [Apa11c]. Hadoop consists of a distributed file system called HDFS (a GFS [GGL03] - like distributed file system) and a MapReduce [DG08] executing framework on top of it. HDFS file metadata is being kept in a single node called the NameNode and raw data is stored in many DataNodes. MapReduce consists of a job scheduler called JobTracker which co-ordinates many worker nodes called TaskTrackers. In a typical setup, the NameNode acts as a JobTracker, and DataNodes are also TaskTrackers.

HBase is used as the NoSQL storage substrate, which is an open source implementation of Google's Bigtable [CDG⁺08], since it is tightly coupled with Hadoop (it uses HDFS as its storage backend and provides I/O hooks to Hadoop's MapReduce framework). An HBase table consists of a large number of sorted rows indexed by a row key and columns indexed by a column key (each row can have multiple different columns). Actual content is stored in HBase cells: an HBase cell is defined by a combination of a row and a column key, in the same way an (x,y) value defines a point in a 2-dimensional space. The primary key of an HBase table is the row key. HBase supports two basic lookup operations on the row key: exact match and range scan. HBase consists of a single master (HMaster) that keeps track of numerous nodes that serve actual content called RegionServers.

Overview: In Figure 5.2, an overview of the system components along with their interactions are presented. The main idea is the following: The raw content (in the form of large XML files, HTML files, SQL database dumps, logfile directories, etc) is submitted to HDFS. The content along with some instructions (the indexing rules) is fed to the Uploader, which is a MapReduce program. The Uploader creates an HBase table with the content in a record oriented view. A second MapReduce task (Indexer) takes as input the Content table, and extracts the Index, which is also stored as an HBase table. Users perform queries using the client API. The API contacts

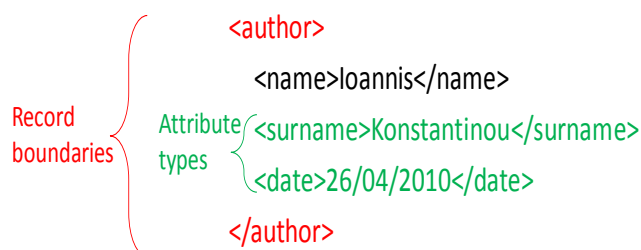


Figure 5.3: Indexing rules. Record boundaries split input into distinct entities and attribute types define record regions to index.

Table 5.1: Content table example. Row key is MD5Hash of the content, and cell content is the record data.

Row key: MD5Hash	Row value: Record Content
2da0ae7cb598ac8e 9455570a9c2f19fe	<author><name>Ioannis</name><surname>Konstantinou </surname><date>20100424</date></author>
223c14b2a8c7bbe2 4ba0d6854dd6f3cc	<author><name>Evangelos</name><surname>Konstantinou </surname><date>20100426</date></author>

the Index table to perform searches, and the Content table to serve objects. In the following, a detailed presentation of the system components is given.

Index rules: According to the content type, users provide the system with instructions of what to index. In this phase, there is also a need to specify what are the boundaries of a single “record”. Records are used to split the entire dataset in a number of distinct entities that will act as processing units. For instance, in the XML case, record boundaries can be considered some specific tags. In the unstructured content situation (e.g. in HTML files) a specific record can be a single HTML document, whereas in the database case, records are table rows. It is important to note that our system can adjust the granularity of the “record” according to the user/application requirements. Apart from this, users also select what specific content regions (attribute types) they want to index (such as text contents of a specific XML tag, contents of an HTML table or paragraph, data from a specific table or column from a database, etc). Index rules are used from every component (Uploader, Indexer and the client API).

In Figure 5.3 we present an example of the indexing rules in the XML case. The records are the `author` XML tags, whereas the indexed areas are the `surname` and `date` tags. Note that in this situation the tag name is left unindexed.

Uploader: The Uploader class reads bulk datasets previously uploaded to HDFS, and creates the Content table. The Content table acts as a content hashmap, where every row contains a single record item and the row key is the MD5Hash of the record content. The content table is in essence a forward index of the total document list. The Uploader class reads data input from HDFS and creates the content table using the MapReduce paradigm as follows: In the

Table 5.2: *Index table example. Row key is the index term followed by the attribute type and row content is the list of the MD5Hashes of the documents that contain this keyword in this attribute.*

Row key: “term”_“attribute type”	Row value: list(MD5Hash)
20100424_date	2da0ae7cb598ac8e9455570a9c2f19fe
20100426_date	223c14b2a8c7bbe24ba0d6854dd6f3cc
Konstantinou_surname	2da0ae7cb598ac8e9455570a9c2f19fe, 223c14b2a8c7bbe24ba0d6854dd6f3cc

Map phase, mappers read from HDFS and emit a list of <key, value> objects where, the key is the MD5Hash of the record, and the value is an HBase cell containing the content. The reduce phase lexicographically sorts the incoming <MD5Hash, HBase cell> key values according to the MD5Hash, stores the results in HFiles (HBase data file format), and informs HBase of the location of the new table data files. The use of the content table is twofold: first, it allows fast random access reads during successful index searches. Second, it enables easy content manipulation in the case of new item additions or deletions of old records. An example of a Content table can be seen in Table 5.1: here, records are the XML tags named “author” and attribute types are the tags “name”, “surname” and “date”.

Indexer: The Indexer module calculates an inverted list of index terms and document locations and stores it in the Index table. The row key of the index table (primary key) is the keyword term followed by the attribute type on which this keyword was encountered (e.g., if the keyword `google` was found in a <revision> tag, then the row key will be `google_revision`). Every row stores a list of MD5Hashes of the records that contain this specific keyword (e.g., `google`) in a specific attribute (e.g., `revision`). This list actually links the Index to the Content table. The Indexer is a MapReduce task that works as follows: in the Map phase, mappers process the Content table, and emit a <keyword_attribute, MD5Hash> key-value pair for every keyword they encounter. Reducers receive all emitted key-values for a specific key and aggregate them in one key-value pair of the type <keyword_attribute, list(MD5Hash)>. These key-value pairs are finally stored in HDFS (HFile format), and HBase is informed about the new table. In Table 5.2 we present the inverted index created by the content table 5.1, after selecting to index only the attributes “date” and “surname”.

The reason for bypassing the HBase API during initial table creations is that it is many times slower for bulk insertions, as it is explicitly stated by the HBase developers. This happens because insertions are first written to a persistent write-ahead-log and are afterwards accumulated in a memstore (a sorted, in-memory buffer). When the memstore is full, they are flushed to the disk in HFiles. By directly storing HFiles, we avoid expensive intermediate interactions with the write-ahead-log and the memstore for each new object.

Client API: The client API provides the basic search and get operations and it is built on top of HBase's client API. Our index design allows us to perform google-style freetext queries over multiple indexed attributes. Users are able to search content in a specific attribute type (e.g., find all documents that contain the keyword `google` in the `<title>` tag) or in any attribute type. In the first case, the user query is translated in an HBase point query with the parameter `"google_title"`, whereas in the second case it is translated in a range scan in all the HBase rows that start with the prefix `"google_"` (one for each attribute type), which by default are lexicographically adjacent. More complex range queries of the type: "find all documents that have a creation date in 2009" or prefix queries such as "find all the documents that contain a keyword `goo*`" are also supported and are translated in an HBase range scan. In any case, the client contacts HBase once for every query, even if this is a range scan, and the network overhead between the client and HBase is only one round-trip message per query. Query results consist of a list of MD5Hashes of the matching documents that can be retrieved with a simple lookup from the Content table. Our system also enables AND-ing and OR-ing of queries through client side processing: queries are executed for each dimension and query results are merged by the client to provide the final list of the matching documents.

5.2 Experimental Results

Our experimental setup consists of 11 worker nodes and a single machine in the role of HDFS, MapReduce and HBase master. The worker nodes have 2 Quad-Core E5405 Intel Xeon® CPUs @ 2.00GHz, 8 GB of RAM (with disabled page swapping) and a 500GB disk (for a total of 88 CPUs, 88 GB RAM and 5.37 TB of disk space), while the master has similar CPUs and disk, but only 2 GB RAM. The version of Hadoop is the latest 0.20.1, re-compiled from source to suite our setup. HBase version is 0.20.2. The network infrastructure for the nodes and master consists of a single Gigabit Ethernet Cisco Catalyst 2960g switch. All machines are running x64 Debian Linux using SMP kernel 2.6.24.2, compiled from source and Sun Java 6™.

The Hadoop MapReduce framework is configured to take full advantage of the available resources. Hadoop and HBase are each given 1 GB of memory in every running machine, and each Mapper or Reducer task is given 512 MB of RAM. Each worker node can spawn 6 Mappers and 2 Reducers running concurrently, for a total cluster capacity of 66 Mappers and 22 Reducers. We have disabled Hadoop's speculative execution, where every task is executed 3 times for redundancy, as this would drop the cluster's effective capacity to one third. HBase is managing its own Zookeeper (an Apache project providing a distributed consistency system) instance with 3 quorum nodes. HDFS was configured with a replication factor of 2.

Table 5.3: *Index size and creation time for different numbers of attribute types (5GB HTML).*

Iteration No	Indexed tags (count #)	size GB	time min
1	[table,li, p,b, i,u, title] (7)	1.049	7
2	1 + [h1, h2, h3, h4, h5, h6, big] (14)	1.097	6.5
3	2 + [blockquote, del, em, s, small] (19)	1.117	9.5
4	3+ [strong, sub, sup, tt, pre, dt, dd, font] (27)	1.296	11

Table 5.4: *DB Index creation time vs number of nodes.*

# of Nodes	time(min)
2	34
4	23
6	16
8	11
11	10

During MapReduce execution, the framework automatically sets the number of Map tasks. To enable full cluster CPU utilization, Reducers for all jobs were manually set to 100. Any number beyond the cluster concurrent capacity of 22 would be sufficient, but a larger number minimizes the effect of failed reduce tasks on job completion time. If the job input comes directly from HDFS, the number of spawned Mappers defaults to the number of the input's 64MB HDFS chunks. Similarly, in the case of HBase this number defaults to the number of HBase's 64MB regions (equivalent to Bigtable's [CDG⁺08] tablets).

Our datasets were downloaded from Wikipedia's dump service [Fou11b] and from project Gutenberg's custom DVD creation service [Gut11]. Our structured data comprises of a 23GB MySQL database dump of the latest English version of Wikipedia. The structured dataset was obtained from the current MediaWiki XML dump available at the Wikipedia download site with the use of `mwdumper` [Fou11a] and uploaded to a local MySQL 5.0.51 database instance, from which a new SQL dump was obtained to form the basis for our experiments. The reason for this process was our desire to have a dataset consistent with actual MySQL dumps.

Our semi-structured dataset comprises of one XML and one HTML dataset: the XML dataset is a 150GB part of a 2.55TB uncompressed XML dump of every English Wikipedia page along with its revisions up till May 2008. The HTML dataset in turn is a 150GB dump containing a static version of Wikipedia from June 2008.

Our unstructured data is a full dump for all languages of Gutenberg's text document collection. The dataset comprises of approximately 46,300 text files, that take 20 GB of hard disk space.

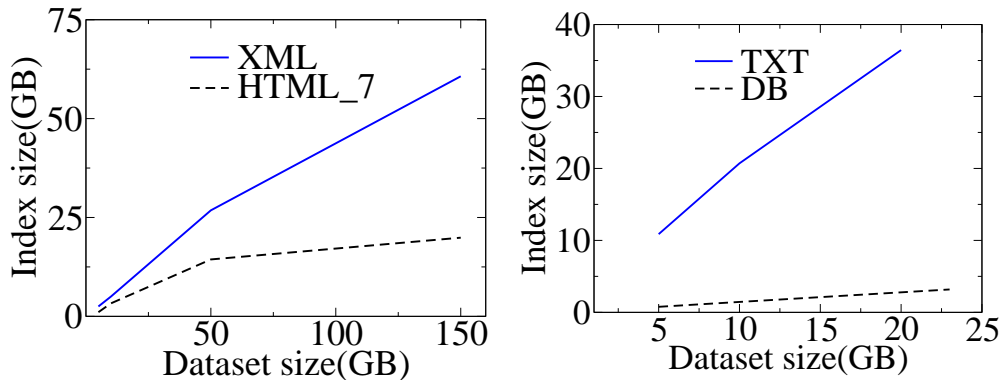


Figure 5.4: Index size for various datasets

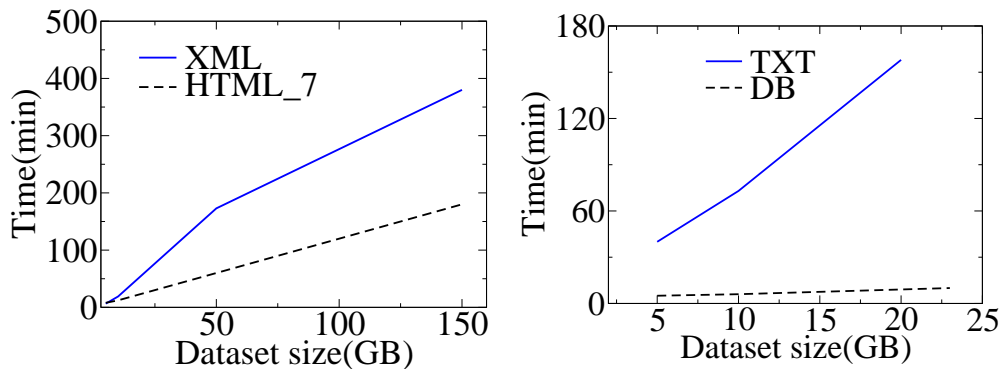


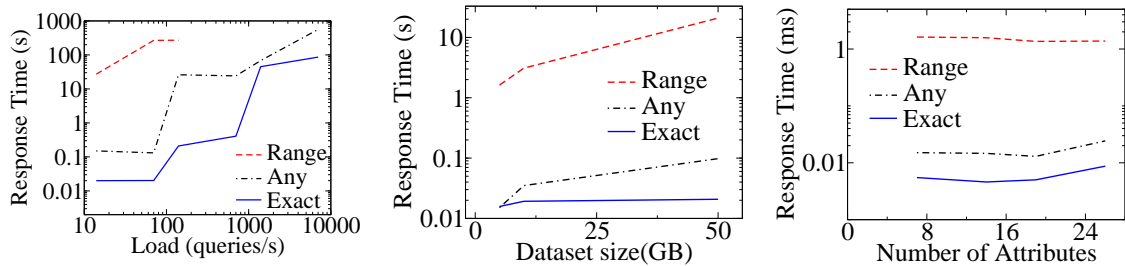
Figure 5.5: Index time for various datasets

To locate attribute types during indexing creation, in the structured (database dump) and the unstructured (text files) case we utilized simple regular expressions, whereas in the semi-structured (XML and HTML) case Xpath expressions were used. In the case of HTML, all documents were filtered through `TagSoup` [Tag11], a parser that converts HTML to well formed XHTML documents.

In order to create query traffic, we utilized a publicly available dataset from AOL that contains twenty million search keywords for over 650,000 users over a 3-month period [Arr06] to calculate a zipfian query frequency distribution. During our experiments, clients were generating both point and prefix queries based on that distribution. The advantages of the AOL keyword dataset compared to a random keyword generator is that it follows a real-life, non-uniform skewed distribution, where popular keywords are requested more often. The experiments try to clarify the performance of our indexing system under heavy user load when using a reasonably large number of structured, semi-structured and unstructured data.

Table 5.5: Content table creation time for various dataset sizes and types

XML		HTML		DB		TXT	
size GB	time min	size GB	time min	size GB	time min	size GB	time min
5	7	5	9	1	3	1	3
10	44	10	42	6	9	5	15
50	192	50	202	12	12	10	33
150	576	150	601	23	20	20	70

**Figure 5.6:** Response time of queries vs system load in queries/s, original data size and number of indexed attributes.

5.2.1 Content table creation

In this section, we present our findings during the content table creation procedure. In Table 5.5 we depict the time it took for the Uploader to create the content tables from raw HDFS data for bulk insertion into HBase. The tests were run using all the available nodes.

From Table 5.5 we can deduce that our system exhibits the expected behavior for bulk insertion of the dataset. The time to completion increases linearly with the size of the dataset and is comparable between different types of data. The diversion from the norm of the structured dataset can be explained by taking into account the reduced processing required for the SQL dataset. Similarly, the larger than expected time to completion for the plain text Uploader is justified, given the fact that it is composed of a large number of very small files. Thus, the initialization penalty for each Mapper is high compared to the processing, and though small (about 1 second), makes a significant difference given the average time to completion for each file (2-5 seconds).

5.2.2 Index table creation

In this section, we present our experiments during the index table creation. We are interested in the index size and creation time. Index tables are created from the content tables described in

Section 5.2.1. In Figure 5.4 we present the growth of the index size as the content table increases. The first figure presents index growth for the HTML and XML dataset, whereas the second figure depicts index growth for the DB and TXT dataset. In Figure 5.5 we present the indexing time of the aforementioned four dataset types for various sizes. In the HTML_7 case, the indexed attributes are the first 7 HTML tags from Table 5.3, whereas in the XML case, `<content>` and `<timestamp>` tags are indexed.

In Figure 5.4 for the XML and HTML inputs we notice that in any case the growth of the Index table gets smaller with the increase of the dataset: this happens because, after a certain Content size, indexed terms size is not significantly increased. What is more, we notice that XML index is larger than the HTML index of the same dataset size: HTML dataset contains a lot of formatting code that gets stripped during keyword extraction, whereas in the case of XML there is (almost) clean text. In the case of DB and TXT, the variations between structured and unstructured data as to the size are justified, as the structured dataset is already indexed in a reasonable way. Moreover, the diversity between the different documents contained in the Gutenberg collection (in terms of language and topic) means that a lot more terms have to be indexed, increasing the size of the index because of the added metadata.

In Figure 5.5 for the XML and HTML we notice that the XML dataset is more demanding in terms of processing time compared to the HTML dataset, because of the HTML formatting code that gets stripped during indexing. In the TXT and DB case, the structured DB dataset is the most easily indexed, as the system does not perform much re-indexing except for re-arranging the data to fit its internal specifications. The TXT dataset obviously requires much more time, as more processing is needed to extract and process metadata. Both datasets however exhibit near linear scalability as the number of data increase, but with different angles as explained before. This is a desirable outcome for the Indexer, as it highlights its robustness and good behavior under these tests.

We now present experiments that show how our indexing mechanism responds when we vary the number of the attribute types. We utilize the 5GB HTML dataset and we increase the number of attribute types in every iteration, as we depict in Table 5.3: in every iteration, all the tags from the previous iterations are included. In Table 5.3 we present the growth in the indexing time and size for every iteration. As expected, both the size and the index time increases along with the attribute type number. Nevertheless, this increase rate remains relatively low.

In the following experiment, we measure the scalability of the indexing mechanism by varying the number of the cluster nodes while keeping a stable index input dataset size. We run the Indexer on the largest DB dataset and in Table 5.4 we present the index creation time variation. As expected by the fully distributed indexing nature of MapReduce, the speed is proportional to the number of processing nodes, something that proves the system's scalability during index

creation. This property is a typical cloud application requirement, since extra nodes are acquired by a cloud vendor in an easy and inexpensive manner.

5.2.3 System performance under query load

In this section we measure our system's ability to respond to a large number of simultaneous user queries. We consider three types of queries on indexed attributes: free keyword search in a specific attribute, free keyword search in any attribute and prefix queries in any attribute type. Prefix queries were generated using the first four characters of randomly selected AOL keywords. The test obtains response times from HBase for these search types. Client instances were run concurrently on 14 machines, to ensure realistic measurements in terms of network load from different machines. Most of the tests were executed using a 14GB index table of a 50GB HTML dataset subset. For the response times vs data size, 5, 10 and 50 GB partitions of the HTML dataset were used with 7 indexed attributes (as seen in Table 5.3).

Our first experiments evaluate the maximum load measured in queries/s that our index table can support in HBase. We have observed the limit of HBase when serving queries by running 14 concurrent clients, each sending queries with a delay that follows an exponential distribution probability function. Our experiments (results seen in Figure 5.6a) were run with values higher than the previously reported limit for sustained queries per second against HBase on HDFS [LRST09]. Although high response times were measured, the system remained stable and kept serving requests even under heavy load, highlighting the robustness of HBase. Range query loads above 140 queries/s however failed to complete and in most cases the clients had to be manually terminated. This is reasonable, as such queries request a large number of data and demand processing on both the server and client, increasing exponentially the load on available resources. For reasonable load on the server, in the order of 14 queries per second from different clients, we have measured response times close to 20 ms for point queries, 150 ms for any attribute queries and 27 seconds for range queries.

We believe that the observed behavior of the system in Figure 5.6 is a consequence of HBase caching: up to 100 queries/s there are available channels to accommodate the clients. Beyond this point, the response time increases because of the increased client requests, who now have to wait in line to be served. Between roughly 100 to 1000 queries/s, HBase caching is significant: each popular keyword is loaded only once in memory and then served as is. This leads to a significant decrease of the average response time of the system because of the skewed distribution of the requests. This tactic fails for load above 1000 queries/s and the average response time for queries increases exponentially.

Running queries for datasets of different sizes, shown in Figure 5.6b, the response times follow an expected pattern. The tests were run with an average load of 14 queries per second for

the system, i.e. 14 clients issued on average one query per second. The choice was made to ensure that range queries would be included in our results. Range queries are the most expensive, and therefore response times are larger, as are searches for a specific keyword in any indexed attribute. Response times for point queries (exact matches of keyword and attribute) remain relatively constant, irrespective of dataset size, while response time for range and any attribute queries increases slowly as the number of records containing them increases with the dataset size.

An increase in the number of attributes (Figure 5.6c) has no effect on the response time of point queries, and this highlights the efficiency of the system. Similarly, times for range queries and queries in any attribute scale almost linearly, keeping the overhead small even for large indices. This might seem unexpected, as for most range queries, it would mean a theoretical increase of factor 27. This is not the case due to the structure of HTML: since the expected nesting depth of HTML tags is low, the number of query results returned by such queries rarely reaches this theoretical maximum. However, since the selection of indexed attributes is left to the user, this behavior could theoretically be simulated by choosing consistently nested tags (e.g. table, tr and td in HTML). Given the limited usefulness of such a selection though, we believe that it is not a concern for practical system use.

5.3 Related Work

The requirement to perform compute-intensive analytics on (semi) structured bulk datasets has pushed sql-like centralized databases to their limits [Aba09]. This fact, along with the highly parallel nature of these tasks, has led to the development of horizontal scalable, distributed non-relational data stores, called NoSQL databases. Google's Bigtable [CDG⁺08] and its replacement Megastore [BBC⁺11], Amazon's Dynamo [DHJ⁺07], Facebook's Cassandra [LM10], IBM's CouchDB [Apa11a] and LinkedIn's Voldermort [Vol09] are a representative sample of such systems. The interest in such data-stores is so big that more than 120 different implementations have been proposed according to the NoSQL site [NoS09] retrieved in April 2011. In favor of scalability and high availability, NoSQL systems relax classic ACID guarantees made by typical DBMS, allowing, for instance, only eventual consistency. NoSQL systems serve a dual purpose: they can efficiently store and index arbitrarily big data sizes while enabling a large amount of concurrent user requests. NoSQL systems are perfect candidates for cloud infrastructures, as their shared nothing architecture enables them to scale by simply acquiring more computational and storage resources from a cloud vendor.

The distributed cooperation of a large number of computational and storage resources is a challenging task. Application specific requirements of large scale data management tasks (e.g. the need to push computation near the data) prohibit the use of typical general purpose job

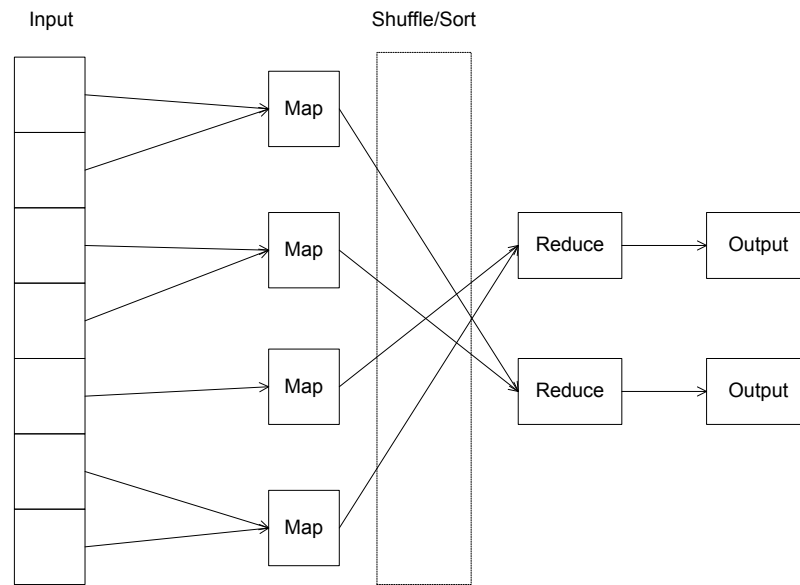


Figure 5.7: Overview of a simple MapReduce operation. Input chunks are fed to a number of mappers, intermediate output is sorted and assigned to a usually smaller number of reducers which produce the final output.

schedulers. To cope with these requirements, “data-aware” distributed data management frameworks have been proposed, with Google’s MapReduce [DG08] as the most prevalent. MapReduce is inspired by the typical “map” and “reduce” functions found in Lisp and other functional programming languages: a problem is separated in two different phases, the Map and Reduce phase. In the Map phase, non overlapping chunks of the input data get assigned to separate processes, called mappers, which process their input and emit a set of intermediate results. Intermediate results are automatically sorted in the shuffling phase. In the Reduce phase, these results are fed to a (usually smaller) number of separate processes called Reducers, that “summarize” their input in a smaller number of results that are the solution to the original problem. In Figure 5.7 we present the execution of a simple MapReduce job. For more complex situations, a workflow of map and reduce steps is followed, where mappers feed reducers and vice versa.

5.3.1 MapReduce based data analysis frameworks.

On top of MapReduce, a number of frameworks have been proposed to facilitate the management and execution of data warehousing tasks. Yahoo’s Pig [ORS⁺08], Facebook’s Hive [TSJ⁺09] and HadoopDB [ABPA⁺09] are a representative subset of such frameworks.

Pig [ORS⁺08] users write their analytical jobs in a declarative scripting language, and Pig translates them to MapReduce jobs in Hadoop. Pig aims to simplify the creation of MapReduce

jobs by enabling code re-usability. Data objects consist of atoms (simple value), tuples (collections of atoms), Bags (collections of tuples) and Maps (key->value pairs, where keys are atoms and values are any of the aforementioned types). Pig loads data and saves output using custom Serializers/Deserializers through its LOAD and SAVE commands respectively. Data manipulation is performed through its commands (FOREACH, FILTER, COGROUP, UNION, CROSS, ORDER and DISTINCT). Each command is actually a MapReduce program. Pig enables nested data (e.g. tuples inside of tuples) and nested commands (e.g. FILTER inside of FOREACH commands).

Hive [TSJ⁺09] offers an SQL-like declarative language called HiveQL. User queries are transformed to a chain of MapReduce steps through its query optimizer component, and Hadoop's HDFS is used as its storage substrate. Hive stores its data in HDFS directories in an hierarchical manner (databaseName/ tableName/ columnName/ PartitionID). Table metadata, called HiveMetastore, is stored outside of HDFS in a random-access optimized storage, such as a database or a local file-system. Both Pig and Hive are not optimized to serve large number of concurrent requests in a low-latency manner, since queries are executed as MapReduce jobs (completion time in the order of seconds) and data is served through HDFS, which is not designed to facilitate a large number of concurrent reads.

HadoopDB [ABPA⁺09] is a MapReduce and database hybrid that aims to bridge the gap between parallel databases and MapReduce systems: it uses Hadoop as its communication layer with the difference that DataNodes are also running local instances of a DBMS (PostgreSQL or MySQL) that store the actual data (in contrast to Pig and Hive that store data directly to HDFS). HadoopDB also extends Hive's query optimizer component to "understand" local DataNode databases, but queries are also executed as MapReduce tasks.

SCOPE [CJL⁺08] is Microsoft's SQL-like scripting language for distributed analytical tasks. SCOPE is executed on top of Cosmos, a distributed storage and execution engine.

A number of content analytic platforms built on top of Hadoop have been proposed recently. In [BEK⁺09], IBM presents a preliminary version of their work on analyzing large datasets using Hadoop, where they do not deal with serving the created content. Their platform utilizes SystemT, an information extraction engine which extracts structured information from unstructured data and Jaql, a general purpose data-flow language that process semi-structured information as abstract JSON values. Jaql queries are translated into MapReduce steps and process unstructured data which is augmented with structured information (in the form of JSON values) from the SystemT engine. Both Jaql and SystemT are used in IBM's ES2, an enterprise semantic search engine. ES2 uses Nutch [Apa11f], the open-source web crawler project from which the Hadoop was created. ES2 process a company's intranet, and extracts an index which is used to perform "smart" queries.

5.3.2 Distributed indexing frameworks based on NoSQL and MapReduce combinations.

Distributed index creation frameworks that exploit the parallelism of Hadoop and HBase have been recently announced. Ivory, [LMEW10] an inverted indexing framework implementation on top of Hadoop, distributes the index creation through a MapReduce job, and the index is served through two different ways: a centralized repository (i.e., from a web server's disk) and a custom distributed repository based on HDFS. The partition of the index is term based, i.e., each node serves a subset of the index terms and the full document list of a term is stored in one node. In the HDFS case, the authors launch a "dummy" MapReduce program with only a Map phase, and each Mapper executes as a resident TCP/IP server which handles queries that search keywords from the local HDFS index shard. User queries are directed to the Partition Server, where mappings of keywords to HDFS files containing postings are stored (Ivory utilizes Metzler's [MC05] retrieval engine during index creation and querying). The Partition Server then connects to the resident TCP/IP server of each HDFS Node, gathers the required information, and returns the answer to the user query. The drawback of this method is the overkill of a resident TCP/IP server in each HDFS Node: instead of this, a NoSQL storage substrate could do the job with less programmatic effort and with the same efficiency both in terms of performance and query precision.

On the other hand, HIndex [LRST09] serves indices through HBase, but the index creation is centralized. In HIndex, the partition of the index is document based, i.e., each node is responsible for a subset of the documents and keeps a "local" inverted index list. The drawback of this approach is that even a single keyword query needs to be sent to all the participating peers, as it is not known in advance whether a node stores documents with this specific query. Nevertheless, this method is more efficient during document updates: when a new version of a document is inserted, only the node that is responsible for that document will be contacted and its local index will be updated, whereas in the term based indexing, a larger number of nodes that index terms of both the new and the old document will be contacted. What is more, HIndex tries to avoid a query flooding to all nodes by carefully selecting document ids and by exploiting HBase's ordering. HIndex utilizes Lucene [Apa11e], a popular open source text index library, as its local inverted index. HIndex stores lucene indexes as "special" HFiles after changing HBase's source code. HIndex has the one-server bottleneck problem during bulk insertions as discussed in [BMBB10]. The authors tackle this with the manual creation of numerous HFiles in different nodes, nevertheless they still do not bypass the slow HBase API. The main drawbacks of HIndex is the query flooding during searches and the inherent difficulty in bulk insertions.

The authors of [BMBB10] present a general purpose bulk insertion mechanism that is built on top of HBase and takes advantage of its shared nothing architecture. They also utilize a parallel approach during insertions where the HBase API is bypassed, and HFiles are directly written

down to the HDFS with the use of the MapReduce framework. The work also deals with the overhead caused by the use of the HBase API. They argue that during bulk insertions of ordered data (for example, log entries ordered by date) range-partitioned data structures like HBase have an inherent load balancing problem, where each time most of the load is handled by a small number of nodes. They deal with this problem by utilizing Hadoop's MapReduce framework to perform parallel bulk insertions of sorted data. Similar to my approach, mappers read the content previously uploaded in the HDFS, sort the values according to a global order-preserving partitioner and reducers materialize ordered ranges to HFiles in the HDFS. They utilize a preliminary sampling phase that approximates the key distribution in order to split the ID space evenly and each reducer gets an equal share of the load (the output of the sampling phase is fed to the Partitioner).

The fundamental problem of bulk insertion in distributed ordered tables where most of the load is directed in a small number of nodes is studied in [SCS⁺08]. The authors deal with the bottleneck problem that results in low throughput during bulk insertions with the use of a preliminary planning phase before the insertion. The planning phase creates new partitions and intelligently distributes partitions among machines to maximize perceived throughput and to minimize data transfer. This works by a first small sampling step, in which the system tries to identify regions and servers that will be overloaded during the bulk insertion. The outcome of the sampling phase is a schedule of partition splits and moves (from one server to another). Existing partitions are pre-splitting before they reach their maximum size and are moved to numerous servers in order to accommodate the foreseen incoming insertions with more than one servers. Nevertheless, moving underfull partitions is also a costly procedure. The authors first show that this partition to server reassignment and movement is an NP-hard problem and present a solution through approximation. They finally apply their algorithm to PNUTS [CRS⁺08], Yahoo's equivalent to BigTable.

The combination of HBase with the MapReduce framework also is applicable to indexing RDF triplestores which is used to answer semantically enriched SPARQL queries, as in the works of [CSC⁺09, KU10, ABC⁺10].

In [CSC⁺09] the authors present SPIDER, a distributed system based on Hadoop and HBase that can efficiently handle the storage of RDF triples and the answering of SPARQL queries. Large amounts of RDF data is stored in HBase region servers called "triple servers". SPARQL queries which require distributed subgraph processing are analyzed into Hadoop MapReduce jobs and are executed in parallel by the triple servers. SPIDER combines HBase's flexibility of storing sparse semi-structured RDF data with the powerful execution framework of Hadoop.

In [KU10] both HBase and MapReduce are used to store RDF triples. In this paper, the authors present a framework for creating and serving large RDF stores using a combination of the MapReduce framework (for creating the index) and a NoSQL system (for serving the index). The authors present SPARQLp grammar, which is considered an extension to the standard SPARQL

grammar and is used to create and query the index. In essence, the authors have added two different functions in the SPARQLp grammar: the first is the “DEFINE” function. This function is similar to an SQL function that creates a materialized view, and is used by their framework to pre-compute and store in the distributed file system some expensive SPARQL queries. This materialization is done through the execution of a MapReduce workflow. For each “DEFINE” function a NoSQL table (with a single index key) is created. This NoSQL table is used by the “GET” function to perform simple exact match NoSQL get operations on the table’s indexed key. Finally, the authors measure the table creation procedure of the “DEFINE” query and the performance of the “GET” operation under different query workloads.

Distributed RDF data management with the combined use of HBase and Hadoop is also performed in the work of [ABC⁺10]. The authors present ProvBase, a distributed system based on HBase and Hadoop which provides a three-table storage schema that can be instantiated in HBase to hold provenance triples and querying algorithms that evaluate SPARQL queries in HBase using its native API.

5.4 Discussion

The indexing system described above is an attempt to leverage the abilities of Hadoop and HBase over similar distributed approaches. In essence, we complement the MapReduce framework’s ability to distribute processing over a large number of nodes with HBase’s flexible and high performance architecture. Similar solutions (see Pig [ORS⁺08] or Hive [TSJ⁺09]) deal with the same problems by running MapReduce jobs for each query submitted. While this allows for complex queries, it also requires significant processing time on the servers for each query and knowledge of the way the data are physically stored. In comparison, our solution aims for speed in simple queries, while most of the processing required on complex queries is performed by the client. This keeps network traffic and CPU load on the data servers at a minimum, allowing for more concurrent client connections. HadoopDB [ABPA⁺09] extends Hive with support for SQL queries but suffers from the same issues, although it does allow for data partitioning, which would be comparable with our indexing process. However, this requires explicit knowledge of the HadoopDB architecture, while alternative partitioning rules (i.e., views in database terminology) cannot be enforced.

In contrast to other approaches (such as Ivory [LMEW10] and HIndex [LRST09]) that distribute only one part of the process, our implementation has the benefit of a fully distributed architecture. This ensures full utilization of the available physical infrastructure and increased scalability. Moreover, our approach significantly reduces the time needed for both index creation and query responses.

Considering that data are already stored in HDFS, storing them again in HBase seems redundant: one could use the offsets of an HDFS file as record identifiers. Yet, such an approach makes insertion of records difficult, as all offsets need to be re-calculated. HBase can perform single imports and requires few updates in the index table per new record. Instead of index updates, if consistency between the context table and its index is relaxed, periodic reruns of the Indexer can be used. An evaluation of the trade-off between these two approaches is left for future work.

At the moment, the content and index creation is a serial procedure that could be otherwise pipelined: the output of an Uploader reducer could be directly fed to the Indexer mapper and written down to HDFS as HFiles at the same time using a chain of MapReduce steps. Nevertheless, this operation is new and, in our opinion, relatively unstable in the current Hadoop version.

The need for complex row keys in the index table (e.g., *google_revision*) was imposed from the diversity of our datasets. Different data types (structured, unstructured, semi-structured) constrain the granularity of the index in different ways. With such a complex row key, a single index table that accommodates all these constraints can be used, while making client lookups for specific attributes from different data types fast and straightforward. This structure can also overcome HBase limitations on the dimensionality of the data stored in its tables.

In the future, our system will be extended to support more complex queries, such as SQL-like joins, and complex table views for the content and index tables. This would make our system more functional while preserving its main advantages, speed and ease of use. Another useful improvement to our implementation would be support for secondary indices to speedup custom searching on different dataset dimensions. This would be implemented using the same basic design as the index table and would immediately increase the usability and lower response times for complex, multidimensional queries. We are also considering the deployment of our system to an actual cloud vendor such as Amazon.

Conclusions and Future Work

6.1 Conclusions

This thesis focused on the behaviour and performance of distributed systems when they are stressed with high and dynamic load. Emphasis was given in the load problems that arise during processing and serving big amounts of data in networks of shared nothing commodity machines. More specific, this work studied how a network of shared nothing commodity machines can efficiently handle two important mechanisms in general data management systems: range queries and index maintenance. My emphasis was on the efficient handling of imbalances created by skewed data accesses in distributed range-queriable data structures, and the distribution of index creation and serving using state-of-the-art cloud enabled data processing techniques. To achieve this, methods from various areas of distributed computing such as peer to peer and cloud-based data processing systems were employed. In both situations, my concern was to devise and utilize adaptive schemes that could scale to a high number of participating nodes in order to satisfy arbitrarily large and diverse workloads with the least possible human interaction.

Regarding range queries in distributed systems, this thesis analyzed existing approaches, compared their performance both experimentally and theoretically and detected their relative advantages and shortcomings. This research led to the design and implementation of *NIX-MIG* [KTK11, KTK09, KTK08], a scalable decentralized algorithm for load balancing of distributed range queriable data structures. Although this problem has received a lot of attention and there have been proposed many solutions during the last years, none of them offered load

balancing without the use of replication, centralized load directories or expensive message probing. In *NIXMIG*, we proposed a novel wave-like locking and load dissemination mechanism which achieves distributed peer co-ordination and collaboration during load exchanges. *NIXMIG* peers, by utilizing only local knowledge, are able to detect global load imbalances due to skewed data access and to perform fast and bandwidth efficient on-line load balancing.

With respect to the distribution of the index creation and serving, the thesis investigated numerous state-of-the art processing frameworks and data stores. This research led to the design and development of the distributed indexer of Chapter 5 [KATK10]. This system combines the power of the widely adopted open source alternatives of MapReduce [DG08] and BigTable [CDG⁺08] in order to create a fully parallel processing and serving framework. The indexer parallelizes both the index creation through MapReduce and the index and content serving through BigTable, compared to other approaches proposed in the literature where they parallelize only one part of the procedure. The experimental evaluation proved that the existing open-source programs like Hadoop [Apa11b] and HBase [Apa11d] have matured and they can be used quite effectively in order to deal with data scalability issues.

Throughout the dissertation, we got involved in various stages of system design and evaluation: from simulation to real implementation and from theoretical to experimental analysis. *NIXMIG* was implemented in a simulated environment and was evaluated both experimentally and theoretically, whereas the distributed indexer of Chapter 5 was an actual implementation that was evaluated in a physical cluster. Each approach has its relative advantages and disadvantages. A simulated environment enables the easy repetition of experiments in order to identify how a specific parameter affects the system's performance. Nevertheless, a small design error in the simulation can completely alter the produced results, and if it remains untraced, it can cause the extraction of erroneous conclusions. On the other hand, an actual implementation produces more indisputable results, but those results are tightly bound to the specific hardware and software configuration which makes their generalization more difficult. A correct theoretical evaluation provides indisputable and generalizable results, but in the case of distributed systems with so many degrees of freedom and randomness, it is very difficult and sometimes impossible to obtain a closed-form formula that captures the system's behavior. In my opinion, most of the times a combination of the aforementioned methods is necessary in order to extract meaningful conclusions.

In the *NIXMIG* case, apart from experimental evaluation, we were also able to theoretically calculate the algorithm's performance and provide closed-formulas with the utilization of legacy load balancing theories [GM96, Cyb89]. Considering the fact that this area of research is quite mature, from one hand we had the opportunity to study many different approaches, but on the other hand it was difficult to prove that our system is significantly different. Nevertheless, our system proved to be more efficient in many cases compared to other popular approaches. In

the case of the distributed indexer of Chapter 5, we managed to build our framework on top of relatively unstable versions of open-source software, we received invaluable help from the communities of both Hadoop and HBase, we contributed with numerous bug fixes in the HBase system, and our research received their attention. The scientific area of cloud-enabled distributed processing systems is relatively new but it is rapidly evolving, mainly because of the increased need for fast, and sometimes realtime, processing of vast amounts of data. In this case, real implementations offer fast solutions to actual problems and therefore are very useful for the respective communities.

If we treat distributed systems as black boxes, they have the same functionality and they perform the same basic tasks with their centralized alternatives. What is more, in small scale, centralized architectures can prove to be more efficient and easier to develop and maintain. For instance, range query support is a de-facto operation in centralized RDBMS for years, and index creation is a straightforward task for single server approaches. An RDBMS like MySQL [MyS11] can evaluate range queries easier than *NIXMIG*, and a centralized index creation mechanism like Lucene [Apa11e] can be more configurable and simple to setup than the distributed indexer. Nevertheless, when both the storage and access load increases, single server approaches fail. In these situations distributed approaches are showing their true strength. As of this, if a system is expected to experience high loads, the design of a distributed approach from the beginning is a wise choice despite the architectural and operational difficulties.

6.2 Future Directions

In this section we discuss about possible research directions in our work. We are considering many axes along which our work can be extended. The first direction constitute an enhancement in *NIXMIG*, the load balancing algorithm presented in Chapter 4, while the other enhancement targets the general area of cloud based shared-nothing distributed data management systems presented in Chapter 5.

6.2.1 Adaptive replication of distributed range partitioned data structures.

NIXMIG balances load by redistributing keys between nodes and does not employ any replication strategy. Throughout our analysis, we consider replication to be an orthogonal approach. In Figure 6.1 we present how *NIXMIG* balances a skewed workload: Range queries $q_1 \dots q_7$ overload nodes N_1 and N_2 that are responsible for the first 8 keys of the ID space. In this scenario, at time $t = t_a$ either N_1 or N_2 (whoever “wins” during the locking procedure) initiates a *NIXMIG* operation, places nodes N_3 , N_4 and N_5 next to it and offloads a number of items to them, dropping

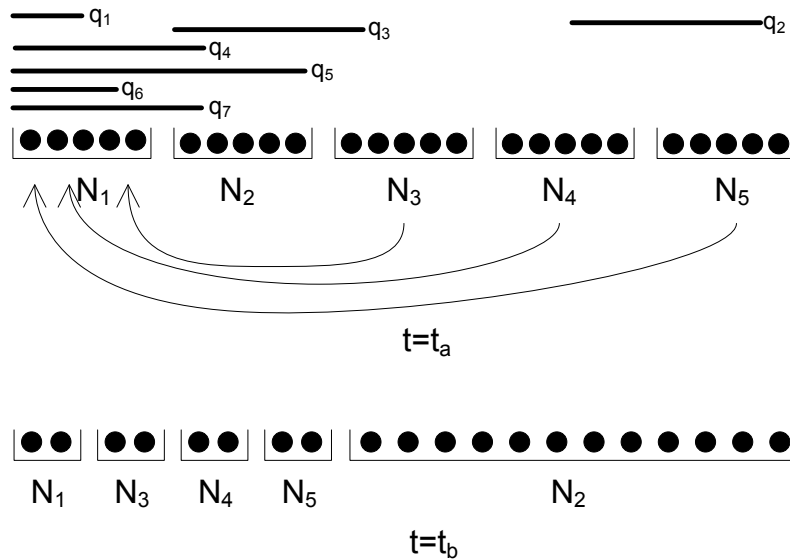


Figure 6.1: NIXMIG balancing. Nodes N_3 , N_4 and N_5 leave their place and take a part of the overloaded area previously served by N_1 and N_2 .

its load below its *thres* value. Finally, when the system is balanced at time $t = t_b$, four nodes are responsible for the first eight “popular” keys, while the rest are served by N_2 .

When we consider replication, an interesting research direction would be to study NIXMIG’s dual approach: instead of rearranging items between nodes, it might be efficient to identify “hot” areas and replicate them in a number of different nodes. In Figure 6.2 we present how the replication approach can balance the same workload as the one of Figure 6.1. An operation similar to a NIXMIG wave is triggered when N_1 or N_2 is overloaded. In this approach, the difference with NIXMIG is that each remote underloaded node N_3 , N_4 and N_5 takes the entire overloaded area of the first eight keys (bottom of Figure 6.2) instead of just taking only a portion of it. Finally, in the balanced state at time $t = t_b$, nodes N_1 , N_2 , N_3 and N_4 are responsible for the first 8 keys, and, assuming a “fair” routing protocol, each node will serve one fourth of the requests for this area. Although this operation resembles a NIXMIG wave, it needs to be designed in a completely different way, as different decisions need to be taken. For instance, during a replication request, nodes need to decide both the item range that will be replicated (e.g. items 1-8 in Figure 6.2) along with the number of total replica holders for this range (e.g. four nodes in Figure 6.2).

6.2.2 NoSQL adaptive replication of popular regions.

NoSQL systems like *HBase* use replication for two reasons: load balancing and fault tolerance. Load is balanced among replica holders as requests are handled by more than one peer, and fault tolerance is achieved as even in the case of failing nodes, the remaining peers continue to

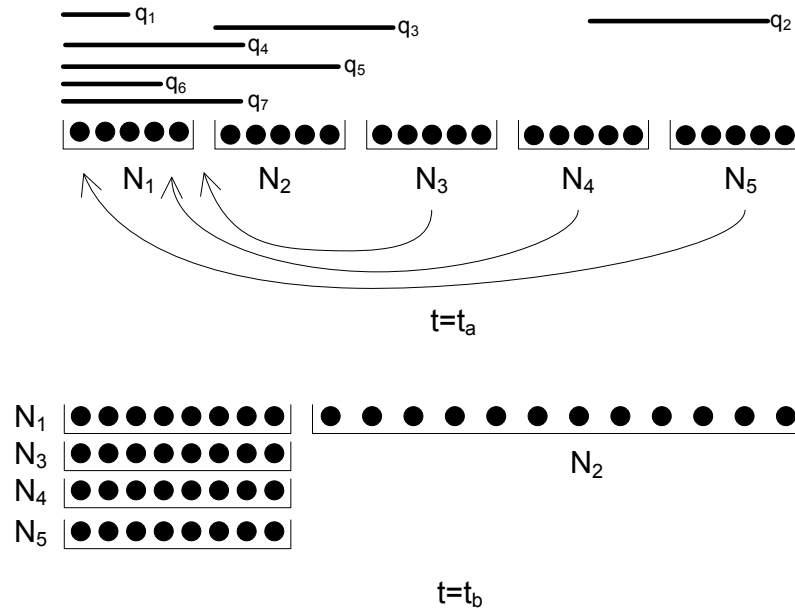


Figure 6.2: Balancing through replication. Nodes N_3, N_4 and N_5 leave their place and, together with N_1 , handle requests for the overloaded area of the first 8 items of the ID space.

serve incoming queries. Nevertheless, *HBase*, like the majority of the existing NoSQL platforms, employ a static replication scheme, with a fixed predefined number of replicas for each region (an *HBase* region is a subset of the ID space containing items whose total size does not exceeds a maximum predefined size, the equivalent of *BigTable* [CDG⁺08]’s Tablet). A fixed number of replicas for each region is not very efficient, considering that, in many cases, item requests are skewed: a small percentage of the total stored items receives the majority of the requests.

In order to tackle the problem posed by static replication, we consider to extend *HBase* to support dynamic replication per stored region. The replication factor per region will be decided at runtime and will be workload adaptive: “popular” regions will increase their available replicas whereas regions that are rarely requested will respectively decrease their replica number. The replication factor per region will be decided at runtime using a monitoring and balancing module. The module will monitor for region specific workload changes, and when the workload value exceed a *thres_high* or drops below a *thres_low* it will increase or decrease respectively the number of available replicas.

Deciding the appropriate number of replicas per region is not a trivial task: one could argue that skewed workload can be tackled by globally setting a large replication factor number. Nevertheless, considering the fact that most NoSQL systems do not create replicas using erasure codes, the total storage space needed for each item gets multiplied by the number of replicas,

thus decreasing the available storage space. Moreover, a big replication factor will make object updates inefficient, as more replica holders need to be synchronized and receive the new object.

What is more, several research issues arise concerning monitoring and replication decisions: is it efficient to deploy the module in the HBase master node? How is the module going to operate in a transparent way, i.e., without further degrading performance in workload intensive situations? Is it feasible to design a generic module so that it can be easily deployed in other NoSQL systems like Cassandra [LM10] or CouchDB [Apa11a]?

Publications

Journals

- *Fast and Cost-Effective Online Load-Balancing in Distributed Range-Queryable Systems.* Ioannis Konstantinou, Dimitrios Tsoumakos and Nectarios Koziris. IEEE Transactions on Parallel and Distributed Systems, Volume 22, Issue 8, August 2011 Pages 1350-1364.
- *A Grid Middleware for Data Management Exploiting Peer-to-Peer Techniques.* Athanasia Asiki, Katerina Doka, Ioannis Konstantinou, Antonis Zissimos, Dimitrios Tsoumakos, Nectarios Koziris and Panayiotis Tsanakas. Future Generation Computer Systems, Volume 25, Issue 4, April 2009, Pages 426-435.

Conferences

- *Distributed Indexing of Web Scale Datasets for the Cloud.* Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos and Nectarios Koziris. In Proceedings of the International Workshop on Massive Data Analytics on the Cloud (MDAC2010 - in conjunction with WWW 2010), Raleigh, NC, USA, 26 April 2010.
- *Distributed Indexing of Web Scale Datasets for the Cloud.* Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos and Nectarios Koziris. 9th Hellenic Data Management Symposium (HDMS 2010), Ayia Napa, Cyprus, July, 2010.

- *Measuring the Cost of Online Load-Balancing in Distributed Range-Queryable Systems.* Ioannis Konstantinou, Dimitrios Tsoumakos and Nectarios Koziris. In Proceedings of the 9th International IEEE Conference on Peer-to-Peer Computing (P2P), Seattle, WA, USA, 8-11 September 2009.
- *GRIDNEWS: A Distributed Automatic Greek Broadcast Transcriptions System.* D. Dimitriadis, A. Metallinou, I. Konstantinou, G. Goumas, P. Maragos and N. Koziris. In Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP-09), Taipei, Taiwan, March 2009.
- *PASS It ON (PASSION): An Adaptive Online Load-Balancing Algorithm for Distributed Range-Query specialized Systems.* Ioannis Konstantinou, Dimitrios Tsoumakos and Nectarios Koziris. In Proceedings of the 16th International Conference on Cooperative Information Systems (CoopIS), Monterrey, Mexico, 12-14 November 2008.
- *A Distributed Architecture for Multi-Dimensional Indexing and Data Retrieval in Grid Environments.* Athanasia Asiki, Katerina Doka, Ioannis Konstantinou, Antonis Zissimos, and Nectarios Koziris. In Proceedings of the Cracow 2007 Grid Workshop (CGW'07), Krakow, Poland, October 16-17, 2007.
- *Gredia Middleware Architecture.* Ioannis Konstantinou, Katerina Doka, Athanasia Asiki, Antonis Zissimos, and Nectarios Koziris. In Proceedings of the Cracow 2007 Grid Workshop (CGW'07), Krakow, Poland, October 16-17, 2007.

Bibliography

- [Aba09] Daniel J. Abadi. Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Engineering Bulletin*, 32(1):3–12, 2009. 86, 123
- [ABC⁺10] John Abraham, Pearl Brazier, Artem Chebotko, Jaime Navarro, and Anthony Piazza. Distributed Storage and Querying Techniques for a Semantic Web of Scientific Workflow Provenance. In *Proceedings of the 2010 IEEE International Conference on Services Computing, SCC '10*, pages 178–185, Miami, Florida, USA, 2010. IEEE Computer Society. 90, 91
- [ABPA⁺09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment*, 2:922–933, August 2009. 87, 88, 91
- [AH08] Jesse Alpert and Nissan Hajaj. We Knew the Web Was Big - The Official Google Blog. <http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html>, 2008. 13, 25
- [AKK04] James Aspnes, Jonathan Kirsch, and Arvind Krishnamurthy. Load Balancing and Locality in Range-Queryable Data Structures. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing, PODC '04*, pages 115–124, St. John's, Newfoundland, Canada, 2004. ACM. 10, 22, 65, 67, 68
- [Ama11a] Amazon. Elastic Compute Cloud (EC2). <http://aws.amazon.com/ec2>, 2011. 5, 19, 124

- [Ama11b] Amazon. Public Data Sets on Amazon Web Services (AWS). <http://aws.amazon.com/publicdatasets/>, 2011. 24
- [Ama11c] Amazon. Simple Storage Service (S3). <http://aws.amazon.com/s3>, 2011. 5, 19
- [Ama11d] Amazon. Web Services (AWS). <http://aws.amazon.com>, 2011. 19
- [Ama11e] Amazon. Website. <http://www.amazon.com>, 2011. 3, 17
- [And08] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2 edition, 2008. 6, 20
- [Apa11a] Apache. CouchDB Project. <http://couchdb.apache.org/>, 2011. 86, 98
- [Apa11b] Apache. Hadoop Home. <http://hadoop.apache.org/>, 2011. 14, 26, 76, 77, 94
- [Apa11c] Apache. Hadoop PoweredBy. <http://wiki.apache.org/hadoop/PoweredBy>, 2011. 77
- [Apa11d] Apache. HBase Home. <http://hbase.apache.org/>, 2011. 14, 26, 75, 94, 123, 124, 125
- [Apa11e] Apache. Lucene. <http://lucene.apache.org/>, 2011. 89, 95
- [Apa11f] Apache. Nutch. <http://lucene.apache.org/nutch/>, 2011. 88
- [APHS02] Karl Aberer, Magdalena Puceva, Manfred Hauswirth, and Roman Schmidt. Improving Data Access in P2P Systems. *IEEE Internet Computing*, 6:58–67, January 2002. 8, 21
- [app] Google AppEngine. code.google.com/appengine. 124
- [Arr06] Michael Arrington. AOL Proudly Releases Massive Amounts of Private Data. <http://techcrunch.com/2006/08/06/aol-proudly-releases-massive-amounts-of-user-search-data/>, 2006. 63, 82
- [AS07] James Aspnes and Gauri Shah. Skip Graphs. *ACM Transactions on Algorithms*, 3, November 2007. 7, 8, 11, 20, 21, 24, 50, 66, 70
- [azua] AzureWatch. <http://www.paraleap.com/azurewatch>. 124, 138
- [azub] Windows Azure Platform. <http://www.microsoft.com/windowsazure/>. 124
- [BAC⁺90] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2:4–24, March 1990. 64

- [BAS04] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM'04*, pages 353–366, Portland, Oregon, USA, 2004. ACM. 10, 22, 65, 68, 72
- [BBC⁺11] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *5th Biennial Conference on Innovative Data Systems Research, CIDR '11*, Asilomar, California, USA, 2011. 86
- [BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008. 70
- [BCM03] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 80–87. Springer Berlin / Heidelberg, 2003. 71
- [BCMR02] John Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed Content Delivery across Adaptive Overlay Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '02*, pages 47–60, Pittsburgh, Pennsylvania, USA, 2002. ACM. 5, 19
- [BEK⁺09] Kevin S. Beyer, Vuk Ercegovic, Rajasekar Krishnamurthy, Sriram Raghavan, Jun Rao, Frederick Reiss, Eugene J. Shekita, David E. Simmen, Sandeep Tata, Shivakumar Vaithyanathan, and Huaiyu Zhu. Towards a Scalable Enterprise Content Analytics Platform. *IEEE Data Engineering Bulletin*, 32(1):28–35, 2009. 88
- [BFH09] Petra Berenbrink, Tom Friedetzky, and Zengjian Hu. A New Analytical Method for Parallel, Diffusion-Type Load Balancing. *Journal of Parallel and Distributed Computing*, 69(1):54–61, 2009. 47
- [BKK⁺03] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up Data in P2P Systems. *Communications of the ACM*, 46:43–48, February 2003. 21

- [BMBB10] Antonio Barbuzzi, Pietro Michiardi, Ernst Biersack, and Gennaro Boggia. Parallel Bulk Insertion for Large-Scale Analytics Applications. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware, LADIS '10*, pages 27–31, Zurich, Switzerland, 2010. ACM. 89
- [BPS06] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a Distributed Architecture for Online Multiplayer Games. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3, NSDI'06*, pages 12–12, San Jose, CA, 2006. USENIX Association. 8, 21
- [CDG⁺08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26:4:1–4:26, June 2008. 7, 14, 20, 26, 76, 77, 81, 86, 94, 97, 123
- [CER08] CERN. Worldwide LHC Computing Grid. <http://public.web.cern.ch/public/en/LHC/Computing-en.html>, 2008. 4, 17
- [CJL⁺08] Ronnie Chaiken, Bob Jenkins, PerAke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment*, 1:1265–1276, August 2008. 24, 88
- [CKR⁺07] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, IMC '07*, pages 1–14, San Diego, California, USA, 2007. ACM. 8, 21
- [CLGS04] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying Peer-to-Peer Networks Using P-trees. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004, WebDB '04*, pages 25–30, Paris, France, 2004. ACM. 8, 21
- [CLM⁺07] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-ring: An Efficient and Robust P2P Range Index Structure. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07*, pages 223–234, Beijing, China, 2007. ACM. 68

- [CLM⁺11] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. Load Balancing and Range Queries in P2P Systems Using P-Ring. *ACM Transactions on Internet Technology*, 10:16:1–16:30, March 2011. 68
- [clo] Amazon CloudWatch. <http://aws.amazon.com/cloudwatch/>. 124, 138
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Chapter 17: Amortized Analysis. In *Introduction to Algorithms, second edition*, pages 347–369. MIT Press, 2001. 36
- [CRS⁺08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment*, 1:1277–1288, August 2008. 90
- [CSC⁺09] Hyunsik Choi, Jihoon Son, YongHyun Cho, Min Kyoung Sung, and Yon Dohn Chung. SPIDER: a System for Scalable, Parallel / Distributed Evaluation of Large-scale RDF Data. In *Proceeding of the 18th ACM Conference on Information and Knowledge Management, CIKM '09*, pages 2087–2088, Hong Kong, China, 2009. ACM. 90
- [CST⁺10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SOCC '10*, pages 143–154, Indianapolis, Indiana, USA, 2010. 124, 125, 128
- [CT08] Chyohwa Chen and Kun-Cheng Tsai. The Server Reassignment Problem for Load Balancing in Structured P2P Systems. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):234–246, 2008. 10, 22, 65, 74
- [Cyb89] George Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989. 6, 20, 47, 94
- [DBK06] Karen D. Devine, Eric G. Boman, and Geogre Karypis. Partitioning and Load Balancing for Emerging Parallel Applications and Architectures. In *Parallel Processing for Scientific Computing*, pages 99–125. Cambridge University Press, 2006. 22
- [DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. 7, 14, 20, 24, 26, 75, 76, 77, 87, 94

- [DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSOP '07, pages 205–220, Stevenson, Washington, USA, 2007. ACM. 86, 123, 124
- [Dij74] Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17:643–644, November 1974. 6, 20
- [DKK⁺01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, 2001. 10, 22, 65, 72
- [DL09] C. Dale and Jiangchuan Liu. apt-p2p: A Peer-to-Peer Distribution System for Software Package Releases and Updates. In *INFOCOM 2009. Twenty-eight Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, pages 864–872, Rio de Janeiro, Brazil, April 2009. 5, 19
- [Dub05] Manek Dubash. Moore's Law is dead, says Gordon Moore. <http://news.techworld.com/operating-systems/3477/moores-law-is-dead-says-gordon-moore/>, 2005. 4, 18
- [DW00] Christian Damgaard and Jacob Weiner. Describing Inequality in Plant Size or Fecundity. *Ecology*, 81(4):1139–1142, 2000. 31, 59
- [eBa11] eBay. Website. <http://www.ebay.com>, 2011. 3, 17
- [Fac11] Facebook. Website. <http://www.facebook.com>, 2011. 3, 5, 17, 18
- [Fou11a] Wikimedia Foundation. MWDumper. <http://www.mediawiki.org/wiki/MWDumper>, 2011. 81
- [Fou11b] Wikimedia Foundation. Wikimedia Downloads. <http://dumps.wikimedia.org/>, 2011. 81
- [Gar11] Gartner. Gartner Identifies the Top 10 Strategic Technologies for 2011. <http://www.gartner.com/it/page.jsp?id=1454221>, 2011. 6, 19
- [GBGM04] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 444–455, Toronto, Canada, 2004. VLDB Endowment. 10, 22, 65, 66, 73

- [GDA07] Sarunas Girdzijauskas, Anwitaman Datta, and Karl Aberer. Oscar: Small-World Overlay for Realistic Key Distributions. In Gianluca Moro, Sonia Bergamaschi, Sam Joseph, Jean-Henry Morin, and Aris Ouksel, editors, *Databases, Information Systems, and Peer-to-Peer Computing*, volume 4125 of *Lecture Notes in Computer Science*, pages 247–258. Springer Berlin / Heidelberg, 2007. 69, 72
- [GDA10] Sarunas Girdzijauskas, Anwitaman Datta, and Karl Aberer. Structured Overlay for Heterogeneous Environments: Design and Evaluation of Oscar. *ACM Transactions on Autonomous and Adaptive Systems*, 5:2:1–2:25, February 2010. 69, 72
- [Geb89] Monica A. Geber. Interplay of Morphology and Development on Size Inequality: a Polygonum Greenhouse Study. *Ecological Monographs*, 59(3):267–288, 1989. 32
- [GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Operating Systems Review*, 37:29–43, October 2003. 77
- [GH05] George Giakkoupis and Vassos Hadzilacos. A Scheme for Load Balancing in Heterogenous Distributed Hash Tables. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 302–311, Las Vegas, NV, USA, 2005. ACM. 69
- [GM96] Bhaskar Ghosh and S. Muthukrishnan. Dynamic Load Balancing by Random Matchings. *Journal of Computer and System Sciences*, 53:357–370, December 1996. 47, 94
- [GoG11] GoGrid. Cloud Infrastructure. <http://www.gogrid.com>, 2011. 5, 19
- [GR10] John Gantz and David Reinsel. The Digital Universe Decade – Are You Ready? <http://www.emc.com/collateral/analyst-reports/idc-digital-universe-201005.pdf>, 2010. 3, 17
- [GS04] Jun Gao and Peter Steenkiste. An Adaptive Protocol for Efficient Support of Range Queries in DHT-Based Systems. In *Proceedings of the 12th IEEE International Conference on Network Protocols*, ICNP '04, pages 239–250, Berlin, Germany, 2004. IEEE Computer Society. 70
- [GS05] P. Brighten Godfrey and Ion Stoica. Heterogeneity and Load Balance in Distributed Hash Tables. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, pages 596 – 606, march 2005. 10, 22, 65, 73

- [GSBK04] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive Replication in Peer-to-Peer Systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 360–369, 2004. 65
- [GSRM⁺09] Fermín Galán, Americo Sampaio, Luis Rodero-Merino, Irit Loy, Victor Gil, and Luis M. Vaquero. Service specification in cloud environments based on extensions to open standards. In *COMSWARE*, 2009. 124, 137
- [Gut11] Project Gutenberg. Custom ISO Creator. <http://snowy.arsc.alaska.edu/pgiso/>, 2011. 81
- [HAMS97] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range Queries in OLAP Data Cubes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, SIGMOD '97*, pages 73–88, Tucson, Arizona, United States, 1997. ACM. 8, 21
- [Hen95] Robert L. Henderson. Job Scheduling Under the Portable Batch System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag. 8, 12, 21, 24
- [HFC⁺08] Yan Huang, Tom Z.J. Fu, Dah-Ming Chiu, John C.S. Lui, and Cheng Huang. Challenges, Design and Analysis of a Large-scale P2P-VoD System. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, SIGCOMM '08*, pages 375–388, Seattle, WA, USA, 2008. ACM. 5, 19
- [HJS⁺03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4, USITS'03*, pages 9–9, Seattle, WA, 2003. USENIX Association. 8, 21
- [HLCH11] Hung-Chang Hsiao, Hao Liao, Ssu-Ta Chen, and Kuo-Chan Huang. Load Balance with Imperfect Information in Structured Peer-to-Peer Systems. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):634–649, april 2011. 10, 22, 65, 73
- [HM90] Joseph Y. Halpern and Yoram Moses. Knowledge and Common Knowledge in a Distributed Environment. *Journal of the ACM*, 37:549–587, July 1990. 6, 20
- [Hor11] Eliot Horowitz. Foursquare Outage Post Mortem. <http://bit.ly/gjA2IK>, 2011. [Online; accessed 02-May-2011]. 6, 20, 124

- [Int11a] InternetArchive. Digital Library of Free Books, Movies, Music and the Wayback Machine. <http://www.archive.org/>, 2011. 24
- [Int11b] InternetArchive. The Wayback Machine. <http://web.archive.org>, 2011. 24
- [Jac09] Adam Jacobs. The Pathologies of Big Data. *Communications of the ACM*, 52(8):36–44, 2009. 4, 17
- [JKR02] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of the 11th International Conference on World Wide Web, WWW '02*, pages 293–304, Honolulu, Hawaii, USA, 2002. ACM. 7, 9, 20, 21
- [Jou08] Yuh-Jzer Joung. Approaching Neighbor Proximity and Load Balance for Range Query in P2P Networks. *Computer Networks*, 52(7):1451–1472, 2008. 10, 22, 65, 72
- [Jov82] Boyan Jovanovic. Selection and the Evolution of Industry. *Econometrica*, 50(3):649–670, May 1982. 32
- [JOV05] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. BATON: a Balanced Tree Structure for Peer-to-Peer Networks. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05*, pages 661–672, Trondheim, Norway, 2005. VLDB Endowment. 8, 21, 70
- [KATK10] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, and Nectarios Koziris. Distributed Indexing of Web Scale Datasets for the Cloud. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud (in conjunction with WWW 2010)*, pages 1–6, Raleigh, North Carolina, 2010. ACM. 75, 94
- [KBHR10] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *Proceedings of the 1st ACM symposium on Cloud Computing, SoCC '10*, pages 75–86, Indianapolis, Indiana, USA, 2010. ACM. 7, 20
- [Kir10] Marshall Kirkpatrick. Google CEO Schmidt: People Aren't Ready for the Technology Revolution. http://www.readwriteweb.com/archives/google_ceo_schmidt_people_arent_ready_for_the_tech.php, 2010. 3, 17
- [KKL10a] Donald Kossmann, Tim Kraska, and Simon Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proceedings of the 2010*

- ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 579–590, Indianapolis, Indiana, USA, 2010. 124, 138
- [KKL⁺10b] Donald Kossmann, Tim Kraska, Simon Loesing, Stephan Merkli, Raman Mittal, and Flavio Pfaffhauser. Cloudy: A Modular Cloud Storage System. *Proceedings of the VLDB Endowment*, 3(2):1533–1536, 2010. 138
- [Kle00] Jon Kleinberg. The Small-World Phenomenon: An Algorithm Perspective. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 163–170, Portland, Oregon, United States, 2000. ACM. 68, 69
- [KLL⁺97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, El Paso, Texas, United States, 1997. ACM. 65, 72
- [KLXH04] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-Peer Support for Massively Multiplayer Games. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 1, 2004. 5, 8, 19, 21
- [KPC89] Robert G. Knox, Robert K. Peet, and Norman L. Christensen. Population Dynamics in Loblolly Pine Stands: Changes in Skewness and Size Inequality. *Ecology*, 70(4):1153–1166, August 1989. 32
- [KR06] David R. Karger and Matthias Ruhl. Simple Efficient Load-Balancing Algorithms for Peer-to-Peer Systems. *Theory of Computing Systems*, 39:787–804, 2006. 10, 11, 12, 22, 24, 29, 38, 53, 65, 66, 73
- [KS03] Manolis Koubarakis and Timos Sellis. Chapter 1: Introduction. In *Spatio-Temporal Databases*, volume 2520 of *Lecture Notes in Computer Science*, pages 1–8. Springer Berlin Heidelberg, 2003. 8, 21
- [KTK08] Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. PASS It ON (PASSION): An Adaptive Online Load-Balancing Algorithm for Distributed Range-Query Specialized Systems. In *Proceedings of the 16th International Conference on Cooperative Information Systems (CoopIS)*, pages 3–5, Monterrey, Mexico, 2008. 29, 93

- [KTK09] Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. Measuring the Cost of Online Load-Balancing in Distributed Range-Queryable Systems. In *Proceedings of the IEEE Ninth International Conference on Peer-to-Peer Computing, P2P '09*, pages 135–138, Seattle, WA, USA, 2009. 29, 93
- [KTK11] Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. Fast and Cost-Effective Online Load-Balancing in Distributed Range-Queryable Systems. *IEEE Transactions on Parallel and Distributed Systems*, 22(8):1350–1364, August 2011. 29, 93
- [KU10] Spyros Kotoulas and Jacopo Urbani. SPARQL Query Answering on a Shared-Nothing Architecture. In *Proceedings of the SemData Workshop and VLDB2010*, 2010. 90
- [KVD⁺07] Yannis Kotidis, Vasilis Vassalos, Antonios Deligiannakis, Vassilis Stoumpos, and Alex Delis. Robust Management of Outliers in Sensor Network Aggregate Queries. In *Proceedings of the 6th ACM international workshop on Data engineering for wireless and mobile access, MobiDE '07*, pages 17–24, Beijing, China, 2007. ACM. 8, 21
- [Lam78] Leslie Lamport. Time Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21:558–565, July 1978. 6, 20
- [LBC10] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated Control for Elastic Storage. In *Proceedings of the 7th international conference on Autonomic computing, ICAC '10*, pages 1–10, Washington, DC, USA, 2010. ACM. 124
- [LCC⁺02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of the 16th International Conference on Supercomputing, ICS '02*, pages 84–95. ACM, 2002. 65
- [LCLC09] Dongsheng Li, Jiannong Cao, Xicheng Lu, and K. Chen. Efficient Range Query Processing in Peer-to-Peer Systems. *IEEE Transactions on Knowledge and Data Engineering*, 21(1):78–91, 2009. 10, 22, 65, 71, 74
- [LKGH03] Xin Li, Young Jin Kim, Ramesh Govindan, and Wei Hong. Multi-Dimensional Range Queries in Sensor Networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 63–75, Los Angeles, California, USA, 2003. ACM. 8, 21

- [LLL00] Sin Yeung Lee, Tok Wang Ling, and Hua-Gang Li. Hierarchical Compact Cube for Range-Max Queries. In *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00*, pages 232–241, Cario, Egypt, 2000. Morgan Kaufmann Publishers Inc. 8, 21
- [LLM88] Michael J Litzkow, Miron Livny, and Matt W Mutka. Condor - A hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS'88)*, pages 104 –111, 1988. 8, 12, 21, 24
- [LM10] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44:35–40, April 2010. 5, 6, 19, 86, 98, 123, 124, 125, 127
- [LMEW10] Jimmy Lin, Donald Metzler, Tamer Elsayed, and Lidan Wang. Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search. In *TREC*, 2010. 89, 91
- [LN01] Qiong Luo and Jeffrey F. Naughton. Form-Based Proxy Caching for Database-Backed Web Sites. In *Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01*, pages 191–200, Roma, Italy, 2001. Morgan Kaufmann Publishers Inc. 8, 21
- [LRST09] Ning Li, Jun Rao, Eugene Shekita, and Sandeep Tata. Leveraging a Scalable Row Store to Build a Distributed Text Index. In *Proceedings of the First International Workshop on Cloud Data Management, CloudDB '09*, pages 29–36, Hong Kong, China, 2009. 85, 89, 91
- [Man04] Gurmeet Singh Manku. Balanced Binary Trees for ID Management and Load Balance in Distributed Hash Tables. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, PODC '04*, pages 197–205, St. John's, Newfoundland, Canada, 2004. ACM. 65, 73
- [MC05] Donald Metzler and W. Bruce Croft. A Markov Random Field Model for Term Dependencies. In *Proceedings of the 28th annual international ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '05*, pages 472–479, Salvador, Brazil, 2005. ACM. 89
- [MCC04] M. L Massie, B. N Chun, and D. E Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004. 126

- [Mit01] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12:1094–1104, October 2001. 69, 71
- [MKF10] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic Site: Using Clouds to Elastically Extend Site Resources. In *Proceedings of the 2009 10th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID'10*, Melbourne, Victoria, Australia, 2010. 124, 137
- [MKG07] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkatasubramanian. L-diversity: Privacy Beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data*, 1, March 2007. 6, 20
- [Mor62] James Morgan. The Anatomy of Income Distribution. *The Review of Economics and Statistics*, 44(3):270–283, August 1962. 32
- [MU05] M. Mitzenmacher and E. Upfal. Balls, Bins, and Random Graphs. In *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, pages 90–125. Cambridge University Press, 2005. 39
- [MyS11] MySQL. Homepage. <http://www.mysql.com/>, 2011. 95
- [Net11] Netflix. Lessons Netflix Learned from the AWS Outage. <http://techblog.netflix.com/2011/04/lessons-netflix-learned-from-aws-outage.html>, 2011. [Online; accessed 02-May-2011]. 6, 20
- [NoS09] NoSQL. Nosql databases. <http://nosql-databases.org>, 2009. 86
- [NW07] Moni Naor and Udi Wieder. Novel Architectures for P2P applications: The Continuous-Discrete Approach. *ACM Transactions on Algorithms*, 3, August 2007. 65
- [NWG⁺09] Daniel Nurmi, Rich Wolski, Chris Grzegorzcyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID '09*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society. 124, 126
- [Ope11] Openstack. Open Source Cloud Computing Software. <http://www.openstack.org>, 2011. 6, 19, 124

- [Opi10] OpinionMatters. Is 2010 the year SMEs fully embrace cloud computing? <http://www.easynetconnect.net/Portals/0/DownloadFiles/IndustryInsight/WhitePapers/Is-2010-the-year-SMEs-fully-embrace-cloud-computing.pdf>, 2010. 5, 19
- [Ore86] Jack A. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 326–336, Washington, D.C., United States, 1986. ACM. 8, 21
- [ORS⁺08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, Vancouver, Canada, 2008. ACM. 12, 15, 24, 26, 87, 91
- [PCW09] PCWorld. Jackson's Death a Blow to the Internet. http://www.pcworld.com/article/167435/jacksons_death_a_blow_to_the_internet.html, 2009. [Online; accessed 02-May-2011]. 6, 20
- [PNT06] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Replication, Load Balancing and Efficient Range Query Processing in DHTs. In *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 131–148. Springer Berlin / Heidelberg, 2006. 64
- [PNT10] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Saturn: Range Queries, Load Balancing and Fault Tolerance in DHT Data Systems. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints), 2010. 65
- [PPR⁺09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD'09, pages 165–178, 2009. 124
- [PT07] Theoni Pitoura and Peter Triantafillou. Load Distribution Fairness in P2P Data Management Systems. In *IEEE 23rd International Conference on Data Engineering*, ICDE '07, pages 396–405, april 2007. 32, 59
- [PT09] Theoni Pitoura and Peter Triantafillou. Distribution Fairness in Internet-Scale Networks. *ACM Transactions on Internet Technology*, 9:16:1–16:36, October 2009. 32, 59

- [Pug90] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990. 49
- [QS04] Dongyu Qiu and R. Srikant. Modeling and Performance Analysis of BitTorrent-like Peer-to-Peer Networks. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 367–378, Portland, Oregon, USA, 2004. ACM. 5, 19
- [Rac11] Rackspace. Rackspace Hosting. <http://www.rackspace.com>, 2011. 5, 19
- [RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer Berlin / Heidelberg, 2001. 9, 22, 64
- [RFH⁺01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 161–172, San Diego, California, United States, 2001. ACM. 9, 22
- [RFI02] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. *IEEE Internet Computing Journal*, 6(1), 2002. 8, 21
- [RHRS04] Sriram Ramabhadran, Joseph Hellerstein, Sylvia Ratnasamy, and Scott Shenker. Prefix Hash Tree: An Indexing Data Structure Over Distributed Hash Tables. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, St. John's, Newfoundland, Canada, 2004. ACM. 8, 21
- [ria11] Riak Homepage. <https://wiki.basho.com/display/RIAK/Riak>, 2011. 124, 125, 128
- [Rig11] Rightscale. Cloud Computing Management Platform. <http://www.rightscale.com>, 2011. 5, 19
- [RLS⁺03] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In *Second International Workshop on Peer to Peer Systems*, IPTPS '03, pages 68–79, Berkeley, CA, USA, 2003. 10, 22, 65, 73

- [SCS⁺08] Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, Erik Vee, Ramana Yerneni, and Raghu Ramakrishnan. Efficient Bulk Insertion into a Distributed Ordered Table. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 765–778, Vancouver, Canada, 2008. ACM. 90
- [SGG03] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts. *Multimedia Systems*, 9:170–184, 2003. 10.1007/s00530-003-0088-1. 5, 19
- [SGL⁺06] Sonesh Surana, Brighten Godfrey, Karthik Lakshminarayanan, Richard Karp, and Ion Stoica. Load Balancing in Dynamic Structured Peer-to-Peer Systems. *Performance Evaluation*, 63:217–240, March 2006. 10, 22, 65, 73, 74
- [sim] Amazon SimpleDB. <http://aws.amazon.com/simpledb/>. 124
- [SKS92] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25:33–44, December 1992. 64
- [Sky11] Skype. Website. <http://www.skype.com>, 2011. 4, 18
- [SMA⁺08] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Abounaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 953–966, Vancouver, Canada, 2008. ACM. 137
- [SMK⁺01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, San Diego, California, United States, 2001. ACM. 9, 21, 64, 65, 70
- [Sta95] S.H. Standard. FIPS 180-1. *Secure Hash Standard, NIST, US Dept. of Commerce, Washington DC April*, 1995. 9, 21
- [SW02] Subhabrata Sen and Jia Wang. Analyzing Peer-to-Peer Traffic Across Large Networks. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement*, IMW '02, pages 137–150, Marseille, France, 2002. ACM. 8, 21

- [SX07] Haiying Shen and Cheng-Zhong Xu. Locality-Aware and Churn-Resilient Load-Balancing Algorithms in Structured Peer-to-Peer Networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):849–862, 2007. 10, 22, 65, 71, 73
- [SX08] Haiying Shen and Cheng-Zhong Xu. Hash-based Proximity Clustering for Efficient Load Balancing in Heterogeneous DHT Networks. *Journal of Parallel and Distributed Computing*, 68(5):686–702, 2008. 10, 22, 65, 71, 73
- [Tag11] TagSoup. Homepage. <http://home.ccil.org/~cowan/XML/tagsoup/>, 2011. 82
- [TR06] Dimitrios Tsoumakos and Nick Roussopoulos. An Adaptive Probabilistic Replication Method for Unstructured P2P Networks. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *Lecture Notes in Computer Science*, pages 480–497. Springer Berlin / Heidelberg, 2006. 65
- [TRFD10] Konstantinos Tsakalozos, Mema Roussopoulos, Vangelis Floros, and Alex Delis. Nefeli: Hint-Based Execution of Workloads in Clouds. In *Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS'10)*, pages 74–85, 2010. 138
- [TSJ⁺09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2:1626–1629, 2009. 15, 26, 87, 88, 91
- [twi11] twitter. Website. <http://twitter.com>, 2011. 3, 4, 17, 18
- [TZX10] Yuzhe Tang, Shuigeng Zhou, and Jianliang Xu. LIGHT: A Query-Efficient Yet Low-Maintenance Indexing Scheme over DHTs. *IEEE Transactions on Knowledge and Data Engineering*, 22(1):59–75, january 2010. 71
- [Vol09] Voldermort. Project Voldemort. <http://project-voldemort.com>, 2009. 5, 19, 86, 123, 124
- [VORT09] Quang Hieu Vu, Beng Chin Ooi, Martin Rinard, and Kian-Lee Tan. Histogram-Based Global Load Balancing in Structured Peer-to-Peer Systems. *IEEE Transactions on Knowledge and Data Engineering*, 21(4):595–608, 2009. 10, 22, 65, 70
- [VRMB11] Luis M. Vaquero, Luis Roderó-Merino, and Rajkumar Buyya. Dynamically Scaling Applications in the Cloud. *SIGCOMM Computer Communication Review*, 41:45–52, 2011. 137

- [VSCF09] Ymir Vigfusson, Adam Silberstein, Brian F. Cooper, and Rodrigo Fonseca. Adaptively Parallelizing Distributed Range Queries. In *Proceedings of the 35th International Conference on Very Large Data Bases, VLDB '09*, pages 682–693, Lyon, France, 2009. VLDB Endowment. 8, 21
- [WBM⁺06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, pages 307–320, Seattle, Washington, 2006. USENIX Association. 6, 19
- [Wei85] Jacob Weiner. Size Hierarchies in Experimental Populations of Annual Plants. *Ecology*, 66(3):743–752, 1985. 32
- [Wik11a] Wikipedia. BigTable - Wikipedia, The Free Encyclopedia. <http://en.wikipedia.org/w/index.php?title=BigTable&oldid=423467455>, 2011. 13, 25
- [Wik11b] Wikipedia. Cloud computing — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cloud_computing&oldid=425003854, 2011. [Online; accessed 20-April-2011]. 5, 19
- [Wik11c] Wikipedia. Peer-to-peer — wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Peer-to-peer&oldid=423442086>, 2011. [Online; accessed 20-April-2011]. 5, 18
- [XCZ⁺11] PengCheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, Calton Pu, and Hakan Hacigümüs. Intelligent Management of Virtualized Resources for Database Systems in Cloud Environment. In *IEEE 27th International Conference on Data Engineering, ICDE'11*, Hannover, Germany, 2011. 124, 137
- [YHP02] Jian Ye, Jiongkuan Hou, and S. Papavassiliou. A Comprehensive Resource Management Framework for Next Generation Wireless Networks. *IEEE Transactions on Mobile Computing*, 1(4):249 – 264, oct-dec 2002. 8, 21
- [You11] YouTube. Website. <http://www.youtube.com>, 2011. 4, 18
- [ZH05] Yingwu Zhu and Yiming Hu. Efficient, Proximity-Aware Load Balancing for DHT-based P2P Systems. *IEEE Transactions on Parallel and Distributed Systems*, 16(4):349–361, 2005. 10, 22, 65, 73, 74

-
- [ZHS⁺04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004. 9, 22

Appendix

On the Elasticity of NoSQL Databases over Cloud Management Platforms

NoSQL databases focus on analytical processing of large scale datasets, offering increased scalability over commodity hardware. One of their strongest features is elasticity, which allows for fairly portioned premiums and high-quality performance and directly applies to the philosophy of a cloud-based platform. Yet, the process of adaptive expansion and contraction of resources usually involves a lot of manual effort during cluster configuration. To date, there exists no comparative study to quantify this cost and measure the efficacy of NoSQL engines that offer this feature over a cloud provider. In this work, we present a cloud-enabled framework for adaptive monitoring of NoSQL systems. We perform a thorough study of the elasticity feature on some of the most popular NoSQL databases over an open-source cloud computing platform.

A.1 Introduction

Computational and storage requirements of applications such as web analytics, business intelligence and social networking over tera- (or even peta-) byte datasets have pushed sql-like centralized databases to their limits [Aba09]. This led to the development of horizontally scalable, distributed non-relational data stores, called NoSQL databases. Google's Bigtable [CDG⁺08] and its open-source implementation HBase [Apa11d], Amazon's Dynamo [DHJ⁺07], Facebook's Cassandra [LM10], and LinkedIn's Voldemort [Vol09] are a representative set of such systems.

NoSQL systems exhibit the ability to store and index arbitrarily big data sets while enabling a large amount of concurrent user requests. They are perfect candidates for cloud platforms that provide infrastructure as a service (IaaS) such as Amazon’s EC2 [Ama11a] or its open-source alternatives such as Eucalyptus [NWG⁺09] and OpenStack [Ope11]: NoSQL administrators can utilize the cloud API to throttle the number of acquired resources (i.e., number of virtual machines – VMs and storage space) according to application needs.

This is highly-compatible with NoSQL stores: Scalability in processing big data is possible through *elasticity* and *sharding*. The former refers to the ability to expand or contract dedicated resources in order to meet the exact demand. The latter refers to the horizontal partitioning over a shared-nothing architecture that enables scalable load processing. It is obvious that these two properties (henceforth referred to as *elasticity*) are intertwined: as computing resources grow and shrink, data partitioning must be done in such a way that no loss occurs and the right amount of replication is conserved.

Many NoSQL systems (e.g., [Apa11d, DHJ⁺07, LM10, Vol09, ria11]) claim to offer adaptive elasticity according to the number of participant commodity nodes. Nevertheless, the “throttling” is usually performed manually, making availability problems due to unanticipated high loads not infrequent (e.g., the recent Foursquare outage [Hor11]). Adaptive frameworks are offered by major cloud vendors as a service through their infrastructure: Amazon’s SimpleDB [sim], Google’s AppEngine [app] and Microsoft’s SQL Azure [azub] are proprietary systems provided through a simple REST interface offering (virtually) unlimited processing power and storage. However, these services run on dedicated servers (i.e., no elasticity from the vendor’s point of view), their internal design and architecture is not publicly documented, their cost is sometimes prohibitive and their performance is questionable [KKL10a].

A number of recent works has provided interesting insights over the performance and processing characteristics of various analytics platforms (e.g., [KKL10a, PPR⁺09, CST⁺10]), without dealing with elasticity in virtualized resources, which is the typical case in cloud environments. The studies presented in [GSRM⁺09, MKF10, XCZ⁺11] deal with this feature but do not address NoSQL databases, while [LBC10] is file-system specific. Finally, proprietary frameworks such as Amazon’s CloudWatch [clo] or AzureWatch [azua] do not provide a rich set of metrics and require a lot of manual labor to be applicable for NoSQL systems. Thus, although both NoSQL and cloud infrastructures are inherently elastic, there exists no actual study to report how effective this is in practice, at least over architecturally different engines.

Our work aims to bridge this gap between individual implementations and practice. Having reviewed the majority of open-source NoSQL solutions (including considerable hands-on experience with many of them) and the Eucalyptus IaaS [NWG⁺09], our task is to design and deploy a distributed framework that allows in a customizable manner *any* NoSQL engine to expand

or contract its resources by using a cloud management platform. Specifically, in this work we perform a thorough study that presents the following tangible contributions:

- Utilizing a VM-based measurement framework, we are able to provide a generic control module that monitors NoSQL clusters. We then identify how each metric of interest (CPU, Memory usage, Network, etc) varies under workloads of various types (reads, writes, updates) and rates.
- We document the costs and gains after a cluster resize. Specifically, using both client and cluster-based metrics, we register the performance gains when increasing the size of the cluster in varying workloads. We also measure the cost in terms of time delay and describe the performance degradation at all stages of adding or deleting a new VM.

Taking advantage of our experience in building and operating this platform, this work may also be used as an invaluable guide to technical issues such as: Automatic VM setup and configuration, NoSQL tuneup and glitches, implementation shortcomings and best workload practices.

To achieve maximum result generalization and show our framework's modularity, we incorporate three popular NoSQL implementations, HBase [Apa11d], Cassandra [LM10] and Riak [ria11] that support elasticity and also offer an acceptable level of development support required. The same hold for the choice of the open-source client [CST⁺10] our system incorporates.

The remainder of this work is organized as follows: the basic system architecture is presented in Section A.2. Section A.3 presents a brief description of the supported NoSQL systems and their elasticity features. Our experimental results are detailed in Section A.4, related work is presented in Section A.5 and Section A.6 contains a thorough analysis of our findings, design decisions and recommendations, while Section A.7 concludes our work.

A.2 Architecture

We now give a brief description of the modules of our elasticity-testing framework: The *Command Issuing* module is used to initiate a cluster resize operation. It interacts with the *Cloud Management* module that contacts the cloud vendor to adjust the cluster's physical resources by releasing or acquiring more virtual machines. The *Rebalancing* module makes sure that newly arrived nodes join the cluster contributing an equal share of the work. The *Cluster Coordinator* module executes higher level add, remove and rebalance commands according to the particular NoSQL system used. Finally, the *Monitoring* module maintains up-to-date performance metrics that collects from the cluster nodes. Below we describe each module in more detail:

Command Issuing Module: This is the 'coordinator' module. In the current implementation phase, this module requests addition or removal of a number of VMs using the Cloud Management and Cluster Coordinator modules.

Monitoring Module: Our system takes a passive monitoring approach. Currently, it receives data from Ganglia [MCC04], a scalable distributed system monitoring tool that allows the user to remotely collect live or historical statistics (such as CPU load averages, network, memory or disk space utilization) through its XML API and present them through its web front-end via real-time dynamic web pages. Apart from general operating-system statistics, Ganglia may also gather NoSQL performance metrics such as the current number of open client threads, number of served operations per second, etc.

Rebalancing Module: The rebalancing module is activated after a newly arrived virtual machine from the cloud vendor has successfully started (i.e., has booted and received a valid IP). When this happens, the module executes a “global rebalance” operation, in which client requests are spread equally among the cluster nodes according to the specific NoSQL implementation and semantics.

Cloud management: Our system interacts with the cloud vendor using the well known euca-tools, an Amazon EC2 compliant REST-based client library. The command issuing module interacts with this module when it commands for a resize in the physical cluster resources, i.e., the number of running VMs. Our cloud management platform is a private Eucalyptus [NWG⁺09] installation. The use of euca-tools guarantees that our system can be deployed in Amazon’s EC2 or in any EC2-compliant IaaS cloud. We have created an Amazon Machine Image (AMI) that contains pre-installed versions of the supported NoSQL systems along with the Ganglia monitoring tool.

Cluster coordinator: The coordination of the remote VMs is done with the remote execution of shell scripts and the injection of on-the-fly created NoSQL specific configuration files to each VM. A higher level “start cluster”, “add NoSQL node” and “remove NoSQL node” command is translated in a workflow of the aforementioned primitives. Our framework implementation makes sure that each step has succeeded before moving to the next one.

For instance, the Command Issuing module requests an “add virtual machine” command using the euca-tools API and waits until it is started and has been assigned with an IP. After this, the Cluster Coordinator creates the appropriate configuration scripts on-the-fly, transfers them to the new VM, and remotely starts the NoSQL service along with the Ganglia tool. The Rebalancer module inserts the node to the cluster and rebalances client requests among the server nodes. In the “remove noSQL node” command, the Cluster Coordinator is instructed to remove it from the cluster calling the “terminate instance” command of the Cloud Management. Data loss during node removal is avoided with the use of NoSQL data replication. In this case, NoSQL systems transparently replicate the removed data in order to maintain the replication factor.

Our framework currently incorporates three popular NoSQL systems that implement rebalancing operations: HBase, Cassandra and Riak (for an overview of these systems, refer to Section

A.3). Yet, the system is extensible enough to include more engines that support elastic operations by implementing the system's abstract primitives in the Cluster Coordinator module and by including the system's binaries to the existing AMI virtual machine image.

A.3 NoSQL Overview

This section provides a short overview of the three NoSQL engines incorporated so far in our framework. In particular, we focus on key architectural characteristics, behaviour and advantages, as these features shed light into many of the results that follow.

A.3.1 HBase

HBase is a Bigtable-like structured store that runs on top of the Hadoop Distributed File System (HDFS). Both HBase and Hadoop implement a centralized master-slave architecture. HBase has an HMaster node that keeps track of the various cluster nodes (*RegionServers*) that serve client requests. Data is divided into regions allocated to the *RegionServers* by HMaster and reside, normally, on the local HDFS *DataNode*. When regions grow above a user-defined limit, they are split in half. Incoming data is cached until a pre-configured size is reached at which point a flush to disk creates a new file. Once the number of newly written files exceeds a configurable threshold a minor compaction is performed. In contrast, a major compaction is scheduled at regular intervals that consolidates all files.

A load balancer is triggered periodically aiming to balance the cluster load by migrating regions as required. Cluster synchronization is accomplished thanks to *Zookeeper*, a distributed coordination service. Clients contact the *Zookeeper* to retrieve the node hosting the necessary metadata for locating the *RegionServer* that owns the corresponding region. Data replication is supported, with the default factor set to 3. HBase inherits a strong level of consistency from HDFS. In the event of new nodes joining the cluster, they are used for storing any new data in the system. Data are not redistributed by default but a load rebalancing can be forced.

A.3.2 Cassandra

Cassandra is a Dynamo-inspired system that follows Bigtable's data model [LM10]. Nodes form a DHT ring with each node holding a specific partition of the key-space and serving as a contact point.

New nodes entering the ring are assigned a partition of the data stored in the cluster, namely, half the key-space of the node with the largest partition is transferred to it. Therefore, no more than the number of nodes present in the ring can be inserted at the same time. Data is replicated for fault-tolerance, with the default replication factor being equal to 3. Cassandra is optimized for

fast write operations. There exist five levels of consistency available for every operation executed. Eventual consistency can favour fast writes whereas strong consistency can be achieved if needed.

A.3.3 Riak

Similarly to Cassandra, Riak is strongly influenced by the Dynamo paper [ria11]. Featuring a DHT architecture, Riak focuses on scalability and high availability supporting adjustable data replication for durability (the default replication factor is 3) and tunable levels of consistence. Data rebalancing is handled automatically as soon as new nodes enter the ring, aiming in dividing data partitions into equal shares and distributing them amongst nodes.

A.4 Experimental Results

Our experimental setup consists of a Eucalyptus 2.0.0 private cluster of 16 Node controllers and a single machine in the role of Cloud and cluster controller. We were allocated enough resources for a cluster of 20 client VMs (load generators) and 28 server VMs. Each server VM has a 4 virtual core processor with 8GB of RAM and 50GB of storage space, while client VMs have 2 virtual CPUs and 4GB of RAM. Cluster peers store their data into their root file system, i.e., no external Elastic Block Storage (EBS) is used. The versions of Hadoop and Ganglia used are 0.20.2 and 3.1.2 respectively, both in their default configuration.

Clients and workloads used: We utilize fixed HBase (v. 0.20.6) Cassandra (v. 0.7.0 beta) and Riak (v. 0.14.0) initial cluster sizes of 8 nodes which are loaded with 20 million objects (i.e., 20GB of plain raw data, since each item takes up 1KB) by utilizing the YCSB [CST⁺10] load function. Every database is configured with a replication factor of 3, which in the case of Cassandra, HBase and Riak results in a total database size of about 60GB, 90GB and 105GB respectively (HBase and Riak use more metadata per record). Since HBase and Cassandra are written in java and they were setup for production mode, a generous heap space of 6GB was supplied. For the most part, all database systems were setup using their default settings, as presented in their online manual pages. The only deviation from this rule is Cassandra's `auto_bootstrap` parameter, which was set to `false`, as it effectively prevents adding more nodes to the cluster ring than the number of already participating nodes.

Our default workload for use through the YCSB tool is a straightforward random uniform read, varying λ appropriately, where λ is the number of active threads of the YCSB tool. YCSB uses two parameters, the number of threads per client and the target number of operations per second they will try to achieve. Consequently, λ defines the number of concurrent pending operations at every time point. In our experiments, we pose a sufficient number (2M to 10M) of simple get queries that collectively cover a significant part of the dataset (10% to 50%).

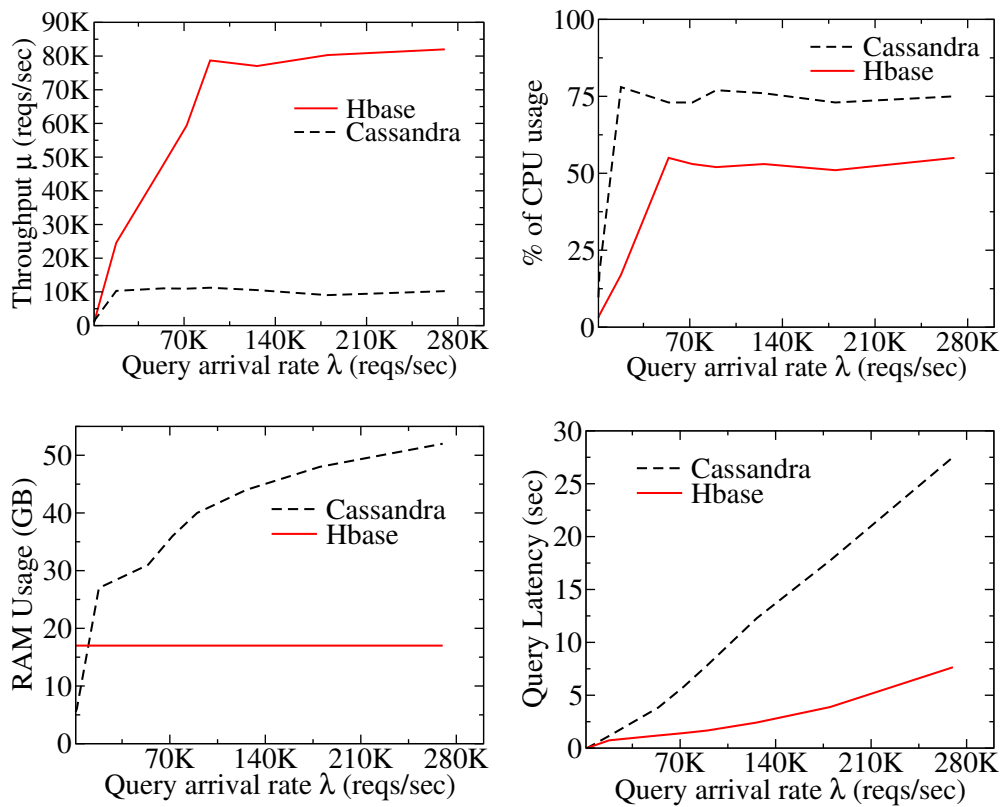


Figure A.1: CPU-RAM usage and mean query throughput-latency for various λ query rates of the UNIFORM_READ workload for an HBase-Cassandra Cluster of 8 nodes.

Although YCSB comes with a set of workloads that simulate real-world conditions, we use 4 consistent workloads (namely UNIFORM_READ, ZIPFIAN_READ, UNIFORM_UPDATE and UNIFORM_RMW) in order to better understand the behaviour of the databases for different types of load. These correspond to simple (i.e., not composite) workloads, where all operations are of the defined type, that is uniform random reads, zipfian random reads, uniform random updates and uniform random read-modify-writes respectively.

Our evaluation conforms to the following methodology: First, we plan to identify the performance metrics that are affected under heavy load of concurrent requests and various workload types. Finally, we plan to identify the cost and performance gain/loss cost after cluster resize for various workloads and resize choices.

A.4.1 Metrics affected during stress-testing

In the first set of experiments, we measure how a number of metrics is affected under variable load. We are interested both in generic, server-side metrics (i.e., CPU load and memory usage)

and application-specific, client-side ones (i.e., query throughput and mean query latency), varying the aggregate query rate as well as the workload type. Figure A.1 presents the results for four metrics, two reported by YCSB (throughput and latency) and two by Ganglia (CPU and Memory usage) for values of aggregate λ up to 280 Kreqs/sec.

Considering Riak, the 8-node cluster could not accommodate such big rates of λ . Riak was able to achieve an average throughput of 10 Kreqs/sec for an 8-node cluster, which fully complies with previously reported performance measurements¹. Yet, above this rate, extra requests are dropped and Riak servers become unresponsive. Thus, we were unable to directly compare Riak with HBase and Cassandra, limiting its participation to the data rebalancing experiment in Section A.4.3.

The maximum throughput for the other two databases can be derived from the first graph of Figure A.1. We notice that HBase achieves a maximum aggregate throughput of about 80 Kreqs/sec at $\lambda \simeq 80$ Kreqs/sec, while Cassandra achieves its maximum throughput of 13K at $\lambda \simeq 20$ Kreqs/sec. The variation of the CPU load as perceived from the cluster seems in accordance with the client-perceived throughput. The maximum usage is a bit over 55% (aggregated over the whole server cluster) for HBase and $\simeq 76\%$ for Cassandra, with the load remaining constant henceforth. As seen in both experiments, increasing λ further has no effect, as both systems are fully utilized. Thus, further arriving requests are simply queued, to be answered later by the system, resulting in mostly flat CPU and throughput curves.

In terms of memory usage, the HBase cluster uses a constant 17 GB of memory while Cassandra seems to take up more memory as the load increases. This could be due to the fact that Cassandra, unlike HBase, has no central node that directs the clients to the appropriate server to get a particular tuple. The server responsible is reached after searching the ring and returns the tuple to the client. This process can create several cached results. To force a memory cleanup, a restart of the cluster is necessary, which results in an original memory usage of 5 GB. Finally, in terms of the average read latency per query, HBase predictably outperforms Cassandra. Both systems exhibit a linear increase in the client-side perceived latency with Cassandra's rate being noticeably higher. The linearly increasing latency curves can be explained from pending server requests which have to wait more since λ increases but the throughput remains constant.

Having determined the way the metrics were affected during the uniform reads for variable λ , we monitor the steady state values of these metrics for different types of workloads. In Table A.1, we present the relevant results for a value $\lambda = 180$ Kreqs/sec. As shown, the average CPU and RAM usage is almost identical for HBase for uniform and zipfian read workloads, while average CPU usage slightly decreases in the case of updates. Cassandra on the other hand exhibits more variation between uniform and zipfian read, with zipfian read producing lower CPU load and

¹<http://bit.ly/lEpqWC>

Table A.1: CPU-RAM usage and mean query throughput-latency under different workload types with a fixed query rate of $\lambda = 180$ Kreqs/sec for an HBase-Cassandra Cluster of 8 nodes.

Workload	Metric	CPU (%)	MEM (GB)	Thr/put (Kreqs/sec)		Latency (sec)
		HbCass	HbCass	Hb	Cass	HbCass
Uniform READ		55 73	17 48	75.5	9	3.9 18.7
Zipfian READ		55 68	17 28	77.4	8.6	1.4 18.4
UPDATE		45 74	24 36	30.4	12.2	4.1 10.8

memory usage. Update operations produce slightly more load but are less heavily dependent on memory. In all cases, Cassandra uses more memory and produces significantly more CPU load than HBase. In terms of average throughput and latency, HBase outperformed Cassandra for short term experiments, even in the case of updates.

A.4.2 Cost of adding and removing nodes

The cost of adding or removing nodes in a NoSQL cluster is mostly measured in time and can be divided in four parts:

VM initialization: Launching new virtual machines takes a significant part of the addition phase. Even when the virtual machine image is cached on the VM container, the VM is available after about 3 minutes, allowing for OS boot time and DHCP negotiation. However, multiple node addition can be done in parallel on multiple VM containers even when multiple VMs are launched on the same container. This means that the previously reported time remains constant. VM removal is done instantaneously (i.e., in less than 10 secs), since it is a much more straightforward operation for the cloud management system.

Node reconfiguration: This phase involves the creation of various configuration files and their propagation to all nodes on the cluster. This is necessary because in both existing and new nodes the configuration files should match and because there are a number of settings that have to be available on both new and existing nodes (for example, hard coded domain name resolution effected by altering the `/etc/hosts` file of all nodes). Given the fact that configuration files are usually small in size, completing this phase takes at most 30 seconds even for large cluster sizes. This phase is necessary during node addition as well as during node removal.

Region rebalancing: This part of the addition/removal process involves the necessary time for the new nodes to become actual serving parts of the cluster. This consists of the time consumed by launching the services/daemons using the database's default scripts, the time for the new node to become active during addition and the time for the data regions to be allocated to each new node during addition or to an old node during removal. The total time for HBase could vary depending on the number of new or removed nodes and the amount of regions, but in our

Table A.2: *Completion time, total moved data, final average query throughput and latency for a 8+8 node cluster resize operation in HBase, Cassandra and Riak with and without data rebalancing*

Metric \ Cluster	HBase		Cassandra		Riak
	Reb	No	Reb	No	Reb
Completion time (min)	98	5	665	5	150
Data moved (GB)	22.5	-	87.7	-	44.8
Throughput (Kreqs/s)	154.5	129.6	18.3	14.9	18
Avg. Latency (s)	0.7	1.1	7.1	9.3	0.2

experiment we have measured times of at most 2 minutes. Given the distributed nature of Cassandra and Riak, the time to add/remove new nodes to the ring is at most 30 seconds irrespective of database size.

Data rebalancing: Data rebalancing is expensive in terms of extra load on the servers and the network infrastructure. It also invalidates data blocks while operations are performed on the cluster as we have witnessed in our tests. HBase and Cassandra can add nodes to the cluster without moving the relevant data. The operational correctness in this case is achieved through the distributed HDFS filesystem in HBase’s case and through extra hops in Cassandra’s case. In Riak, data rebalancing cannot be avoided when bringing new nodes online. This operation, in any case, depends heavily on the amount of data that have to be moved, the number of pre-existing nodes and the number of new nodes that have to fetch the existing data. This means that all cases would have to be treated individually. However, a relevant test outlining performance gains and time costs is presented in Section A.4.3. In general, in the rebalancing tests we performed, we observed erratic cluster behaviour and large time costs.

A.4.3 Cluster resize performance measurements

Our first concern on the costs and gains of a resize operation relates to the rebalancing of the database data. Since data rebalancing is by itself a resource intensive procedure, we only perform node additions and data migration in an idle cluster for this case, i.e., without applying extra client workload. This scenario is valid, since data rebalancing is usually scheduled for off-peak time periods. Nevertheless, in our experiments conducted with extra client workload during data rebalance, all systems exhibited erratic behaviour, and never reached the performance in throughput or latency of the initial cluster before the resize operation. Even worse, disconnects and inconsistencies were propagated to the clients, resulting in a significant number of exceptions.

In Table A.2 we present our results for an original examination of the costs and gains of data rebalancing for HBase, Cassandra and Riak. We have started an 8-node cluster in each case and added 20M tuples. After the data insertion, we expand each cluster by adding 8 more nodes, let

the systems stabilize without performing data rebalance and apply a UNIFORM_READ workload with a $\lambda = 180$ Kreqs/sec. These results are depicted in the “No” column for every database (for Riak that rebalances data automatically as soon as a new node joins the ring, this is not applicable). After this, we perform a manual data rebalancing operation in HBase (through the HDFS balancer²) and Cassandra (through `loadbalance` commands³). When data rebalancing finishes, we apply a UNIFORM_READ workload with a $\lambda = 180$ Kreqs/s for HBase, Cassandra and 18 Kreqs/s for Riak (Riak could not operate with a higher workload). The results for this setup are presented in the “Reb” column for every database.

As can be deduced from Table A.2, the data rebalancing costs far outweigh its benefits for the cases of HBase and Cassandra. For HBase, which also required a restart of the whole cluster that took about 4 minutes, a net gain of about 20% in throughput (154.5 Kreqs/sec balanced vs 129.6 Kreqs/sec not balanced) was achieved compared to a non-rebalanced 16 node cluster, which could be easily offset by adding two extra nodes without using data rebalancing. In Cassandra’s case, the results were better with a net gain of 22% for the average throughput (18.3 Kreqs/sec vs 14.9 Kreqs/sec for a non balanced 16 node cluster). In terms of latency, similar performance benefits were achieved (33% and 23% for HBase and Cassandra respectively). The data moved during data rebalancing for HBase are roughly 22 GB or 25% of the entire dataset. In Cassandra’s case, a much larger number of 87.7 GB of data are moved, which translates to the whole dataset. HDFS’s centralized balancer is more advanced than Cassandra’s when deciding how many data blocks to move to the new nodes since it tries to minimize the moved data by taking into account replica locations. Although this leaves the cluster unbalanced in terms of disk usage, it allows a rebalancing operation to complete in much less time than Cassandra. Cassandra, on the other hand, by halving the ID space of a pre-existing node and assigning it to the newly arrived one, uses a more naive but decentralized approach to balance the cluster, moving more data.

As far as Riak is concerned, rebalancing is also a costly operation both in time, taking 150 minutes to complete in our experiment, and in network bandwidth since 44.8 GB (or 43% of the entire dataset) were transferred to the newly added nodes.

Given the large time costs for data rebalancing operations (93, 660 and 150 minutes in HBase, Cassandra and Riak respectively), and the fact that the benefits can be outweighed by adding more nodes without data migration with a fixed time cost of 5 minutes for each database (see the following results), we conclude that gains do not outweigh costs when aiming for the highest degree of elasticity. Therefore, subsequent tests were run without performing data rebalancing operations.

²<http://bit.ly/iCNsbF>

³<http://bit.ly/kEZZFI>

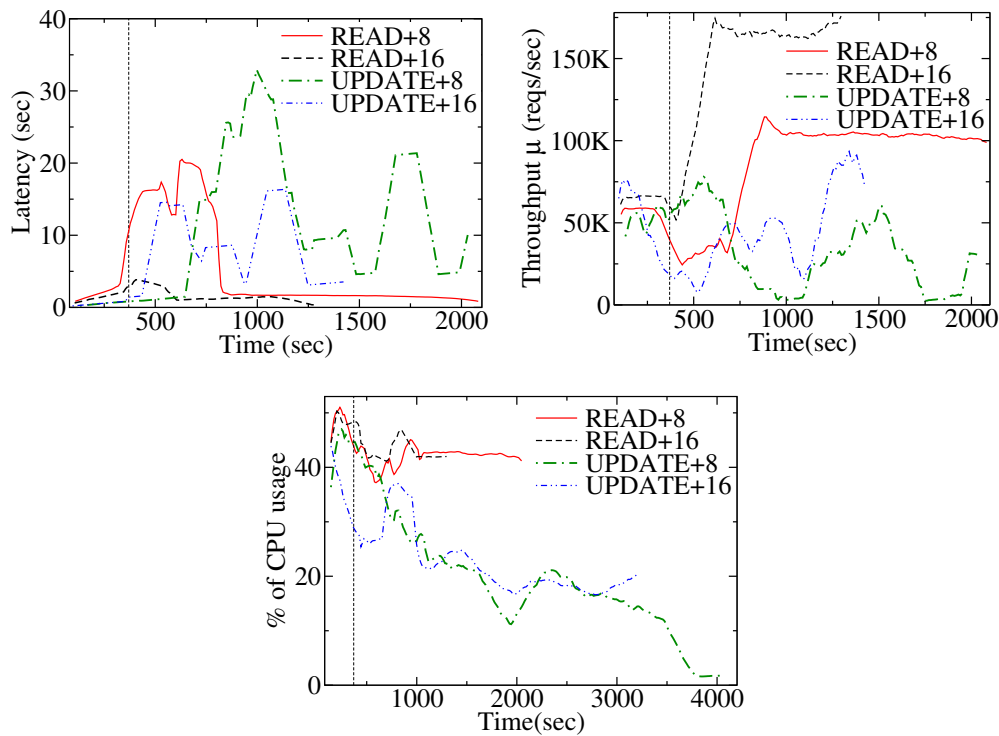


Figure A.2: Query latency, query throughput and CPU usage per time for an HBase cluster of 8 nodes after adding 8 and 16 more nodes for the UNIFORM_READ and UNIFORM_UPDATE workload with a query rate of $\lambda = 180$ Kreqs/sec

We continue our evaluation of the gains in performance when adding nodes under varying workloads without data rebalancing for HBase and Cassandra. Our results describe the behaviour of an 8-node cluster under varying workloads after the addition of a variable number of nodes. We utilize a YCSB-generated load of $\lambda = 180$ Kreqs/sec, which, as previously shown, is well over the maximum load that both Cassandra and HBase can efficiently handle. Two types of workloads are used, UNIFORM_READ and UNIFORM_UPDATE (referred to as READ and UPDATE henceforth). For each combination of workload and database, we perform an addition of 8 (i.e., double the size) and 16 (i.e., triple the size) nodes. The cluster resize occurs at about $t = 370$ sec (shown in Figures A.2 and A.3 as a vertical line). The client-side query latency and throughput μ are measured as well as the total aggregate cluster CPU usage reported by the Ganglia tool. As we have shown, CPU usage is highly indicative of the cluster's status.

Figure A.2 presents our results concerning the HBase cluster. Legends refer to the workload type along with the resizing action (e.g., READ+8 represents a read workload with an 8-node resize). With read operations being faster than writes, we achieve comparable run times by adjusting the amount of objects requested in each workload, i.e., 10M in READ and 4M in UPDATE. The first graph plots present the mean query latency. Adding nodes during READ

loads (READ+8 and READ+16) has a transient negative effect on the query latency, attributed to the fact that HMaster reassigns table regions to the new nodes as soon as the resize occurs. Although data is not actually moved (as explained in section A.3.1), this reassignment poses an extra burden to the already overloaded cluster. Clients cache region locations which however change during cluster resizing. As a result, latency increases due to client cache misses. Nevertheless, this effect lasts for around 4-5 minutes when adding 8 nodes, while only a couple of minutes when adding 16 nodes (with query latency less affected during this interval). This is because more servers quickly join the cluster and take a large portion of the applied load. The total 24 nodes are now adequate to handle the load. Therefore the transient period takes less time and affects the clients at a smaller degree. In the update workloads, we notice an oscillation in both cases: This happens because of the compaction and caching mechanisms of HBase (see Section A.3.1). Incoming data is cached in memory (resulting in low update latencies) but when the memory is full and a I/O flush occurs with a compaction, the latency is increased until the new blocks are written to the file system.

For the client-side throughput μ , depicted in the second graph, we notice a clear gain when adding nodes in read workloads: after the transient phase, adding 16 nodes results in about 170 Kreqs/sec in the steady state and 100 Kreqs/sec for the READ+8 case (compared to about 55K before additions). More servers are able to simultaneously handle more requests that results in a higher throughput (roughly doubling and tripling it respectively). Although items are not actually transferred, this speed is due to the caching effect of the RegionServers: New nodes eventually cache their assigned regions and do not need to contact the nodes that store them. For the update workloads, we notice small improvement in adding 16 compared to adding 8 nodes, but since the update is an I/O-bound operation, μ is not significantly altered. As initial data is not moved, a portion of incoming updates will be handled by the initial nodes. Only when new regions are created, due to a minor compaction for instance, I/O operations will be handled by the new nodes.

The final graph reports the aggregate cluster CPU usage as registered by the Ganglia tool. In the read workloads we notice that the initial load of around 55% is reduced to around 42% in both cases. Evidently, the newly arrived nodes immediately start handling incoming queries and alleviate the initially overloaded cluster. The addition of 16 against 8 nodes does not result in a further decrease in the average CPU, as the load is still large enough for all servers to contribute. The extra 8 nodes make a difference in terms of throughput, as shown in the second graph. Contrarily, a drop in CPU usage is a good indication for adding servers against the maximum load. In the update workloads, we notice that in both experiments the initial CPU load continues to drop until run completion. This drop is due to the slow writes that occur during updates: The server freezes incoming requests until the updated regions are flushed to the file system.

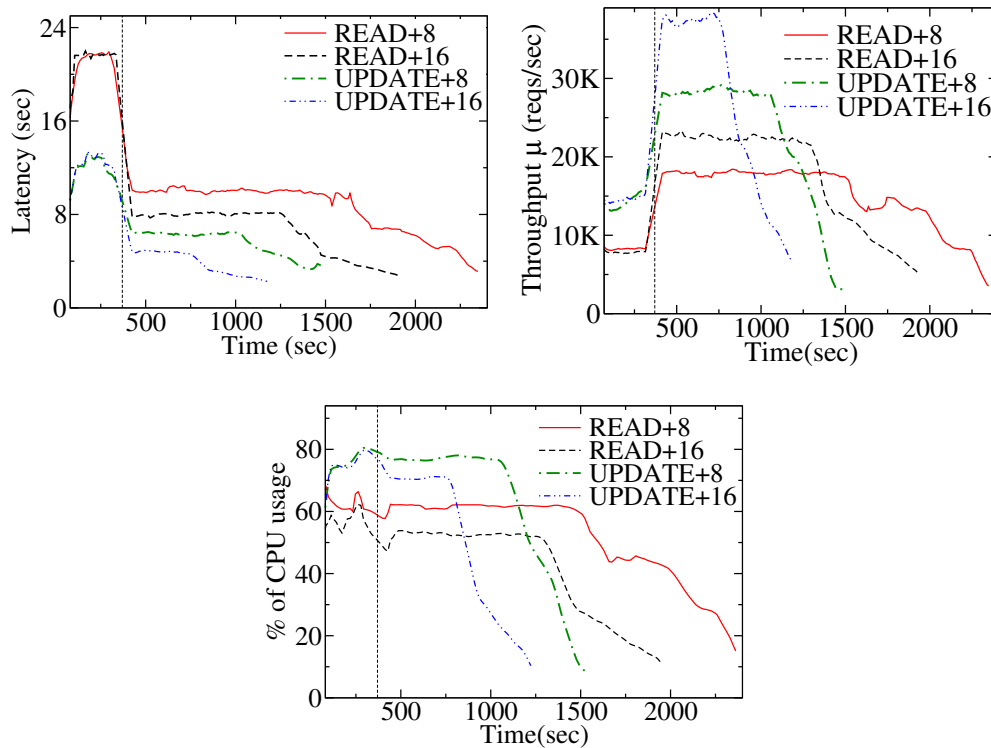


Figure A.3: Query latency, query throughput and CPU usage per time for a Cassandra cluster of 8 nodes after adding 8 and 16 more nodes for the UNIFORM_READ and UNIFORM_UPDATE workload with a query rate of $\lambda = 180$ Kreqs/sec

In Figure A.3 we present the respective results for the Cassandra cluster. In each test, clients request 2M each out of the total 20M objects stored in the database. During the experiment, clients are aware of the up-to-date list of server addresses and each request is directed to a random one. Since Cassandra does not have a mechanism to inform clients about new servers, a custom script was developed to propagate these changes back to the clients, whenever a cluster resize occurred. The first graph presents query latency: In both READ cases, we notice that the latency almost immediately drops from an initial value of around 22 secs to 10 secs and 8 secs respectively: New servers are assigned half of the data partitions of existing servers, they cache portions of their data and answer queries on their behalf. The larger the resize, the bigger the decrease in latency. The same hold for the update workloads: Adding more nodes reduces the query latency from 11 sec to around 7 in the UPDATE+8 case and to around 5 in the UPDATE+16 case. Again, we notice here that writes are in general faster than reads: This is due to the weak consistency model followed by Cassandra, where writes do not have to be propagated to every replica holder for the operation to succeed.

Query throughput μ shows a similar trend. Both in read and update workloads we notice the almost linear effect of adding new nodes. Read throughput is increased from an initial value

of 9 Kreqs/sec to 18 Kreqs/sec when 8 nodes are added and to 22 Kreqs/sec when 16 nodes are added. Update throughput is increased from around 15 Kreqs/sec to 29 Kreqs/sec in the +8 case, and to 35 Kreqs/sec in the +16 case. This behaviour is expected, since extra servers immediately join the p2p ring and take portions of the applied workload. Moreover, the asynchronous nature of eventual consistency enables Cassandra to maintain a stable throughput rate even in a write-heavy workload: Updates are successful upon transmitting the object to a single server, which replicates it later on in a lazy manner.

Finally, in the third graph of Figure A.3 we present the variation of the total cluster's CPU usage during time. In the read case, we notice that adding 8 nodes slightly decreases the initial CPU usage to around 60%, whereas adding 16 extra servers decreases the average CPU load to around 50%. Similar to the HBase case, the 50% load of the 24-node cluster shows that the applied load is big enough for every server to contribute, since new servers are not idle. The same hold for the update workloads: Adding 8 nodes brings the load to around 80% and the addition of 16 nodes drops the load to around 70%. Update workloads are more computationally heavy than simple reads, as in the update case there is an extra cost of disk access that is avoided during reads by caching fetched results.

In this kind of setting, we note how both NoSQL systems take advantage of the addition of extra nodes: HBase exhibits very fast concurrent reads compared to Cassandra: In the READ+16 case it can handle 160 Kreqs/sec with a latency of about 2–3 secs and an aggregate CPU usage of 40%, whereas Cassandra's throughput is 40 Kreqs/sec, a latency of around 8 secs and a higher CPU usage of 50%. On the other hand, Cassandra is more efficient with object updates: It maintains stable throughput and latency curves avoiding oscillations that occur with HBase. Finally, we notice that Cassandra does not exhibit a negative transient effect when new nodes enter the ring. Its decentralized nature allows for a transparent cluster resize, whereas in HBase the HMaster needs to coordinate the whole procedure.

A.5 Related Work

A literature study about the challenges of scaling whole applications in cloud environments is presented in [VRMB11], along with an overview of state of the art efforts towards that goal. In [GSRM⁺09], the authors propose a service specification language for cloud computing platforms that handles automatic deployment and flexible scaling of services. Similarly, the work in [MKF10] designs a model implementing an elastic site resource manager that dynamically consolidates remote cloud resources on demand based on predefined policies and a prototype for extending Torque clusters. The works in [XCZ⁺11, SMA⁺08] solve the problem of optimizing VM resources (CPU, memory, etc) to achieve maximum performance using relational DBs. In comparison, we use the standard VM model that large cloud providers currently offer. As a

general observation, these works do not address NoSQL systems and their performance under (dynamic) cluster resizes.

A thorough analysis of various proprietary NoSQL-like cloud services, namely Google’s AppEngine, Microsoft’s SQL Azure and Amazon’s SimpleDB, is presented in [KKL10a]. The authors test system aspects such as scalability, cost and performance utilize the TCP-W benchmark. All systems are treated as “black boxes”, since no information about their design or implementation is assumed to be known. In contrast, our system is fully aware of the different engines’ inside mechanisms.

Cloudy [KKL⁺10b] is a cloud-enabled framework which supports auto-scaling features according to demand, providing simple key/value put/get primitives. In NEFELI [TRFD10], cloud users inform the cloud management framework with hints about the application nature and the framework modifies its scheduling policies to improve the application performance.

CloudWatch [clo] is Amazon’s commercial product for monitoring and managing cloud resources. CloudWatch offers a set of metrics for every VM instance and a policy framework to trigger balancing actions when some conditions are met. Its metric support is limited, offering only hypervisor-related information such as CPU usage and network traffic. Memory usage or application-specific metrics are not supported out of the box, in contrast to Ganglia which, apart from its own rich set of probes, has inherent support from many applications, including some NoSQL systems. Other frameworks for commercial cloud platforms like AzureWatch [azua] feature similar characteristics, resulting in expensive vendor lock-ins.

A.6 Discussion

As our architecture and experimental section makes clear, there exists a number of often conflicting factors that our framework needs to take into account. During our design and implementation, we have observed that there are numerous pitfalls and dubious assumptions about several components, e.g., about the elastic capabilities of NoSQL databases. In the following, we argue on the design choices and offer recommendations based on our experience in setting up this system.

A.6.1 Monitoring module

Certain design decisions have to be made concerning the monitoring module. The most important of them are the type of monitoring, the relevant metrics that will be monitored, and the avoidance of single points of failure. Monitoring can be either active, achieved by periodically injecting probing queries, or passive, thus collecting statistics from already posed user queries.

Similarly, monitored metrics could include either general-purpose metrics (such as network traffic, memory and CPU usage per VM) or high-level application-specific metrics (such as mean query response time), and they should ideally be collected in a fast and scalable way.

Active monitoring implies running a specialized tool for determining the state of the database cluster as a whole, while passive monitoring could use statistics exported from the VMs and database-specific metrics reported from the database system. Our design has opted for the passive monitoring solution without database-specific metrics for two reasons; speed and accuracy. Active monitoring entails a significant trade-off between speed, accuracy and system abuse. This means that executing long running tests would interfere with the system's normal operation stressing or altering it (in the case of writes). On the other hand, if small tests were to be chosen, they would fail to spot possible stress on very active data regions, giving an inaccurate state of the cluster's performance. Results of Section A.4.1 show that we can fairly accurately distinguish both the critical state of each database system and the amount of load that currently stresses the system using passive, general purpose metrics. We have selected to utilize the VM-related metrics for generality, because not every database uses the same metrics or reports them in a comparable manner.

Passive data-collection through Ganglia, besides using a unified way to report accurate statistics in human readable format, is easy to setup in an elastic way, requiring only a single configuration file to be propagated amongst different monitored VMs. Ganglia offers proven scalability. Data can be collected remotely for individual hosts or for the whole cluster by probing the metadata aggregation service (gmetad). These collected statistics are then remotely accessed and evaluated in order for a system administrator or automated system to decide on appropriate rebalancing actions.

A.6.2 Database elasticity

Deciding the best way to elastically alter the cluster size for each database, one has to take into account each DB's characteristics. This includes data and region rebalancing, the ability to expose changes to the DB's clients and avoid data loss when scaling down. Although HBase, Cassandra and Riak claim that they behave in an elastic way, the practicalities of scaling up or down differ in each system due to their architectural differences, and therefore affect the performance gain in each case.

The ability to automatically rebalance different database regions (or shards in RDBMS terminology) is crucial for any elastic system. In HBase, region rebalancing is automatically performed by the HMaster when new nodes are added or removed from the cluster, and the Zookeeper is responsible for propagating and resolving conflicts between RegionServers, as per the HMaster's decision. Given that all RegionServers operate on top of a shared file system (HDFS) there is no

preferential treatment when assigning new regions, thus HMaster can easily deal the available regions in a fair way. Therefore, HBase is extremely elastic, as all new nodes can quickly assume load, increasing the cluster's performance in very short time, as reported.

Conversely, Cassandra does not split data into regions of equal size. As a decentralized system, it reassigns regions on a per node basis, i.e., region rebalancing is performed in node pairs between newly arrived and previously existing ones. Every new node acquires and is responsible for serving half the key-space of an existing one, meaning that for each request it has to retrieve data from the previous owner. Despite the fact that retrieved data are cached, new nodes are practically much slower, as they have to rely on existing nodes for results. In order for a new node to assume all the data corresponding to its key-space, a data rebalance operation has to be performed. Although our experiments show a 22% boost in terms of throughput compared to an unbalanced cluster of the same size, data migration took more than 11 hours and moved almost the entire dataset of 90GB. The need for such an expensive operation limits the elasticity of a Cassandra cluster for short-term load variations.

Riak, on the other hand, divides data space into partitions and as new nodes are added, they claim an equal slice of partitions so that both data and requests are allocated evenly to all nodes participating in the ring. However, the fact that the cluster turns unresponsive for a throughput higher than what its nodes can handle, prevents us from examining its elasticity under high load. Overall, the important question is whether data rebalancing operations are necessary and when. In our opinion, data rebalancing should only be performed when the system administrator can accurately predict that the load will remain constant for a large amount of time, that is in the order of days. In this case, a data rebalancing operation should be performed, even if it affects peak system performance, as the overall gain justifies the time and performance cost of the rebalancing. On the contrary, variable and unpredictable load in the order of hours does not justify such an expensive operation, and should be avoided.

In this work, our goal is to test NoSQL systems undergoing extremely elastic operations under variable high load. This behaviour can be achieved using Cassandra and HBase, although the results are not similar. HBase's operation over a similarly elastic, shared file system, and infrequent compactions provide the best results.

Elastic behaviour has to also be exploited on the client side, which means that the clients can be made aware of the addition of new servers. This is achieved automatically in HBase's case, as clients negotiate with the HMaster. However, Cassandra and Riak need an explicit load-balancer external to the database system to simulate such a behaviour, affecting performance in terms of latency.

Finally, there is the matter of avoiding data loss during node removal. To avoid data loss, there should remain at least one replica during node removal. For this guarantee, only 2 nodes can be removed at once, in the case of a replication factor of 3. What is more, after the removal, NoSQL

systems must deal with the degraded replicas. Since Cassandra and HBase do not perform this automatically, it must be manually handled by the elasticity framework by explicitly issuing data rebalancing commands, which is, as we have shown in A.4.3, a costly procedure.

A.7 Conclusions

In this work we presented a thorough study to quantify and analyze the costs and gains of various NoSQL cluster resize operations, utilizing three popular NoSQL implementations. To this end, we designed and implemented a fully modular and cloud-enabled framework that allows efficient resource monitoring and direct interaction with the cloud vendor and the cluster manager.

During this study, lots of valuable lessons were learned and much experience was gained relative to both cloud-platforms and NoSQL engines. A primary concern relates with the provisioning of a platform-agnostic, non-intrusive and metric-complete monitoring module. Our Ganglia choice offers a complete set of metrics reported in a scalable manner, even with native NoSQL support (e.g., for HBase). Moreover, requiring just an image from a NoSQL engine enables different implementations being tested over the same framework.

The ease of setup and the performance under elastic operations weigh on choosing a particular NoSQL. All three engines we tested can be setup with relative ease. There are few configuration files that need to be injected during launch and most of the operational parameters can be adjusted by altering the appropriate settings. There is no need to provide new configuration files or commands during normal operation, especially during the rebalancing phases. Nevertheless, their quite distinctive behavior and performance under different scenarios make the decision quite application-specific: HBase is the fastest and scales with node additions (only for reads though); Cassandra performs fast writes and scales also, without any transitional phase during node additions; Riak was unresponsive in high request rates, could scale only at lower rates but rebalances automatically; all three achieve small gains from a data rebalance, provided they are under minimal load.