ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Διαχείριση Δεδομένων σε Κατανεμημένα Συστήματα Μεγάλης Κλίμακας για Χρήση σε Εφαρμογές Αναλυτικής Επεξεργασίας

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Αικατερίνη Γ. Δόκα

Αθήνα, Απρίλιος 2011

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Διαχείριση Δεδομένων σε Κατανεμημένα Συστήματα Μεγάλης Κλίμακας για Χρήση σε Εφαρμογές Αναλυτικής Επεξεργασίας

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Αικατερίνη Γ. Δόκα

Συμβουλευτική Επιτροπή: Νεκτάριος Κοζύρης (επιβλέπων)
Παναγιώτης Τσανάκας
Τιμολέων Σελλής

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 28η Απριλίου 2011

Νεκτάριος Κοζύρης Αναπ. Καθηγητής Ε.Μ.Π.	Παναγιώτης Τσανάκας Καθηγητής Ε.Μ.Π.	Τιμολέων Σελλής Καθηγητής Ε.Μ.Π.
Μανόλης Κουμπαράκης Αναπ. Καθηγητής Ε.Κ.Π.Α.	Νικόλαος Παπασπύρου Επικ. Καθηγητής Ε.Μ.Π.	Ιωάννης Κωτίδης Επικ. Καθηγητής Ο.Π.Α.

Ιωάννης Ιωαννίδης
Καθηγητής Ε.Κ.Π.Α.

Αθήνα, Απρίλιος 2011

Αικατερίνη Γ. Δόκα
Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

BI	Business Intelligence – Επιχειρηματική Ευφυία
DHT	Distributed Hash Table – Κατανεμημένος Πίνακας Κατακερματισμού
DoS	Denial of Service – Άρνηση Παροχής Υπηρεσιών
DW	Data Warehouse – Αποθήκη Δεδομένων
ETL	Extract-Transform-Load – Εξαγωγή-Μετασχηματισμός-Φόρτωση
IS	Information System – Πληροφοριακό Σύστημα
IT	Information Technology – Τεχνολογία της Πληροφορίας
LAN	Local Area Network – Τοπικό Δίκτυο Υπολογιστών
MR	MapReduce Programming Model – Προγραμματιστικό Μοντέλο MapReduce
OLAP	Online Analytical Processing – Σε Απευθείας Σύνδεση Αναλυτική Επεξεργασία
OLTP	Online Transaction Processing – Σε Απευθείας Σύνδεση Επεξεργασία Συναλλαγών
P2P	Peer-to-Peer – Δίκτυα Ομότιμων Κόμβων
PC	Personal Computer – Προσωπικός Υπολογιστής

Πρόλογος

Κατά την πολυετή μου παρουσία στο Πολυτεχνείο έγινα μάρτυρας σπουδαίων αλλαγών και μεταβάσεων στον τομέα των Κατανεμημένων Συστημάτων. Από την εποχή του Cluster Computing και των παράλληλων εφαρμογών περάσαμε στην εποχή των δικτύων P2P, αναζητώντας περισσότερη αυτονομία. Λίγο αργότερα γεννήθηκε η ιδέα του Grid Computing, το οποίο ξεκίνησε από μια ανάγκη των επιστημόνων του CERN και παρέμεινε κατά κύριο λόγο προσανατολισμένο στον επιστημονικό κόσμο, ενώ τελευταία μπήκαμε δυναμικά στην εποχή του Cloud Computing, μετενσάρκωση του Grid, που φαίνεται πώς έχει προσελκύσει την προσοχή όχι μόνο του ακαδημαϊκού αλλά και του επιχειρηματικού κόσμου και θα φέρει την πολυαναμενόμενη επανάσταση που τόσα χρόνια ονειρεύονται οι επιστήμονες του τομέα. Καθώς οι εποχές και οι ορολογίες αλλάζουν, αυτό που μένει πάντα ίδιο είναι η ανάγκη που οδήγησε στη γέννηση των Κατανεμημένων Συστημάτων: Να σπάσουμε τα όρια του νόμου του Moore εν όψει των ολοένα αυξανόμενων απαιτήσεων σε υπολογιστική ισχύ, αποθηκευτικό χώρο και εύρος ζώνης. Εύχομαι και ελπίζω η προσπάθειά μου να συνεισφέρει, έστω και λίγο, στην εξέλιξη αυτού του τομέα αλλά και να αποτελέσει εφαλτήριο για περαιτέρω έρευνα.

Η δύσκολη αλλά και συναρπαστική πορεία που με οδήγησε ως εδώ δε θα ήταν η ίδια χωρίς τη συμβολή κάποιων ανθρώπων, στους οποίους οφείλω ένα μεγάλο ευχαριστώ. Πρώτα από όλους, ευχαριστώ τον επιβλέποντά μου, όχι μόνο για την ευκαιρία που μου έδωσε να ασχοληθώ με τεχνολογίες αιχμής, αλλά και για την ελευθερία που μου παρείχε να κάνω τις δικές μου επιλογές, έχοντας σαν δίχτυ ασφαλείας τη στήριξή του. Η πίστη του στις δυνατότητές μου σε περιόδους που ακόμα κι εγώ η ίδια αμφέβαλα μου έδωσε το κουράγιο και την αυτοπεποίθηση να συνεχίζω να προσπαθώ για το καλύτερο. Επίσης, θα ήθελα να ευχαριστήσω τους καθηγητές που με προθυμία ανέλαβαν την παρακολούθηση και κρίση της δουλειάς μου. Η συμμετοχή

επιστημόνων που θαυμάζω στην επιτροπή του διδακτορικού μου αποτελεί για μένα μεγάλη τιμή και ικανοποίηση.

Από το ξεκίνημα της διατριβής μου μέχρι σήμερα, το Εργαστήριο Υπολογιστικών Συστημάτων υπήρξε το δεύτερο σπίτι μου. Όχι μόνο λόγω του χρόνου που πέρασα σε αυτό, αλλά κυρίως χάρη στη ζεστή και φιλική ατμόσφαιρα που δημιουργούν τα μέλη του. Τους ευχαριστώ για την άψογη συνεργασία μας, τις εποικοδομητικές συζητήσεις για ερευνητικά αλλά και φιλοσοφικά θέματα, τη θετική τους διάθεση. Θεωρώ ότι μπορώ να τους αποκαλώ φίλους και τους εύχομαι ό,τι καλύτερο σε όποιο δρόμο επιλέξει ο καθένας.

Δε θα μπορούσα να μην αναφερθώ στους φίλους μου από τα σχολικά και προπτυχιακά χρόνια. Μοιραζόμαστε πολλές ευχάριστες αναμνήσεις και εύχομαι το μέλλον να μας χαρίσει ακόμα περισσότερες. Τους ευχαριστώ για την υπομονή και την κατανόησή τους, αλλά και για την εντυπωσιακή ικανότητά τους να δρουν πάνω μου τόσο αγχολυτικά σε περιόδους πίεσης.

Για το τέλος άφησα τους σημαντικότερους ανθρώπους στη ζωή μου: Τους γονείς μου, που μου προσφέρουν απλόχερα τα πάντα, την αδερφή μου, αιώνια σύμμαχό μου σε κάθε δυσκολία, τον άντρα μου, πηγή της έμπνευσης και της ευτυχίας μου. Καμία λέξη δε μου φαίνεται αρκετή για να εκφράσει την αγάπη και την ευγνωμοσύνη μου. Δε μου μένει τίποτα άλλο παρά να τους αφιερώσω την παρούσα διατριβή.

Κατερίνα Δόκα,
Απρίλης 2011

Abstract

The increasing size of the data collected and generated by industrial and academic information systems has created new sets of demands from data management platforms. Besides the well-documented need for offline analytics, the requirement to immediately detect interesting trends is ever-growing, rendering real-time analytics a necessity. Indeed, data processing applications that incorporate, analyze and extract useful information in near real-time are taking center stage in the enterprise IS infrastructure. In such applications, data are usually determined by a temporal aspect and presented at different levels of granularity. Thousands or millions of such records are produced per second and modern systems are expected to be able to both incorporate and process them.

This thesis deals with the storage, indexing and querying of multidimensional data used for analytical processing in large scale distributed systems and aims to create an always-on, real-time data access and support system. To that end, the basic requirements of such a system are studied and identified: Powerful data processing and high-rate updates. Existing methodologies inadvertently fail to simultaneously meet both these requirements. To alleviate the problem, techniques from the field of distributed data management and data warehousing are applied in order to disseminate, query and update high volumes of multidimensional data characterized by hierarchies. The goal is to maintain the best of both worlds: Powerful indexing/analytics engine for immense volumes of data both over historical and real-time incoming updates and a shared-nothing architecture that ensures scalability and availability at low cost. Geographically spanned users, without the use of any proprietary tool, can share information that arrives from distributed locations at a high rate and query it in different levels of granularity.

The research process towards this goal starts with an attempt, the first to the best of our knowledge, to support concept hierarchies in DHTs, in order to store historical data in various levels of granularity. The resulting system, *HiPPIS*, greatly simplifies the insertion and update operations due to the lack of data pre-processing. Moreover, it employs an adaptive scheme that automatically adjusts the level of indexing according to the granularity of the incoming queries, without assuming any prior knowledge of the workload.

In an attempt to include an a-priori consideration for group-by queries, as well as to explicitly deal with the query performance versus variable data availability or load skew, a well known, highly effective centralized structure is distributed over an unstructured network of interconnected commodity nodes on-the-fly, reducing cube creation and query times by enforcing parallelization. *BrownDwarf* performs online querying and updating and employs an adaptive replication scheme that adjusts to sudden shifts in workload skew as well as network churn by expanding or shrinking the units of the distributed data structure. Thus, the system ensures elasticity of resources and content availability.

Finally, to improve the handling of *time series* data, namely data determined by a temporal aspect, *HORAE* is proposed, a system that employs a hybrid solution for data storage and processing: High-rate updates and queries targeting the most recent items are handled by a DHT-based system that enables fast insertion times and multidimensional indexing. The large bulk of the data is handled through a distributed data cube structure that adaptively materializes and replicates according to demand. The two components seamlessly integrate to offer the advantages of powerful aggregate data processing along with scalability and elasticity of commodity resources.

All of the proposed systems have been implemented and deployed either on a well known simulator or on an actual LAN testbed of commodity PCs. Their extensive experimental evaluation under a variety of datasets, workload distributions and network setups demonstrates their ability to efficiently handle large rates of both updates and queries, tolerate high failure ratios and adjust their indexing structure and their available resources according to demand. A direct comparison with centralized as well as distributed state-of-the-art warehousing solutions proves the advantages of the proposed systems in both performance and elasticity: Query resolution is accelerated, updates are performed online, the load is handled efficiently even after sudden bursts and the functionality remains unaffected with a considerable fraction of frequent node failures.

Εκτεταμένη Περίληψη

0.1 Εισαγωγή

0.1.1 Κίνητρο

Την τελευταία δεκαετία έχουμε γίνει μάρτυρες μιας έκρηξης δεδομένων, που ακόμα βρίσκεται σε εξέλιξη. Καθώς η Τεχνολογία της Πληροφορίας κάνει όλο και πιο αισθητή την παρουσία της σε όλες τις εκφάνσεις της ζωής μας, η ποσότητα των δεδομένων που παράγονται και αποθηκεύονται εξακολουθεί να αυξάνεται με εκπληκτικούς ρυθμούς: Σύμφωνα με την IBM [CCMR06], ο όγκος των δεδομένων παγκοσμίως διπλασιάζεται κάθε δύο χρόνια, ενώ έχει ήδη ξεπεράσει το όριο του zettabyte [IDC10]. Η αύξηση αυτή αποδίδεται τόσο στην εξέλιξη των ίδιων των δεδομένων όσο και στις διαδικασίες παραγωγής και διαχείρισής τους.

Το γεγονός αυτό έχει γίνει εμφανές ακόμα και στην καθημερινότητά μας, όπου η ανάδυση των νέων τεχνολογιών αλλά κυρίως η εμφάνιση του Web 2.0 επιτρέπει στους χρήστες πολλά περισσότερα από την απλή ανάκτηση πληροφορίας: Ο καθένας μπορεί εκτός από καταναλωτής να είναι και παραγωγός περιεχομένου. Όσο οι τιμές του υλικού (από τις συσκευές αποθήκευσης μέχρι τις ψηφιακές κάμερες υψηλής τεχνολογίας) μειώνονται, η ποσότητα των δεδομένων που παράγει ένας μέσος χρήστης υπό τη μορφή μηνυμάτων ηλεκτρονικού ταχυδρομείου, εικόνων, βίντεο, προσωπικών αρχείων κλπ., αυξάνεται ραγδαία. Επιπλέον, με τις οικιακές διαδικτυακές συνδέσεις να κερδίζουν έδαφος, τις ταχύτητες σύνδεσης να αυξάνονται και τις διαδικτυακές υπηρεσίες να γίνονται πιο προσβάσιμες, ογκώδη δεδομένα όπως φωτογραφίες, αρχεία εικόνων και βίντεο κλπ. μπορούν εύκολα να μεταφορτωθούν στο διαδίκτυο. Τα στατιστικά από τα κοινωνικά δίκτυα αποδεικνύουν αυτήν την τάση: Το Facebook [fac] για παράδειγμα, με περισσότερους

από 500 εκατομμύρια ενεργούς χρήστες είναι υπεύθυνο για τη μεταφόρτωση 20 εκατομμυρίων βίντεο και πάνω από 2 δισεκατομμυρίων φωτογραφιών μηνιαίως [fac11], ενώ το [twia] μετρά περισσότερους από 160 εκατομμύρια χρήστες που παράγουν πάνω από 90 εκατομμύρια tweets την ημέρα [twi10].

Πολλές εφαρμογές στον επιστημονικό τομέα, όπως στην βιοπληροφορική, τη φυσική και την αστρονομία βασίζονται κατά ένα μεγάλο μέρος στην ανάλυση δεδομένων που παράγονται με υψηλό ρυθμό και σε μεγάλες ποσότητες από γεωγραφικά απομακρυσμένες επιστημονικές συσκευές όπως αισθητήρες, δορυφόρους, ψηφιακές κάμερες κλπ. Για παράδειγμα, ο μεγάλος επιταχυντής αδρονίων, Large Hadron Collider (LHC), στο CERN [lhc] δημιουργεί δεκάδες terabytes δεδομένων ημερησίως, τα οποία μεταφέρονται σε ακαδημαϊκά ινστιτούτα ανά τον κόσμο σε αναζήτηση του μποζονίου του Higgs [atl, ali]. Ένα άλλο παράδειγμα είναι το Παρατηρητήριο Βαρυτικών Κυμάτων – Laser Interferometer Gravitational Wave Observatory (LIGO) [lig], μία ερευνητική εγκατάσταση της οποίας ο στόχος είναι η ανίχνευση βαρυτικών κυμάτων, παράγοντας 1 TB δεδομένων κάθε μέρα. Λόγω του αυξανόμενου μεγέθους τέτοιων δεδομένων, είναι αναγκαία η ύπαρξη ενός διαχειριστικού πλαισίου για τη διασφάλιση της γρήγορης και αξιόπιστης πρόσβασης από χρήστες που βρίσκονται σε κατανεμημένες τοποθεσίες.

Στον επιχειρηματικό τομέα, οργανισμοί επενδύουν συνεχώς σε πολύπλοκα εργαλεία επιχειρηματικής ευφυίας και ανάλυσης ώστε να βασίζουν τις αποφάσεις τους σε αξιόπιστες πληροφορίες. Ένα από τα πιο δημοφιλή τέτοια εργαλεία είναι η *αποθήκες δεδομένων – data warehouses*. Στις αποθήκες δεδομένων, τεράστιες ποσότητες ιστορικών στοιχείων μαζί με δεδομένα από πολλαπλές λειτουργικές βάσεις αποθηκεύονται και αναλύονται ώστε να αναγνωριστούν μοτίβα συμπεριφορών και να ανακαλυφθούν χρήσιμες συσχετίσεις. Η παγκοσμιοποίηση των αγορών, η αυτοματοποίηση των επιχειρηματικών διαδικασιών και η αυξημένη χρήση αισθητήρων και άλλων συσκευών που παράγουν δεδομένα σε συνδυασμό με τις προσιτές τιμές των τεχνολογικών προϊόντων έχουν συντείνει σε αυτό το φαινόμενο [Sie08]. Πράγματι, πρόσφατη έρευνα [Gro10] αποκάλυψε πως οι μεγάλες επιχειρήσεις υφίστανται κατά μέσο όρο ετήσια αύξηση 32% στον όγκο των δεδομένων τους.

Πέρα από την καλά τεκμηριωμένη ανάγκη για ασύγχρονη ανάλυση, η απαίτηση για άμεση ανίχνευση τάσεων που παρουσιάζουν ενδιαφέρον γίνεται όλο και πιο επιτακτική [Kno09, AFG+09], καθιστώντας την ανάλυση πραγματικού χρόνου μια αναγκαιότητα [dat10]. Για παράδειγμα, επιθέσεις άρνησης παροχής υπηρεσιών – Denial of Service (DoS) attacks – ή εισβολές θα πρέπει να ανιχνεύονται από τους παρόχους διαδικτυακών υπηρεσιών τη στιγμή που συμβαίνουν, ώστε να λαμβάνονται τα απαραίτητα μέτρα για αποκατάσταση των λειτουργιών με την ελάχιστη διακοπή της διαθεσιμότητας των υπηρεσιών [SJ06]. Ένα άλλο παράδειγμα της δύναμης της ανάλυσης πραγματικού χρόνου είναι η χρήση της για την πρόβλεψη της πορείας, της έντασης και άλλων χαρακτηριστικών ενός τυφώνα, ώρες ή ακόμα και μέρες πριν την εμφάνισή

του [PHAML98], εξοικονομώντας πολύτιμο χρόνο για την εκκένωση περιοχών και τη σωστή προετοιμασία, που μπορεί να σώσει ζωές και περιουσίες.

Η αναλυτική επεξεργασία των δεδομένων του παγκόσμιου ιστού σε πραγματικό χρόνο είναι μια ιδιαίτερη κατηγορία, με μια πληθώρα εμπορικών προϊόντων (π.χ. Clicky [cli], Woopra [woo], Chartbeat [cha], κλπ.) που διατείνονται ότι εποπτεύουν την επισκεψιμότητα μιας ιστοσελίδας αλλά και παρακολουθούν σε πραγματικό χρόνο τις συμμετοχές κάποιου χρήστη σε εφαρμογές κοινωνικής δικτύωσης. Επιπλέον, αναλύοντας τον επονομαζόμενο "ιστό πραγματικού χρόνου" ("real-time Web"), όπως για παράδειγμα το περιεχόμενο από tweets, blogs και ιστοσελίδες επικαιρότητας, μπορούν να εξαχθούν πολύτιμες πληροφορίες σχετικά με συμπεριφορές και συναισθήματα: Ένα δημόσιο πρόσωπο μπορεί να εκτιμήσει τη δημοφιλία του (Twittercounter [twib]), οι αναλυτές της αγοράς μπορούν να υπολογίσουν τον αντίκτυπο της κυκλοφορίας ενός προϊόντος (WebTrends [webb], WebAbacus [weba], κλπ.), ακόμα και πολιτικοί υποψήφιοι μπορούν να εξάγουν συμπεράσματα για την πολιτική κατεύθυνση των πολιτών και να προβλέψουν τον νικητή μιας εκλογικής αναμέτρησης [DS10, MM10]. Αναγνωρίζοντας τη δύναμη της ανάλυσης σε πραγματικό χρόνο δεδομένων που προέρχονται από συστήματα όπως το Twitter, τα οποία είναι αφ'εαυτού πραγματικού χρόνου, η ομάδα ανάλυσης του Twitter πρόσφατα (το Φεβρουάριο του 2011) παρουσίασε το Rainbird [Wei11], ένα σύστημα αναλυτικής επεξεργασίας μεγάλου όγκου δεδομένων που, πέρα από το γεγονός ότι είναι πραγματικού χρόνου, κλιμακώνεται οριζόντια.

Έτσι, οι εφαρμογές επεξεργασίας δεδομένων που εξάγουν, αποθηκεύουν και αναζητούν χρήσιμη πληροφορία σε (σχεδόν) πραγματικό χρόνο λαμβάνουν ιδιαίτερη θέση στα πληροφοριακά συστήματα των εταιριών. Σε τέτοιου είδους εφαρμογές, τα δεδομένα συνήθως χαρακτηρίζονται από μια χρονική διάσταση και παρουσιάζονται σε διαφορετικά επίπεδα λεπτομέρειας με τη χρήση των *εννοιολογικών ιεραρχιών* (concept hierarchies). Μια τέτοια ιεραρχία θα μπορούσε να είναι η `Μέρα`<`Μήνας`<`Τρίμηνο`<`Έτος`. Χιλιάδες ή και εκατομμύρια τέτοιες εγγραφές παράγονται ανά δευτερόλεπτο και τα σύγχρονα συστήματα πρέπει να είναι σε θέση τόσο να τα συμπεριλαμβάνουν στις αναλύσεις τους όσο και να τα επεξεργάζονται αποδοτικά. Είναι επομένως ξεκάθαρο πως η αποδοτική διαχείριση αυτού του τεράστιου όγκου πληροφορίας είναι εξέχουσας σημασίας για την πλήρη εκμετάλλευση της υπάρχουσας πληροφορίας και για τη λήψη αποφάσεων που στηρίζονται σε ακριβή, πλήρη και όσο το δυνατόν πιο πρόσφατα στοιχεία.

Επιπλέον, τα ίδια τα πληροφοριακά περιβάλλοντα είναι κατανεμημένα. Επιχειρήσεις αποτελούνται από πολλές εταιρίες ανά τον κόσμο, οι οποίες, παρόλο που λειτουργούν αυτόνομα, πρέπει να παρέχουν στην έδρα της επιχείρησης συνοπτική πληροφορία για τη λήψη αποφάσεων. Είναι επίσης γεγονός ότι η τεχνολογία της πληροφορίας κινείται προς περιβάλλοντα όπου οι πόροι παρέχονται ως υπηρεσία μέσω διαδικτύου και οι επιχειρηματικές εφαρμογές είναι προσβάσιμες μέσω φυλλομετρητή ιστού (web browser) [Eco10]. Το Υπολογιστικό Νέφος (*Cloud Computing*) είναι το πιο πρόσφατο τέτοιο παράδειγμα, που έχει προσελκύσει το ενδιαφέρον τόσο του ερευνητικού όσο και του επιχειρηματικού κόσμου. Μεγάλες εταιρίες πληροφορικής συμμετέχουν σε

αυτό παρέχοντας υποδομή (Microsoft [mic], Amazon [ec2], Google [gap], κλπ.), εταιρίες που κατασκευάζουν λύσεις για αναλυτική επεξεργασία το υποστηρίζουν (Vertica [ver], Terradata [ter], GoodData [gooa], κλπ.) και πλήθος επιχειρήσεων το χρησιμοποιεί. Δεν είναι τυχαίο ότι οι μεγαλύτεροι παροχείς περιεχομένου και οι σημαντικότερες σελίδες κοινωνικής δικτύωσης έχουν ήδη υιοθετήσει πρακτικές του υπολογιστικού νέφους: Το Twitter [twia], το Digg [dig], το Reddit [red] και πολλά άλλα χρησιμοποιούν την κατανεμημένη βάση Apache Cassandra [cas] για να αποθηκεύουν τον τεράστιο όγκο δεδομένων τους, η υποδομή μηνυμάτων του Facebook [fac] στηρίζεται στην HBase [hba], ενώ το LinkedIn [lin] έχει υλοποιήσει τη δική του βάση, τη Voldemort [vol], για να χειρίζεται τα δεδομένα αλλά και τον αυξανόμενο ρυθμό προσπέλασής τους. Εκτιμάται ότι τουλάχιστον 15% του ψηφιακού κόσμου θα είναι αποθηκευμένο στο νέφος μέχρι το 2020 και τουλάχιστον το ένα τρίτο όλων των δεδομένων θα περνούν από αυτό κάποια στιγμή της ζωής τους [IDC10]. Σε τέτοια περιβάλλοντα το λογισμικό αλλά και τα δεδομένα αποθηκεύονται σε εξυπηρετητές συχνά γεωγραφικά απομακρυσμένους, συνεπώς η διαχείρισή τους απαιτεί εξελιγμένες, κατανεμημένες τεχνικές.

Μέχρι πρόσφατα η διαχείριση δεδομένων μεταφραζόταν κυρίως στην πρόσβαση σε κεντρικές βάσεις, στενά συνδεδεμένες με την εκάστοτε εφαρμογή. Όμως αυτές οι συμβατικές τακτικές δε μπορούν να ακολουθήσουν το ρυθμό των αυξανόμενων αναγκών των σύγχρονων εφαρμογών δεδομένων. Ιδιαίτερα στον τομέα της αναλυτικής επεξεργασίας, τα υφιστάμενα συστήματα αποτυγχάνουν να συνδυάσουν την ισχυρή επεξεργασία δεδομένων με τον υψηλό ρυθμό ενημερώσεων: Οι παραδοσιακές αποθήκες δεδομένων (π.χ., [LPZ03, SDRK02, WLFY02]), ενώ είναι αποδοτικές για πολύπλοκα ερωτήματα πάνω σε ιστορικά δεδομένα μεγάλου όγκου, αποτελούν μια αυστηρά κεντρική και ασύγχρονη προσέγγιση. Οι όψεις (views) συνήθως υπολογίζονται ημερησίως ή εβδομαδιαίως, αφού τα λειτουργικά δεδομένα έχουν μεταφερθεί από τις διάφορες βάσεις. Κατανεμημένες παραλλαγές όπως οι [AAC$^+$08, ABJ$^+$03]) πρακτικά απλώς διασυνδέουν συμβατικές αποθήκες δεδομένων, παραμένοντας κεντρικές στη βασική τους λειτουργικότητα.

Από την άλλη πλευρά, υπάρχει αξιόλογη δουλειά σχετικά με το διαμοιρασμό σχεσιακών δεδομένων τόσο σε αδόμητα (τύπου Gnutella) όσο και σε δομημένα (π.χ., *Κατανεμημένοι Πίνακες Κατακερματισμού* ή *DHTs* εν συντομία) δίκτυα ομότιμων κόμβων (Peer-to-Peer ή P2P), που συνδυάζουν τα πλεονεκτήματα μιας κατανεμημένης λύσης με την απόδοση ενός συστήματος βάσης δεδομένων. Στα Peer Database Management Systems [KTSR09, HHL$^+$03, NOTZ03] οι κόμβοι διατηρούν βάσεις με διαφορετικά σχήματα και επικοινωνούν μεταξύ τους με κατανεμημένο και ανεκτικό σε σφάλματα τρόπο, χρησιμοποιώντας αναδιατύπωση (reformulation) για τη μετάφραση ενός ερωτήματος από ένα σχήμα σε ένα άλλο. Ωστόσο, δε δίνεται κάποια ιδιαίτερη προσοχή σε πολυδιάστατα δεδομένα με ιεραρχίες ούτε και σε δεδομένα χρονικών σειρών.

Μια νέα κλάση μηχανών ανάλυσης [ABPA$^+$09, TSJ$^+$09] που έχει κάνει πρόσφατα την εμφάνισή της βασίζεται σε αρχιτεκτονικές χωρίς κοινόχρηστους πόρους (shared-nothing) χρησιμοποιώντας υπολογιστές του εμπορίου και καλύπτει την νέα απαίτηση για κλιμακωσιμότητα,

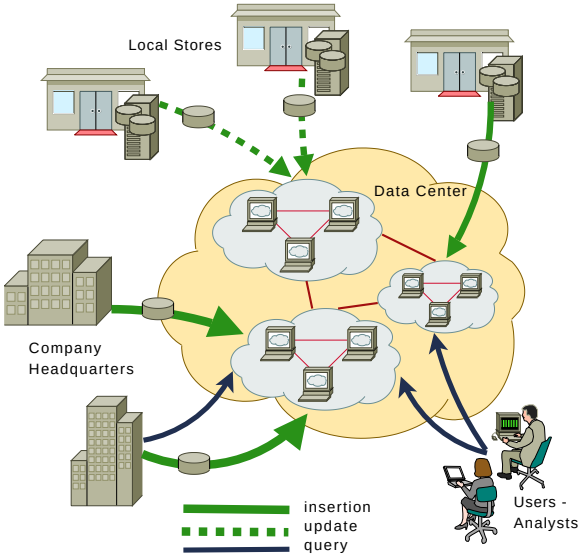
ευρωστία και διαθεσιμότητα σε χαμηλό κόστος. Ωστόσο, ακόμα και οι νέες πλατφόρμες έχουν κάποιους περιορισμούς: Βασιζόμενες στο προγραμματιστικό μοντέλο MapReduce, στοχεύουν περισσότερο σε μαζικές αναλυτικές εργασίες παρά σε επεξεργασία ανά πλειάδα σε πραγματικό χρόνο. Αυτό το μειονέκτημα αναγνωρίστηκε πρόσφατα από την Google και προτάθηκαν εναλλακτικοί τρόποι για αυξητική επεξεργασία και διαδραστικούς χρόνους απόκρισης [DP10, MGL+10].

Για να ολοκληρώσουμε με τις σχετικές λύσεις, οι παράλληλες βάσεις [ora, ter] προσφέρουν μεγάλη αποδοτικότητα εις βάρος της ελαστικότητας και της ευρωστίας σε σφάλματα [PPR+09]. Παρόλο που η κλιμακωσιμότητα θεωρητικά παρέχεται,τα υπάρχοντα συστήματα λειτουργούν σε συστοιχίες υπολογιστών της τάξης των εκατοντάδων. Παραβλέποντας το κόστος της αγοράς και της υποστήριξης τέτοιων συστημάτων, η δυσκολία εγκατάστασης και σωστής διαμόρφωσής τους αποτρέπουν την αυτόματη και διαφανή στο χρήστη διεύρυνσή τους για το χειρισμό της αυξανόμενης ζήτησης.

Οι απαιτήσεις μας συνεπάγονται τη δημιουργία ενός συστήματος για την αποθήκευση, ενημέρωση και προσπέλαση των δεδομένων σε πραγματικό χρόνο και την ταυτόχρονη επεξεργασία υψηλού ρυθμού ερωτημάτων χωρίς επιδείνωση του χρόνου απόκρισης. Ως κινητήριο σενάριο, ας εξετάσουμε μια επιχειρηματική εγκατάσταση που διατηρεί τα αρχεία των εργασιών της. Τα αρχεία αυτά θα μπορούσαν κάλλιστα να είναι καταγραφές ασφαλείας, δικτύου ή συμβάντων του συστήματος. Η αναζήτηση και η ανάλυση αυτών των δεδομένων αποτελεί ουσιαστικό μέρος της διαχείρισης, της ασφάλειας και του ελέγχου χρήσης της τεχνολογικής υποδομής της εταιρείας. Αντί της δημιουργίας μιας κεντρικής αποθήκης δεδομένων επί τόπου με μεγάλο κόστος συντήρησης, επιλέγεται η κατανομή των δεδομένων σε εγκαταστάσεις υπολογιστών με εύκολη πρόσβαση. Με τον τρόπο αυτό, μειώνεται σημαντικά το κόστος συντήρησης και υλικού ενώ παρέχει ένα κλιμακούμενο σύστημα λήψης αποφάσεων σε πραγματικό χρόνο. Το Σχήμα 1 απεικονίζει αυτό το σενάριο όπου πολλαπλές εγκαταστάσεις μιας επιχείρησης εισάγουν, ενημερώνουν και αναζητούν δεδομένα σε μία τέτοια κατανεμημένη αποθήκη.

0.1.2 Συμβολή

Στην παρούσα εργασία, ασχολούμαστε με την αποθήκευση, δεικτοδότηση και αναζήτηση πολυδιάστατων δεδομένων που χρησιμοποιούνται για την αναλυτική επεξεργασία σε κατανεμημένα συστήματα μεγάλης κλίμακας. Διερευνούμε τον τρόπο δημιουργίας ενός συστήματος που θα παρέχει πρόσβαση πραγματικού χρόνου σε δεδομένα, επιτρέποντας την ταυτόχρονη επεξεργασία υψηλού ρυθμού ερωτημάτων χωρίς σημαντική υποβάθμιση του χρόνου απόκρισης. Αυτό επιτυγχάνεται εφαρμόζοντας τεχνικές από το χώρο της κατανεμημένης διαχείρισης δεδομένων και των αποθηκών δεδομένων. Ο στόχος είναι να διατηρηθούν τα πλεονεκτήματα και από τις



Σχήμα 1: *Κινητήριο σενάριο για την κατανομή μιας αποθήκης δεδομένων*

δύο περιοχές: Η ισχυρή δεικτοδότηση και αναλυτική επεξεργασία για τεράστιες ποσότητες δεδομένων τόσο ιστορικών όσο και πραγματικού χρόνου και μια αρχιτεκτονική shared-nothing που να διασφαλίζει την επεκτασιμότητα και τη διαθεσιμότητα σε χαμηλό κόστος. Γεωγραφικά κατανεμημένοι χρήστες, χωρίς τη χρήση κάποιου εξειδικευμένου εργαλείου, μπορούν να μοιράζονται πληροφορίες που φθάνουν από διάφορες τοποθεσίες με υψηλό ρυθμό και να τις αναζητούν σε διάφορα επίπεδα λεπτομέρειας.

Η ερευνητική διαδικασία προς το στόχο αυτό ξεκινά με την προσπάθεια, την πρώτη στη βιβλιογραφία, για την υποστήριξη ιεραρχιών σε δίκτυα DHT, ώστε να αποθηκεύεται ιστορική πληροφορία σε διάφορα επίπεδα λεπτομέρειας. Το σύστημα που προέκυψε, το *HiPPIS*, απλοποιεί σημαντικά τις διαδικασίες εισαγωγής και ενημέρωσης λόγω της έλλειψης προεπεξεργασίας των δεδομένων. Παρολαυτά αυξάνει την μετέπειτα επεξεργασία από την πλευρά του πελάτη.

Στην προσπάθειά μας να προνοήσουμε για ερωτήματα group-by αλλά και να διατηρήσουμε υψηλή την απόδοση κάτω από διάφορες συνθήκες διαθεσιμότητας των δεδομένων και πόλωσης του φόρτου, κατανέμουμε μια γνωστή, ιδιαίτερα αποτελεσματική κεντρική δομή, το Dwarf [SDRK02], σε ένα δίκτυο συνδεδεμένων κόμβων, μειώνοντας το χρόνο της δημιουργίας του κύβου αλλά και το χρόνο αναζήτησης επιβάλλοντας παραλληλοποίηση.

Για να βελτιωθεί η διαχείριση *χρονικών σειρών* δεδομένων (*time series*), δηλαδή δεδομένων που καθορίζονται από μια χρονική διάσταση, προσδιορίζονται οι ειδικές απαιτήσεις του συγκεκριμένου τύπου δεδομένων και εφαρμόζονται οι κατάλληλες τροποποιήσεις στα προτεινόμενα συστήματα.

Με την εξέταση και την αξιολόγηση των προτεινόμενων συστημάτων, αποκαλύπτουμε τα πλεονεκτήματα και τις αδυναμίες τους. Υπάρχει μια αντιστάθμιση μεταξύ της απλότητας των λειτουργιών από τη μία και της κατανάλωσης αποθηκευτικού χώρου καθώς και της αποτελεσματικότητας της αναζήτησης από την άλλη. Ως συμπέρασμα της έρευνάς μας συνδυάζουμε τα πλεονεκτήματα των παραπάνω συστημάτων, δημιουργώντας το σύστημα *HORAE*, μια υβριδική λύση για την αποθήκευση και την επεξεργασία δεδομένων.

Η ερευνητική δουλειά της διατριβής αυτής χωρίζεται σε τρία μέρη:

Το Σύστημα HiPPIS

Το Σύστημα Ομότιμων Κόμβων για τη Δεικτοδότηση Ιεραρχιών (Hierarchical Peer-to-Peer Indexing System ή *HiPPIS*) [DATK08, DTK08, DTK11] είναι ένα κατανεμημένο σύστημα σχεδιασμένο να αποθηκεύει, να αναζητά και να ενημερώνει αποδοτικά πολυδιάστατα δεδομένα, οργανωμένα σε εννοιολογικές ιεραρχίες. Το *HiPPIS* υιοθετεί μια μέθοδο που προσαρμόζει αυτόματα το επίπεδο δεικτοδότησης βάσει του επιπέδου λεπτομέρειας των εισερχόμενων ερωτημάτων χωρίς να προϋποθέτει εκ των προτέρων γνώση του φορτίου. Αποδοτικές λειτουργίες roll-up και drill-down λαμβάνουν χώρα για τη μεγιστοποίηση της απόδοσης, ελαχιστοποιώντας το επικοινωνιακό κόστος Οι ενημερώσεις γίνονται σε πραγματικό χρόνο, με κόστος που εξαρτάται από το επίπεδο της συνέπειας (consistency) που απαιτείται. Η εκτεταμένη πειραματική αξιολόγηση δείχνει ότι, εκτός από τα πλεονεκτήματα που προσφέρει η κατανεμημένη αποθήκευση, η μέθοδός μας απαντά την πλειοψηφία των εισερχόμενων ερωτημάτων, τόσο σημειακών όσο και συγκεντρωτικών, χωρίς να πλημμυρίζει το δίκτυο και χωρίς να προκαλεί σημαντικές ανισορροπίες σε αποθηκευτικό χώρο ή φορτίο. Το σύστημά μας αποδεικνύεται ιδιαίτερα αποτελεσματικό σε περιπτώσεις ασύμμετρου φόρτου εργασίας, ακόμη κι όταν αυτός αλλάζει δυναμικά με το χρόνο. Ταυτόχρονα, διατηρεί την ιεραρχική φύση των δεδομένων. Σύμφωνα με όσα γνωρίζουμε, αυτή είναι η πρώτη προσπάθεια για την υποστήριξη των ιεραρχιών σε DHT δίκτυα.

Το Σύστημα Brown Dwarf

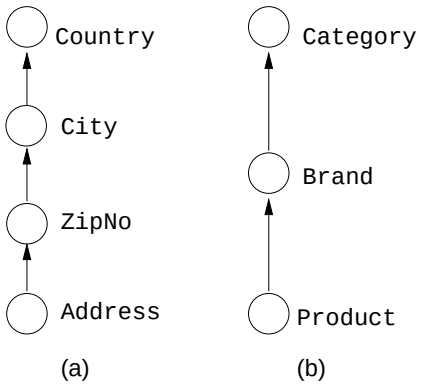
Το *Brown Dwarf* [DTK10a, DTK10c, DTK10b] είναι ένα σύστημα το οποίο διανέμει μια κεντρική δομή δεικτοδότησης, το Dwarf [SDRK02] στους κόμβους ενός αδόμητου δικτύου "εν κινήσει" (on-the-fly), επιταχύνοντας τόσο τη δημιουργία του όσο και την επίλυσης των ερωτημάτων χάρη στην παραλληλοποίηση που επιβάλλει. Τα ερωτήματα ανάλυσης καθώς και οι ενημερώσεις πραγματοποιούνται online μέσω των συνεργαζόμενων κόμβων του αδόμητου δικτύου επικάλυψης, εξαλείφοντας τη δαπανηρή συμβατική διαδικασία που συνήθως γίνεται ασύγχρονα. Η ελαστικότητα και η διαθεσιμότητα του περιεχομένου είναι

απαραίτητα στοιχεία: Το σύστημα χρησιμοποιεί ένα μηχανισμό αντιγραφής που προσαρμόζεται σε ξαφνικές αλλαγές του φόρτου εργασίας αλλά και στην κινητικότητα των κόμβων, διευρύνοντας ή συρρικνώνοντας τις μονάδες της κατανεμημένης δομής. Τα χαρακτηριστικά αυτά μαζί με την αποτελεσματικότητα και το χαμηλό κόστος στο απαιτούμενο υλικό και λογισμικό καθιστά το *Brown Dwarf* ιδανική υποψήφια εφαρμογή για ενσωμάτωση σε περιβάλλον υπολογιστικού νέφους [AFG+09]. Η πειραματική αξιολόγηση στους κόμβους πραγματικού δικτύου δείχνει ότι επιταχύνει τη δημιουργία του κύβου μέχρι και 5 φορές και την επίλυση επερωτήσεων αρκετές δεκάδες φορές σε σύγκριση με την κεντρική λύση, αξιοποιώντας τις δυνατότητες των διαθέσιμων κόμβων του δικτύου που εργάζονται παράλληλα. Επίσης, καταφέρνει να προσαρμόζεται γρήγορα, ακόμη και μετά από ξαφνικές αλλαγές στο φορτίο των ερωτημάτων και παραμένει ανεπηρέαστο από συχνές αστοχίες κόμβων. Τα πλεονεκτήματα αυτά είναι ακόμη πιο εμφανή για πυκνούς κύβους και πολωμένα φορτία εργασίας.

Το Σύστημα HORAE

Το *HORAE* είναι ένα υβριδικό σύστημα που εστιάζει στην διαχείριση χρονικών σειρών με πλήρως κατανεμημένο τρόπο. Τα δεδομένα χρονικών σειρών είναι μια σημαντική κατηγορία δεδομένων που περιέχουν τη διάσταση του χρόνου, όπως την ημερομηνία σε μια ροή δεδομένων για πωλήσεις ή το χρόνο μιας αγοράς με πιστωτική κάρτα. Αφού εξετάσουμε τη συμπεριφορά του *HiPPIS* καθώς και του *Brown Dwarf* όσον αφορά τα δεδομένα χρονικών σειρών σχεδιάζουμε ένα ολοκληρωμένο σύστημα που χρησιμοποιεί μια υβριδική λύση για την αποθήκευση και επεξεργασία τέτοιου είδους δεδομένων: Τα πιο πρόσφατα δεδομένα, τα οποία ενημερώνονται γρήγορα και αναζητούνται σε λεπτομερέστερο επίπεδο αποθηκεύονται σε ένα DHT σύστημα όμοιο με το *HiPPIS*, που επιτρέπει γρήγορη εισαγωγή και πολυδιάστατη δεικτοδότηση. Τον κύριο όγκο των δεδομένων διαχειρίζονται κύβοι όμοιοι με το *Brown Dwarf*, που υλοποιούνται και αντιγράφονται ανάλογα με τη ζήτηση.

Τα δύο αυτά συστατικά του συστήματος ενσωματώνονται εύκολα, συνδυάζοντας τα πλεονεκτήματα της ισχυρής κεντρικής επεξεργασίας με την κλιμακωσιμότητα και την ελαστικότητα. Η υλοποίηση του συστήματος και η εφαρμογή του σε πραγματική πλατφόρμα δοκιμών αποδεικνύει ότι το HORAE είναι σε θέση να χειριστεί αποτελεσματικά μεγάλους ρυθμούς ενημερώσεων και ερωτημάτων, είναι ανεκτικό σε υψηλά ποσοστά αποτυχίας κόμβων και συστέλλει ή διαστέλλει τους πόρους του ανάλογα με τη ζήτηση. Η άμεση σύγκριση με μια σύγχρονη λύση αποθήκης δεδομένων αποδεικνύει τα πλεονεκτήματα του HORAE σε απόδοση και ελαστικότητα υπό μεταβλητό φόρτο εργασίας: Το σύστημά μας επιταχύνει την επίλυση ερωτημάτων κατά τάξεις μεγέθους, προσαρμόζεται γρήγορα, ακόμη και μετά από ξαφνικές εκρήξεις στο εισερχόμενο φορτίο και παραμένει ανεπηρέαστο από συχνές αστοχίες σημαντικού ποσοστού των κόμβων.



Σχήμα 2: *Εννοιολογικές ιεραρχίες για τις διαστάσεις (α)* `location` *(β)* `product`

0.2 Το Σύστημα Ομότιμων Κόμβων για τη Δεικτοδότηση Ιεραρχιών

0.2.1 Επισκόπηση

Ως σενάριο χρήσης, ας θεωρήσουμε μια πολυεθνική εταιρία πωλήσεων που παράγει μεγάλες ποσότητες δεδομένων. Οι χρήστες/αναλυτές επιθυμούν να θέτουν ερωτήματα που αφορούν τα δεδομένα αυτά σε όλες τις διαστάσεις τους, να εφαρμόζουν απλές αλλά σημαντικές λειτουργίες εξόρυξης (mining) όπως roll-up και drill-down στις ιεραρχίες και να υπολογίζουν συναθροιστικές όψεις. Έστω ότι η βάση δεδομένων της εταιρίας περιέχει δεδομένα που οργανώνονται στις διαστάσεις `location` και `product` (βλ. Σχήμα 2).

Χρησιμοποιώντας ένα απλό DHT θα έπρεπε για κάθε διάσταση να επιλεχθεί ένα επίπεδο της ιεραρχίας ώστε να εφαρμοστεί η συνάρτηση κατακερματισμού στην τιμή του επιπέδου αυτού για όλες τις πλειάδες προς εισαγωγή στο σύστημα. Υποθέτοντας ότι επιλέγονται τα επίπεδα `city` και `category`, θα υπάρχει ένας κόμβος υπεύθυνος για την τιμή *Athens*, ένας για *Milan*, κλπ., όπως επίσης και κόμβοι υπεύθυνοι για *Electronics*, *Household*, κλπ. Μια τέτοια δομή είναι πολύ αποδοτική για την επίλυση των ερωτημάτων που αφορούν τα δεικτοδοτημένα επίπεδα (ακόμα και τότε χρειάζεται να βρεθεί η τομή των διαφόρων συνόλων πλειάδων), αλλά ερωτήματα που αναφέρονται σε διαφορετικά επίπεδα απαιτούν καθολική επεξεργασία.

Η πολλαπλή εισαγωγή κάθε πλειάδας εφαρμόζοντας τη συνάρτηση κατακερματισμού σε όλα τα ιεραρχικά επίπεδα όλων των διαστάσεων δεν είναι βιώσιμη λύση: Καθώς αυξάνονται οι διαστάσεις, αυξάνεται εκθετικά και ο επιπλέον χώρος που χρειάζεται για την αποθήκευση των δεδομένων. Επιπλέον η λύση αυτή αδυνατεί να συμπεριλάβει τις ιεραρχικές σχέσεις. Για παράδειγμα, δε μπορούν να απαντηθούν ερωτήματα όπως "*Σε ποια χώρα ανήκει η Πάτρα*" ή "*Πόσο είναι το συνολικό εισόδημα για πωλήσεις ηλεκτρονικών*".

Στόχος μας είναι να ερευνήσουμε το πρόβλημα της δεικτοδότησης ιεραρχικών δεδομένων σε DHT δίκτυα με τρόπο που να διατηρεί τη σημασιολογία των ιεραρχιών και να επιλύει αποδοτικά σημειακά αλλά και συναθροιστικά ερωτήματα. Για το σκοπό αυτό προτείνουμε το *Σύστημα Ομότιμων Κόμβων για τη Δεικτοδότηση Ιεραρχιών* (*Hierarchical Peer-to-Peer Indexing System* ή *HiPPIS*), ένα σύστημα που βασίζεται σε DHT για την αποθήκευση και δεικτοδότηση δεδομένων υπό τη μορφή πινάκων (όπως του Πίνακα 1) σε πολλαπλές τοποθεσίες σε ένα δίκτυο. Επιπλέον επιτρέπει την αποδοτική αναζήτηση σε πολλές διαστάσεις που χαρακτηρίζονται από ιεραρχίες. Έτσι το σύστημα επωφελείται από τα εγγενή χαρακτηριστικά των P2P συστημάτων όπως κλιμακωσιμότητα, ανοχή σε σφάλματα και διαθεσιμότητα. Οι κόμβοι του *HiPPIS* εποπτεύουν την λεπτομέρεια των ερωτημάτων που λαμβάνουν ώστε να προσαρμόσουν το επίπεδο δεικτοδότησης ανάλογα με τη ζήτηση. Ο μηχανισμός αυτός σε συνδυασμό με τους εφήμερους (soft-state) δείκτες που δημιουργούνται μετά από κάθε αστοχία κάποιου ερωτήματος ελαχιστοποιεί τον αριθμό των μηνυμάτων που πλημμυρίζουν το δίκτυο (floodings) και διατηρεί τις σημασιολογικές σχέσεις των ιεραρχιών.

Οι κόμβοι αρχικά δεικτοδοτούν σε ένα συγκεκριμένο συνδυασμό επιπέδων που ονομάζουμε *pivot*. Οι πλειάδες που εισάγονται αποθηκεύονται εσωτερικά σε μια δενδρική δομή που διατηρεί τις ιεραρχικές σχέσεις. Οι αστοχίες των ερωτημάτων συνοδεύονται από τη δημιουργία soft-state δεικτών ώστε μελλοντικά ερωτήματα να απαντώνται χωρίς πλημμύρα. Οι κόμβοι διατηρούν τοπικά στατιστικά που μπορούν να χρησιμοποιηθούν για να αποφασιστεί αν χρειάζεται επαναδεικτοδότηση σε κάποιον άλλο συνδυασμό επιπέδων, ανάλογα με την τάση των ερωτημάτων. Εκτός από σημειακά ερωτήματα, το *HiPPIS* μπορεί να απαντήσει και συναθροιστικά.

Η συνεισφορά της δουλειάς αυτής είναι η εξής:

- Αντιμετωπίζει το πρόβλημα της αποθήκευσης και αναζήτησης ιεραρχικών δεδομένων σε DHT συστήματα, τα οποία δε υποστηρίζουν ευθέως ερωτήματα σε πολλαπλές διαστάσεις που χαρακτηρίζονται από ιεραρχίες. Η μέθοδός μας, λαμβάνοντας υπόψιν τις προτιμήσεις των χρηστών επιτρέπει την αναδιοργάνωση των δεικτών προς όφελος των πιο δημοφιλών δεδομένων. Επίσης καταφέρνει να διατηρεί τη σημασιολογία των ιεραρχιών, η οποία καταστρέφεται με τη συνάρτηση κατακερματισμού.

- Επιτρέπει τις online ενημερώσεις, σε αντίθεση με τις παραδοσιακές τεχνικές των αποθηκών δεδομένων. Το επιπλέον επικοινωνιακό κόστος εξαρτάται από το επίπεδο συνέπειας που απαιτεί η εκάστοτε εφαρμογή

- Το σύστημα έχει υλοποιηθεί σε προσομοιωτή και έχει αποτιμηθεί με μια πληθώρα από δεδομένα και κάτω από διάφορες συνθήκες δικτύου και φόρτου ερωτήσεων. Το *HiPPIS* επιτυγχάνει υψηλό ποσοστό από ακριβείς απαντήσεις ακόμα και όταν τα φορτία ερωτημάτων

Πίνακας 1: *Ο πίνακας δεδομένων για το παράδειγμά μας*

	Location			*Product*		Fact
TupleID	Country	City	Zip	Category	Brand	Sales
ID2	Greece	Athens	16674	Electronics	Apple	11,500
ID5	Greece	Athens	15341	Electronics	Sony	1,900
ID51	Greece	Athens	15341	Electronics	Philips	22,900
ID31	Greece	Athens	16732	Household	AEG	2,450
ID55	Greece	Larissa	20100	Electronics	Sony	12,100
ID190	Greece	Patras	19712	Household	Unilever	1,990
ID324	Greece	Athens	17732	Electronics	Philips	2,450
ID501	Greece	Athens	17843	Electronics	Sony	12,000
ID712	Greece	Athens	17843	Electronics	Apple	32,000

αλλάζουν δυναμικά με το χρόνο και αποδεικνύεται ιδιαίτερα αποδοτικό με πολωμένες κατανομές δεδομένων και ερωτημάτων (που συναντώνται και πιο συχνά στην πλειονότητα των εφαρμογών). Επιπλέον, ακόμα και με υψηλό ρυθμό ενημερώσεων καταφέρνει να επιστρέφει ενημερωμένα αποτελέσματα.

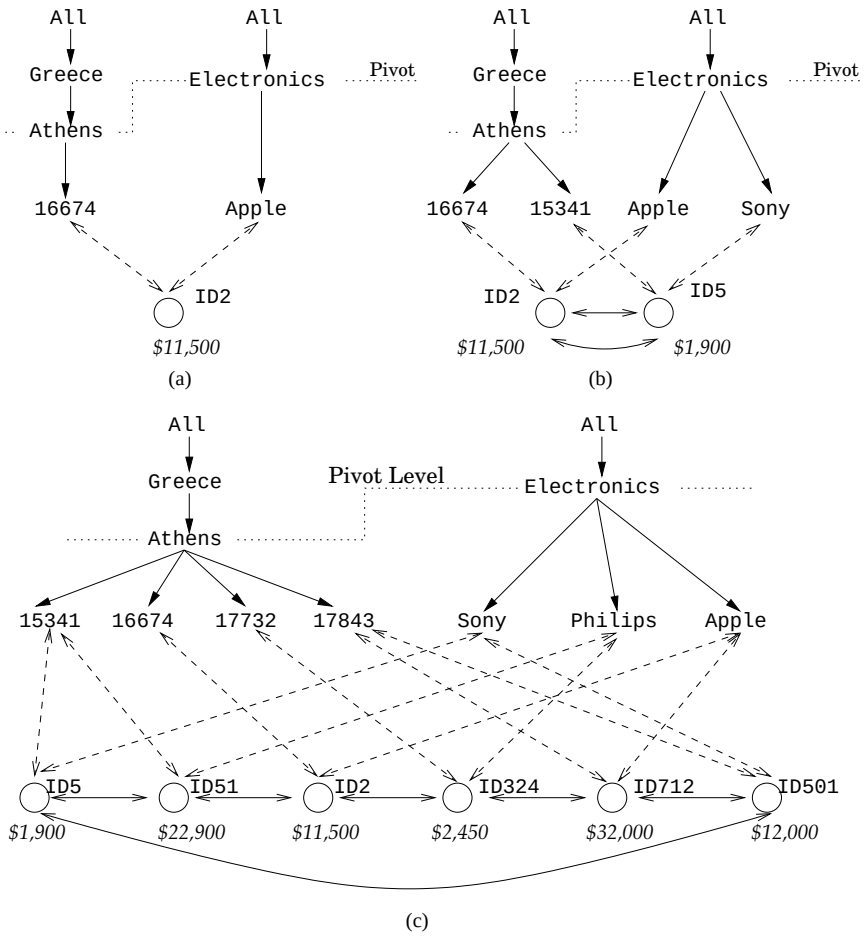
0.2.2 Σχεδίαση

Συμβολισμός

Τα δεδομένα που χειρίζεται το σύστημα είναι d διαστάσεων. Κάθε διάσταση i οργανώνεται κατά μήκος $L_i + 1$ ιεραρχικών επιπέδων: $H_{i0}, H_{i1}, \ldots, H_{iL_i}$, με την τιμή H_{i0} να είναι η ειδική τιμή *ALL (∗)*. Θεωρούμε ότι οι πλειάδες της βάσης μας είναι της μορφής:
$\langle tupleID, D_{11} \ldots D_{1L_1}, \ldots, D_{d1} \ldots D_{dL_d}, fact_1, \ldots, fact_k \rangle$, όπου $D_{ij}, 1 \leq i \leq d$ και $1 \leq j \leq L_i$ είναι η τιμή του επιπέδου j της διάστασης i και τα $fact_i, 0 \leq i \leq k$ είναι τα αριθμητικά ποσά ενδιαφέροντος. Στόχος είναι η αποδοτική δεικτοδότηση των πλειάδων ώστε να επιλύονται ερωτήματα της μορφής $q = \langle q_1, q_2, \ldots, q_d \rangle$, όπου κάθε στοιχείο q_i του ερωτήματος ανήκει σε κάποιο από τα επίπεδα της διάστασης i, συμπεριλαμβανομένης της τιμής $*$.

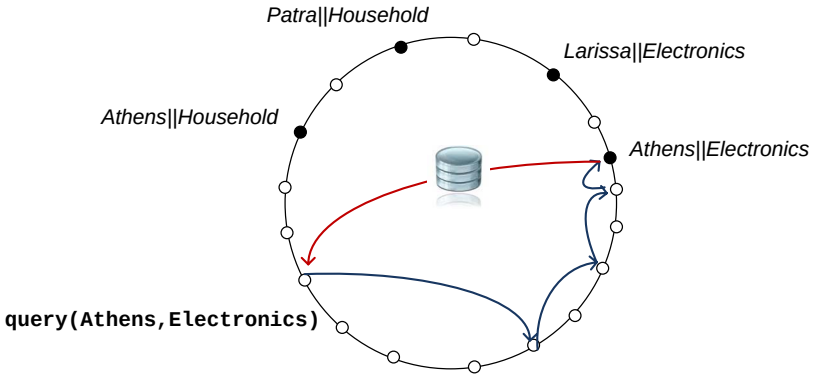
Εισαγωγή Δεδομένων

Η εισαγωγή μιας πλειάδας (ή ενός δείκτη στην πραγματική της θέση) γίνεται ως εξής: Αρχικά επιλέγεται καθολικά ένας συνδυασμός ιεραρχικών επιπέδων (ένα επίπεδο για κάθε διάσταση, συμπεριλαμβανομένης και της ειδικής τιμής $*$), που ονομάζεται *pivot* και συμβολίζεται P. Το ID της κάθε πλειάδας προς εισαγωγή είναι η τιμή που προκύπτει από την εφαρμογή της συνάρτησης κατακερματισμού στο συνδυασμό των τιμών που αντιστοιχούν στο P. Στη συνέχεια το DHT αναθέτει κάθε πλειάδα στον κόμβο με ID αριθμητικά πλησιέστερο στην τιμή αυτή. Για πλειάδες που εισάγονται αργότερα, οι κόμβοι πληροφορούνται για το P από κάποιον γειτονικό τους κόμβο.

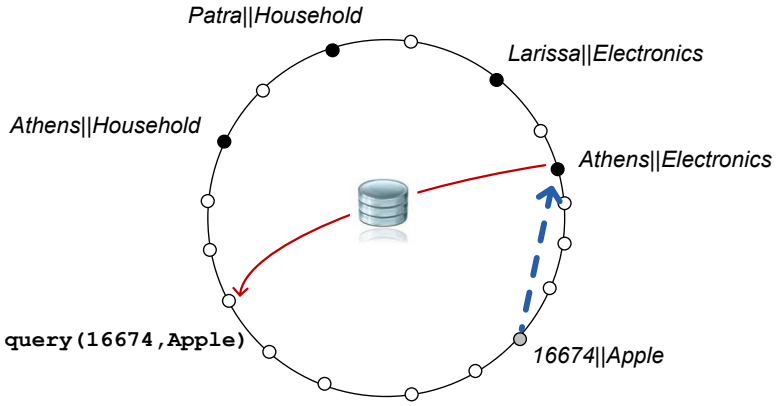


Σχήμα 3: *Η δενδρική δομή στον κόμβο υπεύθυνο για το* `Athens,Electronics` *μετά την εισαγωγή (a) της πρώτης, (b) της δεύτερης και (c) όλων των πλειάδων του Πίνακα 1*

Τα δεδομένα που εισάγονται στο σύστημα αποθηκεύονται σε μια δενδρική δομή, που διατηρεί την ιεραρχική φύση τους. Κάθε κόμβος αποθηκεύει πολλαπλά δένδρα, ένα για κάθε συνδυασμό από d τιμές για τον οποίο είναι υπεύθυνος. Έτσι, κάθε ξεχωριστή τιμή του P αντιστοιχεί σε ένα δένδρο που αποκαλύπτει μέρος της ιεραρχίας. Θεωρώντας τα δεδομένα του Πίνακα 1 και την ιεραρχία του Σχήματος 2 (χωρίς το τελευταίο επίπεδο σε κάθε διάσταση) με $\langle city, category \rangle$ ως το καθολικό pivot, η πρώτη πλειάδα προς εισαγωγή λαμβάνει το ID που προκύπτει από το hash της τιμής *Athens||Electronics* και δημιουργεί τη δομή του Σχήματος 3(a). Όσο δεδομένα με ίδια ID φτάνουν στον κόμβο, οι διαφορετικές τιμές σε επίπεδα χαμηλότερα του pivot δημιουργούν παρακλάδια, σχηματίζοντας δενδρικές δομές (Σχήματα 3(b) και (c)).



Σχήμα 4: *Αναζήτηση για ⟨Athens,Electronics⟩*



Σχήμα 5: *Αναζήτηση και δημιουργία δείκτη για το ⟨16674,Apple⟩*

Αναζήτηση και Δεικτοδότηση Δεδομένων

Τα ερωτήματα που αφορούν το P ορίζονται ως *ακριβή (exact match)* και μπορούν να απαντηθούν μέσα σε $O(\log N)$ βήματα. Ερωτήματα για οποιονδήποτε άλλο συνδυασμό επιπέδων μπορούν να απαντηθούν μόνο με πλημμύρα στο δίκτυο DHT. Για να αποσβέσουμε το κόστος αυτής της λειτουργίας, εισάγουμε τους *soft-state δείκτες*. Αυτοί οι δείκτες δημιουργούνται κατ' απαίτηση, μόλις απαντηθεί κάποιο ερώτημα που αφορά συνδυασμό επιπέδων διάφορο του pivot. Αφού οι απαντήσεις συλλεχθούν από τους κόμβους μετά από πλημμύρα, ο κόμβος που έθεσε το ερώτημα εφαρμόζει τη συνάρτηση κατακερματισμού στην τιμή του ερωτηθέντος συνδυασμού και στέλνει δείκτες προς τα δεδομένα που συνέλεξε στον κόμβο που είναι υπεύθυνος για κλειδί που προέκυψε.

Οι soft-state δείκτες δίνουν στους χρήστες την ψευδαίσθηση ότι οι τιμές που ρωτώνται ανακτώνται αμέσως, σαν να ήταν εξ' αρχής δεικτοδοτημένες. Στην πραγματικότητα, $O(\log N)$ βήματα χρειάζονται για την εύρεση των δεικτών που στη συνέχεια χρησιμοποιούνται για την ανάκτηση των πλειάδων, οι οποίες μετά από κατάλληλους υπολογισμούς θα επιστρέψουν το σωστό αποτέλεσμα. Ο αριθμός των δεικτών που ακολουθούνται εξαρτάται από το ερώτημα και από το

P: Αν το ερώτημα αφορά επίπεδα ίσα ή μικρότερα του pivot, τότε υπάρχει μόνο ένας δείκτης. Σε αντίθετη περίπτωση πρέπει να ακολουθηθούν πολλοί δείκτες (ο ακριβής αριθμός εξαρτάται από τα δεδομένα).

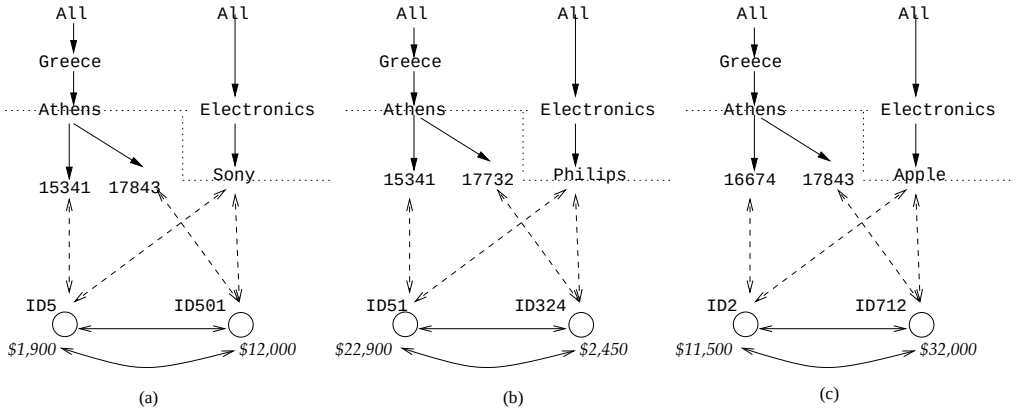
Οι δείκτες που δημιουργούνται είναι soft-state, ώστε να ελαχιστοποιείται η πρόσθετη πληροφορία. Αυτό σημαίνει ότι λήγουν μετά από ένα προκαθορισμένο διάστημα (Time-to-Live or *TTL*), εκτός και αν ανανεωθούν λόγω κάποιου ερωτήματος για τη συγκεκριμένη τιμή. Ο μηχανισμός αυτός διασφαλίζει ότι πιθανές αλλαγές στο σύστημα δε θα έχουν ως αποτέλεσμα την ύπαρξη άκυρων δεικτών, επηρεάζοντας με αυτόν τον τρόπο την απόδοση. Σε περιπτώσεις μεγάλου όγκου δεδομένων το μέγεθος των δεικτών ενδέχεται να αυξηθεί υπερβολικά. Για το λόγο αυτό θέτουμε ένα άνω όριο I_{max} στον αριθμό των δεικτών, πανω από το οποίο οι καινούριοι δείκτες που δημιουργούνται αντικαθιστούν τους παλαιότερους. Έτσι το σύστημα διατηρεί τους πιο χρήσιμους δείκτες, δηλαδή αυτούς που αναφέρονται σε δεδομένα που αναζητούνται πιο συχνά.

Σαν παράδειγμα, ας υποθέσουμε πως έχουμε την ίδια ιεραρχία με πριν, με P το ⟨`city`, `category`⟩. Ένα ερώτημα για το ⟨*Athens,Electronics*⟩ είναι exact match και μεταφράζεται σε απλή αναζήτηση σε DHT (Σχήμα 4). Όταν αναζητείται το ⟨*16674,Apple*⟩, ανακαλύπτουμε ότι δεν υπάρχει τέτοιο κλειδί στο DHT. Το ερώτημα πλημμυρίζει το δίκτυο και ο κόμβος *Athens||Electronics* απαντά με την αντίστοιχη πλειάδα. Ο κόμβος που έλαβε την απάντηση δημιουργεί στον κόμβο που είναι υπεύθυνος για το *16674||Apple* ένα δείκτη προς τον κόμβο *Athens||Electronics*. Έτσι, σε περίπτωση νέου ερωτήματος για την ίδια τιμή, θα αποφευχθεί η πλημμύρα και η απάντηση θα βρεθεί μέσα σε logN+C βήματα. Η ίδια διαδικασία απεικονίζεται στο Σχήμα 5 όπου οι μαύροι κόμβοι είναι αυτοί που αποθηκεύουν δεδομένα ενώ οι κόμβοι που περιέχουν δείκτες είναι γκρι. Οι ίδιοι οι δείκτες απεικονίζονται με διακεκομμένα βέλη.

Επαναδεικτοδότηση Δεδομένων

Σε μια αποθήκη δεδομένων, η κατανομή των δεδομένων και των ερωτημάτων μπορεί να αλλάζει με το χρόνο. Είναι λοιπόν πιθανόν η επιλογή του P, που γίνεται κατά την αρχική εισαγωγή των δεδομένων, να μην ευνοεί την απόδοση του συστήματος. Το *HiPPIS* προσαρμόζεται στην κατανομή του φορτίου των ερωτημάτων χωρίς να προϋποθέτει προηγούμενη γνώση του, υποστηρίζοντας δυναμικές αλλαγές του pivot βασιζόμενο αποκλειστικά σε τοπικά στατιστικά. Η επαναδεικτοδότηση βάσει διαφορετικού συνδυασμού επιπέδων στοχεύει στην αύξηση του ποσοστού των exact match ερωτημάτων, μειώνοντας τις πλημμύρες.

Αν ο αριθμός των ερωτημάτων ενός κόμβου που αφορούν συνδυασμούς επιπέδων διαφορετικούς του P ξεπερνά των αριθμό των ερωτημάτων που το αφορούν κατά ένα *όριο (threshold)*, τότε ο κόμβος εξετάζει την πιθανότητα επαναδεικτοδότησης. Κάθε κόμβος υπολογίζει την *δημοφιλία (popularity)* κάθε συνδυασμού επιπέδων ($\prod_{i=0}^{d} L_i$ στο σύνολο) μετρώντας τον αριθμό των ερωτημάτων κατά το πιο πρόσφατο χρονικό πλαίσιο (time-frame) W. Το W επιλέγεται κατάλληλα ώστε να αντιλαμβάνεται τις αλλαγές στην κατανομή των ερωτημάτων αλλά ταυτόχρονα να μένει ανεπηρέαστο από στιγμιαίες διακκυμάνσεις στο φορτίο.



Σχήμα 6: *Οι δενδρικές δομές μετά την επαναδεικτοδότηση*

Αν το ποσοστό των ερωτημάτων για τον πιο δημοφιλή συνδυασμό επιπέδων c_{max} διαφέρει περισσότερο από *threshold* του αντίστοιχου ποσοστού για το pivot, τότε ο κόμβος είναι θετικός στο ενδεχόμενο της επαναδεικτοδότησης. Σε δεύτερη φάση, πρέπει αυτή η τοπική διαίσθηση να επαληθευθεί ή να διαψευστεί με τη χρήση καθολικών στατιστικών. Ο κόμβος του οποίου τα τοπικά στατιστικά έδειξαν πιθανή αλλαγή του P στέλνει ένα μήνυμα *SendStats* σε όλους τους κόμβους. Μετά τη συλλογή των καθολικών στατιστικών ο κόμβος υπολογίζει εκ νέου το c_{max} και επαναλαμβάνει την προηγούμενη διαδικασία, εμπλουτισμένη με μια στρατηγική για τη βέλτιστη επιλογή του pivot.

Στην περίπτωση που επιλεχθεί νέο P, όλοι οι κόμβοι επαναδεικτοδοτούν τα δεδομένα τους. Ο αρχικός κόμβος πλημμυρίζει ένα μήνυμα *Reindex* για να υποχρεώσει όλους τους κόμβους να αλλάξουν το pivot τους. Κάθε κόμβος που λαμβάνει το μήνυμα αυτό διατρέχει τις πλειάδες του, βρίσκει όλες τις διαφορετικές τιμές για το νέο συνδυασμό που θα γίνει pivot και τις περνά από τη συνάρτηση κατακερματισμού, στέλνοντας τις πλειάδες με το ίδιο ID στον αντίστοιχο κόμβο. Όταν η διαδικασία ολοκληρωθεί, ο κόμβος σβήνει όλα τα παλιά δεδομένα και τους δείκτες του.

Σύμφωνα με το παράδειγμά μας, όταν ο κόμβος *Athens*||*Electronics* λαμβάνει μήνυμα *Reindex* για το ⟨`city`, `brand`⟩ βρίσκει ότι οι τιμές που αντιστοιχούν στο νέο συνδυασμό επιπέδων είναι οι *Athens*||*Sony*, *Athens*||*Philips* και *Athens*||*Apple*. Οι τιμές αυτές περνούν από τη συνάρτηση κατακερματισμού και πλέον οι αντίστοιχοι κόμβοι είναι υπεύθυνοι τα τις πλειάδες που τις περιέχουν (Σχήμα 6).

Κλείδωμα

Για να εξασφαλιστεί η ορθότητα των απαντήσεων κατά τη διάρκεια της επαναδεικτοδότησης και να αποφευχθούν ταυτόχρονες επαναδεικτοδοτήσεις από πολλούς κόμβους, εισάγουμε ένα μηχανισμό κλειδώματος. Αφού ο κόμβος αποφασίσει επαναδεικτοδότηση σύμφωνα με τα καθολικά στατιστικά, στέλνει ένα μήνυμα *Lock* σε όλους τους κόμβους του συστήματος πριν την εφαρμόσει. Όταν κάποιος κόμβος λάβει το μήνυμα *Lock* αλλάζει την κατάστασή του σε LOCKED και τη

διατηρεί για προκαθορισμένο χρονικό διάστημα, το οποίο ορίζεται έτσι ώστε να καλύπτει επαρκώς το χρόνο που χρειάζεται το σύστημα για να μεταβεί στο καινούριο pivot. Όσο ένας κόμβος είναι LOCKED συνεχίζει να απαντά ερωτήματα μέσω πλημμύρας. Έτσι το σύστημα μένει συνεχώς online.

Ενημερώσεις

Οι ενημερώσεις σε εφαρμογές αποθήκης δεδομένων μεταφράζονται στην εισαγωγή νέων πλειάδων στο σύστημα. Ενώ η εφαρμογή της συνάρτησης κατακερματισμού και η εισαγωγή ως προς το τρέχον επίπεδο είναι απλή, ενδέχεται να υπάρχουν δείκτες που πρέπει να ενημερωθούν καθώς η νέα πλειάδα πρέπει να συμπεριληφθεί στα αποτελέσματα διαφόρων ερωτημάτων. Για παράδειγμα, έστω ότι μια νέα πλειάδα αφορά στις πωλήσεις ηλεκτρονικών σε μια καινούρια πόλη της Ελλάδας. Ο υπάρχον δείκτης για ⟨*Greece, Electronics*⟩ θα πρέπει να συμπεριλάβει το ID του κόμβου που αποθήκευσε τη νέα πλειάδα. Σημειώνεται ότι ασυνέπειες προκύπτουν μόνο από πλειάδες που περιέχουν νέους συνδυασμούς pivot δημιουργώντας καινούριες δενδρικές δομές. Καθώς η δημιουργία ενός δείκτη μπορεί να συνοδεύεται από το σβήσιμο κάποιου ή κάποιων άλλων (λόγω περιορισμών μνήμης), ο κόμβος που εισάγει μια νέα πλειάδα δε μπορεί να ξέρει εκ των προτέρων την ύπαρξη ή όχι ενός σχετικού με αυτήν δείκτη. Αυτή η κατάσταση λύνεται με δύο τρόπους, ανάλογα με το επίπεδο συνέπειας που απαιτεί η εφαρμογή:

- *Ισχυρή συνέπεια:* Για εφαρμογές που βασίζονται σε συνεχή ανάλυση και άμεσο εντοπισμό αλλαγών, είναι ζωτικής σημασίας κάθε ερώτημα να λαμβάνει τα πιο πρόσφατα αποτελέσματα (π.χ. ανίχνευση εισβολών και επιθέσεων DoS). Για να επιτευχθεί ισχυρή συνέπεια, μετά την εισαγωγή της πλειάδας ο κόμβος εκτελεί $\prod_{i=0}^{d} L_i - 1$ αναζητήσεις για να εξακριβώσει την ύπαρξη πιθανών δεικτών για όλους τους συνδυασμούς επιπέδων και να τους ενημερώσει. Έτσι εξασφαλίζεται η συνέπεια με αντάλλαγμα περισσότερο επικοινωνιακό κόστος, που εξαρτάται από το ρυθμό των ενημερώσεων λ_{upd}.

- *Χαλαρή συνέπεια:* Όταν η εφαρμογή δεν έχει ανάγκη από τόσο "φρέσκα" δεδομένα, εφαρμόζεται μια μέθοδος χαλαρής συνέπειας. Οι κόμβοι επισυνάπτουν τις εισερχόμενες πλειάδες σε ένα καθολικά γνωστό κατάλογο. Οι κόμβοι που αποθηκεύουν δείκτες μπορούν ασύγχρονα να ανακτούν αυτόν τον κατάλογο και να ενημερώνουν τους δείκτες τους. Στο ενδιάμεσο, ενδέχεται οι απαντήσεις σε κάποια ερωτήματα να μην περιέχουν όλα τα πιο πρόσφατα δεδομένα. Η φρεσκάδα των απαντήσεων (freshness) εξαρτάται από το λ_{upd}, αλλά και από το ρυθμό λ_{index} με τον οποίο κάθε κόμβος λαμβάνει τον κεντρικό κατάλογο και ενημερώνει τους δείκτες του. Το επικοινωνιακό κόστος είναι μικρότερο από την προηγούμενη περίπτωση της ισχυρής συνέπειας, αφού $\lambda_{index} < \lambda_{upd}$. Έτσι, αυτή η προσέγγιση ενδείκνυται για περιπτώσεις που το εύρος ζώνης είναι περιορισμένο και δεν απαιτείται 100% συνέπεια.

0.2.3 Συζήτηση

Επιλογή του Threshold

Η τιμή του *threshold* παίζει σημαντικό ρόλο στην αποτελεσματικότητα του συστήματος και γι' αυτό πρέπει να ορίζεται προσεκτικά ώστε να αποφεύγονται άσκοπες επαναδεικτοδοτήσεις. Συχνές αναδιοργανώσεις των δεικτών πρέπει να αποθαρρύνονται, ωστόσο δεν πρέπει να αναστέλλονται ωφέλιμες επαναδεικτοδοτήσεις. Ο κόμβος που ξεκινά τη συλλογή καθολικών στατιστικών υπολογίζει την τιμή του *popularity* κάθε συνδυασμού επιπέδων, δηλαδή το ποσοστό των ερωτημάτων που αναφέρονται σε κάθε συνδυασμό, και τις ταξινομεί ($C : c_0 < c_1 < \ldots < c_{max}$). Η ολική κατανομή των ερωτημάτων πρέπει να λαμβάνεται υπόψιν καθώς το σύστημα ενδεχομένως να ωφελείται περισσότερο από την επιλογή ενός λιγότερο δημοφιλούς συνδυασμού επιπέδων από το c_{max}. Το συμπέρασμα αυτό προέρχεται από την εξής παρατήρηση:

- Παραμένοντας στο τρέχον P αποφεύγουμε τη δαπανηρή επαναδεικτοδότηση αλλά και την καταστροφή των soft-state δεικτών που έχουν δημιουργηθεί.
- Το $*$ εμπεριέχει όλα τα ιεραρχικά επίπεδα μιας διάστασης.

Η επιλογή του pivot διαμορφώνεται ως εξής: Οι συνδυασμοί των επιπέδων που βρίσκονται μέσα σε *threshold* απόσταση από το c_{max} θεωρούνται υποψήφια pivot:

$$\{\forall c_i \in C,\ 0 \leq i \leq max |\ \textit{popularity}_{c_{max}} - \textit{popularity}_{c_i} < \textit{threshold} \Rightarrow c_i \in C_{cand}\}$$

όπου το C_{cand} είναι το σύνολο των υποψήφιων pivot συνδυασμών. Η τιμή του ορίου είναι ανάλογη της Μέσης Διαφοράς (Mean Difference ή Δ) των τιμών του popularity και συγκεκριμένα *threshold*=$k \cdot \Delta$, $k \geq 1$. Η παράμετρος Δ, που ισούται με τη μέση απόλυτη διαφορά ανάμεσα σε δύο ανεξάρτητες τιμές, επιλέγεται ως μέτρο στατιστικής διασποράς:

$$\Delta = \frac{1}{max \cdot (max+1)} \sum_{i=0}^{max} \sum_{j=0}^{max} |c_i - c_j|$$

Ανάμεσα σε όλα τα υποψήφια pivot $c_i \in C_{cand}$ επιλέγουμε ένα με την εξής στρατηγική:

1. Αν το τρέχον επίπεδο $P \in C_{cand}$, το σύστημα δεν προβαίνει σε αλλαγή.
2. Αλλιώς, από όλα τα $c \in C_{cand}$ που περιέχουν $*$ σε μία ή περισσότερες διαστάσεις ξεχωρίζουμε μόνο τους συνδυασμούς που έχουν μέχρι $\lceil \frac{D}{2} \rceil$. Αυτό εξασφαλίζει ότι δε θα χρειαστεί υπερβολική τοπική επεξεργασία για τα εισερχόμενα ερωτήματα. Για καθέναν από τους συνδυασμούς που περιέχουν $*$, υπολογίζεται ξανά το *popularity* προσθέτοντας εκείνο άλλων επιπέδων που εμπεριέχονται σε αυτό. Για παράδειγμα αν θεωρήσουμε ότι τα $\langle$`Country`, `Brand`$\rangle$, $\langle$`City`, $*\rangle$ και $\langle *$, `Brand`$\rangle$ είναι τα υποψήφια pivot, με τιμές *popularity* ίσες

με 10%, 20% και 15% αντίστοιχα, τότε επειδή το $\langle *, \texttt{Brand} \rangle$ μπορεί να απαντήσει ερωτήματα που αφορούν το $\langle \texttt{Country}, \texttt{Brand} \rangle$, η δημοφιλία του ανεβαίνει στο 25% και υπερτερεί σε σχέση με το $\langle \texttt{City}, * \rangle$.

3. Αν δεν ισχύει τίποτα από τα παραπάνω, το σύστημα υιοθετεί το συνδυασμό επιπέδων με την υψηλότερη δημοφιλία.

0.2.4 Πειραματική αποτίμηση

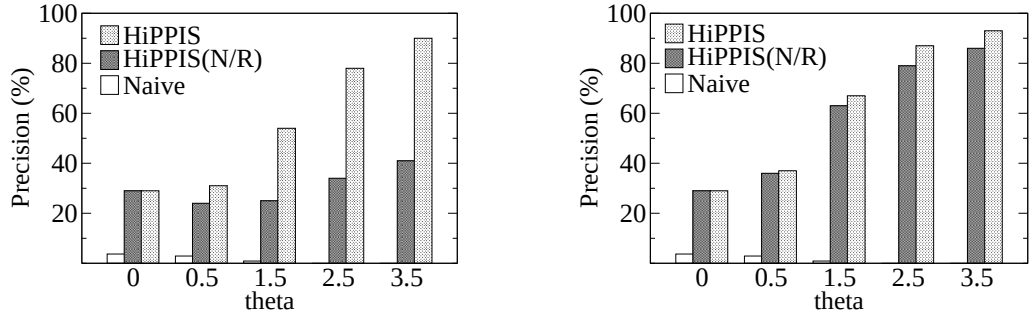
Η υλοποίηση του *HiPPIS* έχει γίνει με τη βοήθεια του προσομοιωτή FreePastry [fre] και παρουσιάζονται τα πιο αντιπροσωπευτικά πειράματα για την αποτίμηση του συστήματος. Στα πειράματα που παρουσιάζονται θεωρούμε δίκτυο $N = 256$ κόμβων, αν και έχουν διεξαχθεί πειράματα με δίκτυα που περιέχουν μέχρι 8K κόμβους.

Τα δεδομένα που χρησιμοποιούμε είναι συνθετικά και παράχθηκαν τόσο από το δικό μας γεννήτορα αλλά και από το γεννήτορα του APB-1 benchmark [apb]. Στην πρώτη περίπτωση, τα δεδομένα αποτελούνται εξ' ορισμού από 22k πλειάδες, οργανωμένες σε 3 διαστάσεις με 3 ιεραρχικά επίπεδα στην καθεμία. Το αρχικό pivot είναι το $\langle H_{12}, H_{22}, H_{32} \rangle$. Τα δεδομένα που παράχθηκαν από το APB-1 περιγράφονται στο αντίστοιχο πείραμα.

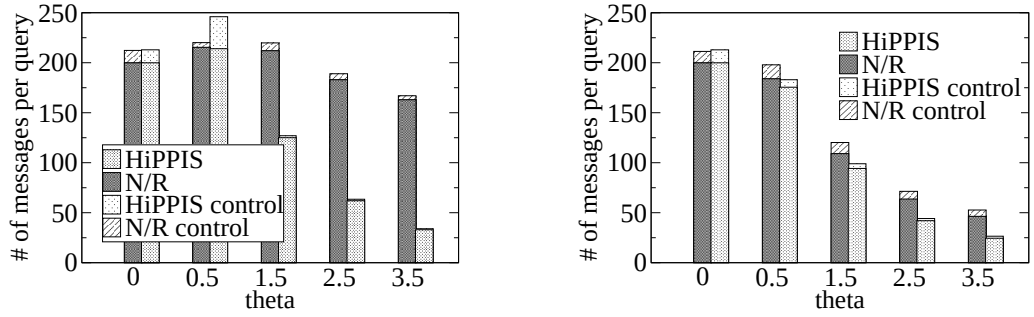
Για τα φορτία ερωτημάτων, ακολουθούμε μία προσέγγιση δύο σταδίων: Πρώτα ορίζουμε την πιθανότητα να ερωτηθεί κάθε συνδυασμός επιπέδων σύμφωνα με την κατανομή *levelDist* και μετά επιλέγουμε το κάθε ερώτημα από το συνδυασμό αυτό ακολουθώντας την κατανομή *valueDist*. Όλοι οι πιθανοί συνδυασμοί επιπέδων ταξινομούνται λεξικογραφικά, π.χ. $\langle H_{13}, H_{21}, H_{31} \rangle > \langle H_{11}, H_{23}, H_{33} \rangle$ και η κατανομή Zipf χρησιμοποιείται ως *levelDist* (ο αριθμός των ερωτημάτων για το συνδυασμό i είναι ανάλογος του $1/i^{\theta}$). Στα πειράματα που παρουσιάζονται μεταβάλλουμε τόσο την τιμή του θ όσο και την κατεύθυνση της πόλωσης. Για την κατανομή *valueDist* χρησιμοποιούμε τον κανόνα 80/20 εξ' ορισμού.

Τα φορτία που χρησιμοποιούνται αποτελούνται κατά κανόνα από 35k ερωτήματα με ρυθμό άφιξης λ_{query} ίσο με 10 ερωτήματα ανά μονάδα χρόνου. Για λόγους απλότητας θέτουμε τη μονάδα του χρόνου ίση με 1 sec, οπότε λ_{query} = 10 $\frac{queries}{sec}$. Θέτουμε την τιμή του W ίση με 50 sec ενώ οι soft-state δείκτες πρακτικά δε λήγουν ποτέ. Τέλος, η τιμή του I_{max} τίθεται μετά από πειράματα στην τιμή 2k. Αυτό πρακτικά σημαίνει ότι κάθε κόμβος αφιερώνει το πολύ 100KB μνήμης στους δείκτες soft-state.

Τα πειράματα έχουν σκοπό να αποδείξουν την αποδοτικότητα του συστήματος καθώς και την προσαρμοστικότητά του σε διάφορες συνθήκες. Σε αυτή την κατεύθυνση, μετράμε το ποσοστό των ερωτημάτων που επιλύονται χωρίς πλημμύρα (*precision*) και καταγράφουμε το μέσο αριθμό μηνυμάτων που ανταλλάσσονται ανά λειτουργία. Το *HiPPIS* συγκρίνεται με ένα απλό πρωτόκολλο (που ονομάζουμε *Naive*), όπου η τιμή του precision ισούται με το ποσοστό των ερωτημάτων που αφορούν το αρχικό pivot, καθώς και με την ειδική περίπτωση του *HiPPIS*, όπου



Σχήμα 7: *Η τιμή του precision για διαφορετικούς βαθμούς πόλωσης (με πιο δημοφιλείς συνδυασμούς επιπέδων τους $\langle H_{13}, H_{23}, H_{33}\rangle$ και $\langle H_{11}, H_{21}, H_{31}\rangle$ αντίστοιχα)*



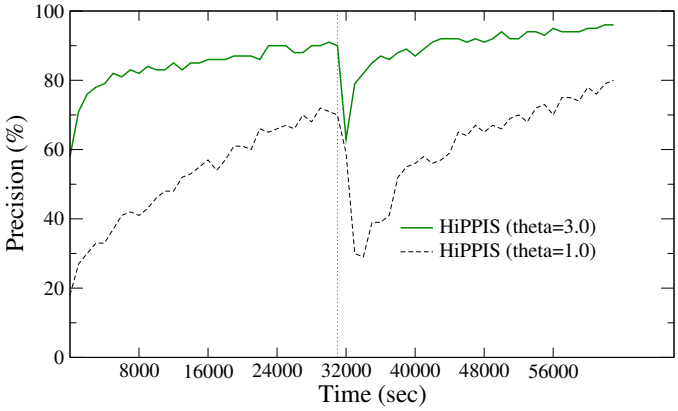
Σχήμα 8: *Ο μέσος αριθμός μηνυμάτων που απαιτούνται ανά ερώτημα για διαφορετικούς βαθμούς πόλωσης (με πιο δημοφιλείς συνδυασμούς επιπέδων τους $\langle H_{13}, H_{23}, H_{33}\rangle$ και $\langle H_{11}, H_{21}, H_{31}\rangle$ αντίστοιχα)*

χρησιμοποιούνται μόνο οι soft-state δείκτες χωρίς να επιτρέπεται επαναδεικτοδότηση (που ονομάζουμε *HiPPIS(N/R)* ή απλώς *N/R*).

Απόδοση με Διάφορες Κατανομές Φορτίου

Σε αυτήν την ομάδα πειραμάτων, μεταβάλλεται η παράμετρος θ της κατανομής levelDist καθώς και η κατεύθυνση της πόλωσης. Στο πρώτο γράφημα του Σχήματος 7, τα δεδομένα είναι πολωμένα με κατεύθυνση το $\langle H_{13}, H_{23}, H_{33}\rangle$. Όσο το θ αυξάνεται, τόσο πιο πολωμένο είναι το φορτίο και τόσο πιο βελτιωμένη η απόδοση του *HiPPIS*, καθώς η επαναδεικτοδότηση λαμβάνει χώρα νωρίτερα. Επιπλέον οι δείκτες συνεισφέρουν περισσότερο στο precision, αφού ο αριθμός των διαφορετικών ερωτημάτων για επίπεδα διάφορα του pivot μειώνεται. Για ομοιόμορφες κατανομές ο αριθμός των ερωτημάτων δεν επιτρέπει στη μέθοδό μας να επωφεληθεί από τη δεικτοδότηση.

Το επόμενο γράφημα παρουσιάζει αποτελέσματα για φορτία που ευνοούν το συνδυασμό $\langle H_{11}, H_{21}, H_{31}\rangle$. Παρατηρούμε παρόμοια τάση με προηγουμένως όσον αφορά την απόδοση όσο αυξάνεται η τιμή του θ. Παραταύτα, το *HiPPIS* είναι περισσότερο αποδοτικό από πριν, με τη διαφορά του από το *N/R* να αυξάνει με την αύξηση του θ. Αυτό οφείλεται στον περιορισμένο αριθμό των διαφορετικών τιμών του $\langle H_{11}, H_{21}, H_{31}\rangle$, που διευκολύνει τη διατήρηση των soft-state δεικτών, ωφελώντας έτσι περισσότερο το *N/R* έναντι του *HiPPIS*. Το τελευταίο σβήνει όλους τους



Σχήμα 9: *Η τιμή του precision ως προς το χρόνο για διαφορετικά φορτία όταν συμβαίνει μια ξαφνική αλλαγή στην πόλωση τη στιγμή* $t_c = 31000sec$

δείκτες που έχει δημιουργήσει κατά τη διαδικασία της επαναδεικτοδότησης. Ωστόσο το *HiPPIS* τελικά ξεπερνά τον ανταγωνιστή του στην σταθερή κατάσταση, αφού η απόδοσή του αυξάνεται με το χρόνο.

Το Σχήμα 8 απεικονίζει τον αριθμό των μηνυμάτων ανά ερώτημα ως ένδειξη κατανάλωσης του εύρους ζώνης. Τα μηνύματα που αφορούν την επίλυση των ερωτημάτων και τα μηνύματα ελέγχου παρουσιάζονται ξεχωριστά. Ποιοτικά, ο αριθμός μηνυμάτων είναι αντιστρόφως ανάλογος της τιμής του precision του συστήματος. Όπως παρατηρείται σε όλα τα πειράματα, ο επιπλέον φόρτος των μηνυμάτων ελέγχου είναι μικρός και αντισταθμίζεται από το κέρδος σε precision (λιγότερο από 8% του συνολικού αριθμού μηνυμάτων). Αυτό οφείλεται στο γεγονός ότι το *HiPPIS* εκτελεί τον ελάχιστο αριθμό επαναδεικτοδοτήσεων, που μεταφράζεται σε μία επαναδεικτοδότηση ανά κατεύθυνση πόλωσης. Παρατηρούμε ακόμα ότι ο επιπλέον φόρτος των μηνυμάτων ελέγχου μειώνεται με την αύξηση της πόλωσης (σχεδόν αμελητέος για $\theta > 1.5$). Αυτό εξηγείται από το γεγονός ότι το *HiPPIS* είναι πιο σίγουρο για το επίπεδο επαναδεικτοδότησης όσο το θ αυξάνει.

Απόδοση σε Δυναμικά περιβάλλοντα

Το πείραμα αυτό αποτιμά την απόδοση του *HiPPIS* σε δυναμικά περιβάλλοντα, όπου συμβαίνουν ξαφνικές αλλαγές στο φορτίο των ερωτημάτων. Χρησιμοποιούμε ένα φορτίο ερωτημάτων για το οποίο τη χρονική στιγμή $t_c = 31000sec$ η πόλωση αλλάζει κατεύθυνση και από το $\langle H_{13}, H_{23}, H_{33}\rangle$ στοχεύει στο $\langle H_{11}, H_{21}, H_{31}\rangle$. Τα αποτελέσματα για δύο επίπεδα πόλωσης παρουσιάζονται στο Σχήμα 9.

Σε όλες τις περιπτώσεις, το *HiPPIS* αυξάνει γρήγορα την τιμή του precision λόγω του συνδυασμού της αυτόματης επαναδεικτοδότησης και των soft-state δεικτών. Οι πλημμύρες αυξάνουν μετά το t_c, όμως το σύστημα σύντομα καταφέρνει να ανακάμψει καθώς λαμβάνει χώρα η επαναδεικτοδότηση και χτίζονται οι απαραίτητοι δείκτες. Ο ρυθμός με τον οποίο συμβαίνουν τα

Πίνακας 2: *Ποσοστό ασυνεπών απαντήσεων για διάφορα* λ_upd

λ_{upd} (ενημερώσεις/sec)	inconsistency (%) U_1	U_2
0.01	0.10	1.18
0.1	1.26	5.02
1.0	8.15	18.23
10.0	19.21	20.01

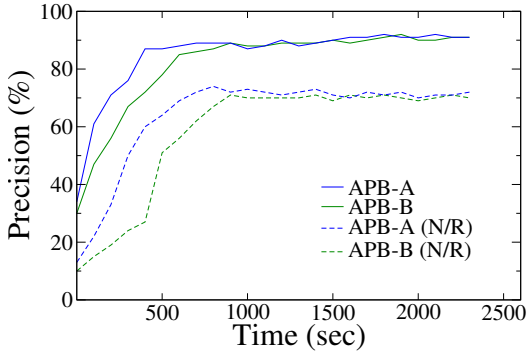
γεγονότα αυτά εξαρτάται από το βαθμό της πόλωσης. Για $\theta = 3.0$ η τιμή του precision αυξάνεται εντυπωσιακά και σχεδόν αμέσως μετά την αλλαγή στην κατεύθυνση της πόλωσης ανακτά την τιμή που είχε πριν την αλλαγή. Για λιγότερο πολωμένες κατανομές ο ρυθμός σύγκλισης μειώνεται, ωστόσο στη σταθερή κατάσταση η απόδοση παραμένει σταθερά πολύ υψηλή.

Ενημερώσεις

Στο σημείο αυτό στοχεύουμε στην αποτίμηση του μηχανισμού ενημερώσεων χαλαρής συνέπειας του *HiPPIS*. Συγκεκριμένα, θέτουμε στα δεδομένα του προηγούμενου πειράματος δύο φορτία ενημερώσεων, το U_1 και το U_2 με πόλωση $\theta = 2.0$. Το U_1 περιέχει αποκλειστικά σημειακά ερωτήματα, ενώ το 30% των ερωτημάτων του U_2 είναι συναθροιστικά. Κατά τη διάρκεια της προσομοίωσης θέτουμε ενημερώσεις στο σύστημα με ρυθμό λ_{upd} που κυμαίνεται από 0.01 μέχρι 10 $\frac{updates}{sec}$. Η περίοδος της διαδικασίας ενημέρωσης των δεικτών τίθεται ίση με 100 sec. Λαμβάνοντας υπόψιν ότι τα ερωτήματα εισέρχονται στο σύστημα με ρυθμό $\lambda_{query} = 10\frac{queries}{sec}$, τα ερωτήματα τίθενται 1,000 φορές πιο συχνά από τον έλεγχο των δεικτών. Στον Πίνακα 2 καταγράφεται το ποσοστό των ερωτημάτων των οποίων οι απαντήσεις είναι ατελείς, μέγεθος το οποίο ορίζουμε ως *inconsistency*.

Όσο πιο γρήγορος ο ρυθμός των ενημερώσεων, τόσο μεγαλύτερος ο αριθμός των ασυνεπών απαντήσεων. Ωστόσο, η ασυνέπεια τείνει να συγκλίνει όσο το λ_{upd} αυξάνεται. Ακόμα και όταν το λ_{upd} είναι ίσο με το το ρυθμό εισερχόμενων ερωτημάτων, δηλαδή κάθε ενημέρωση ακολουθείται από ένα ερώτημα, η ασυνέπεια παραμένει σε ανεκτά επίπεδα, λόγω του ότι μετά την απαραίτητη επαναδεικτοδότηση η πλειονότητα των ερωτημάτων επιλύεται χωρίς τη συνδρομή δεικτών. Η επίδραση της επαναδεικτοδότησης είναι εμφανώς μεγαλύτερη για το U_1. Επειδή δεν περιέχει συναθροιστικά ερωτήματα, το φορτίο είναι περισσότερο προσανατολισμένο στο νέο pivot, επομένως χρησιμοποιεί λιγότερο τους δείκτες.

Τέλος, αξίζει να σημειωθεί ότι όταν ακολουθείται η μέθοδος χαλαρής συνέπειας, το κόστος των ενημερώσεων είναι ανεξάρτητο του λ_{upd} και ισούται με N μηνύματα ανά περίοδο ενημέρωσης των δεικτών ($2.56\frac{msg}{sec}$ στην περίπτωση αυτή). Αντιθέτως, η μέθοδος ισχυρής συνέπειας



Σχήμα 10: *Η τιμή του precision του* HiPPIS *για το φορτίο APB*

παρέχει 100% ακρίβεια, αλλά απαιτεί $(\prod_{i=0}^{d} L_i - 1) \cdot \log(N)$ μηνύματα ανά ενημέρωση, με αποτέλεσμα ο μέσος ρυθμός να να κυμαίνεται από 2.16 σε 2160 $\frac{msg}{sec}$ για τις ρυθμίσεις της προσομοίωσής μας, ανάλογα με το λ_{upd}. Επομένως για υψηλές τιμές λ_{upd} ο μηχανισμός ισχυρής συνέπειας θα πρέπει να αποφεύγεται λόγω του σημαντικού επικοινωνιακού κόστους που επιφέρει.

Δεδομένα και Ερωτήματα του APB Benchmark

Τέλος, εξετάζουμε την απόδοση του *HiPPIS* χρησιμοποιώντας πιο ρεαλιστικά δεδομένα και ερωτήματα που δημιουργούνται από το APB-1 benchmark [apb]. Το APB-1 δημιουργεί μια βάση δεδομένων με πολλές διαστάσεις καθώς και ένα σύνολο λειτουργιών που αντανακλούν τις λειτουργίες μιας εφαρμογής OLAP. Θέτοντας την παράμετρο density σε 0.1 και 1, δημιουργούνται δύο ομάδες δεδομένων τεσσάρων διαστάσεων (APB-A και APB-B) με πληθαρίθμους 9000, 900, 9 και 24. Κάθε διάσταση εμπεριέχει μια ιεραρχία με 7, 4, 2 και 3 επίπεδα αντίστοιχα. Το APB-A περιέχει 1.2M και το APB-B 12M πλειάδες αντιστοίχως ενώ το φορτίο αποτελείται από 25k ερωτήματα. Τα αποτελέσματα απεικονίζονται στο Σχήμα 10.

Παρατηρείται ότι το *HiPPIS* εκδηλώνει υψηλή απόδοση, ξεπερνώντας το 90% σε precision στη σταθερή κατάσταση, περίπου 4000 ερωτήματα μετά την έναρξη του πειράματος για το APB-A. Το πείραμα αποδεικνύει ότι και για ρεαλιστικά σενάρια το *HiPPIS* προσαρμόζεται ταχύτατα και εξυπηρετεί την πλειονότητα των ερωτημάτων χωρίς πλημμύρα. Η απλή χρήση δεικτών μειώνει την τιμή του precision πάνω από 20%, ενώ υπάρχει και σχετική καθυστέρηση στην πρόσβαση στη σταθερή κατάσταση (χρειάζεται περίπου ο διπλάσιος χρόνος).

0.2.5 Ανακεφαλαίωση

Περιγράφηκε το *HiPPIS*, ένα κατανεμημένο σύστημα που αποθηκεύει και δεικτοδοτεί πολυδιάστατα ιεραρχικά δεδομένα σε δίκτυα DHT. Το *HiPPIS*, χωρίς καμία εκ των προτέρων γνώση του φορτίου ερωτημάτων και χωρίς καμία προεπεξεργασία, απαντά ερωτήματα σε διάφορες διαστάσεις και επίπεδα λεπτομέρειας. Το σύστημά μας προσαρμόζεται δυναμικά στο φορτίο, επαναδεικτοδοτώντας τα δεδομένα του σύμφωνα με τα εισερχόμενα ερωτήματα. Συνδυάζοντας

την προσαρμοστική δεικτοδότηση με soft-state δείκτες, το *HiPPIS* καταφέρνει να αποφύγει την πλημμύρα μηνυμάτων στις περισσότερες περιπτώσεις, ενώ επιτρέπει λειτουργίες πάνω σε μεγάλο όγκο δεδομένων σε πραγματικό χρόνο. Ανάλογα με τις ανάγκες της εφαρμογής, το *HiPPIS* μπορεί να εφαρμόσει μεθόδους ενημερώσεων με διαφορετικό επίπεδο συνέπειας, ώστε να πετύχει την επιθυμητή ακρίβεια χωρίς υπερβολικό κόστος στην επικοινωνία.

Οι προσομοιώσεις με τη χρήση διαφόρων κατανομών δεδομένων και φορτίων δείχνουν καλή απόδοση και χρήση εύρους ζώνης. Το *HiPPIS* είναι ιδιαίτερα αποτελεσματικό για πολωμένα φορτία, επιτυγχάνοντας υψηλά ποσοστά ακρίβειας και γρήγορη προσαρμογή σε δυναμικές αλλαγές στην κατεύθυνση της πόλωσης. Ακόμα και με λίγα επαναλαμβανόμενα ερωτήματα το *HiPPIS* καταφέρνει να απαντά την πλειονότητα χωρίς να πλημμυρίζει το δίκτυο, εντοπίζοντας τον πιο δημοφιλή συνδυασμό επιπέδων και επαναδεικτοδοτώντας ως προς αυτόν. Ταυτοχρόνως, το σύστημα αποφεύγει τις ανισορροπίες στο φόρτο και στον αποθηκευτικό χώρο.

0.3 Το Σύστημα Brown Dwarf

0.3.1 Επισκόπηση

Στόχος είναι η δημιουργία ενός αποτελεσματικού συστήματος αποθήκευσης δεδομένων, όπου γεωγραφικά κατανεμημένοι χρήστες, χωρίς τη χρήση κάποιου εξειδικευμένου εργαλείου, θα μπορούν να μοιράζονται και να αναζητούν πληροφορίες. Ως κίνητρο, ας εξετάσουμε μια επιχειρηματική εγκατάσταση που διατηρεί τα αρχεία για τις εργασίες της. Τα αρχεία αυτά θα μπορούσαν κάλλιστα να είναι αρχεία ασφάλειας δικτύου ή καταγραφής συμβάντων. Αντί της δημιουργίας μιας κεντρικής αποθήκης δεδομένων με μεγάλο κόστος αγοράς και συντήρησης, επιλέγεται η διανομή των δεδομένων και της επεξεργασίας τους σε πιθανώς πολλαπλές τοποθεσίες, όπου η πρόσβαση θα γίνεται εύκολα.

Για το σκοπό αυτό, προτείνουμε το σύστημα *Brown Dwarf**, που διανέμει online μια κεντρική δομή αποθήκευσης (*Dwarf* [SDRK02]) σε ένα δίκτυο κόμβων με τέτοιο τρόπο, ώστε όλα τα ερωτήματα που αρχικά επιλύονταν μέσω της κεντρικής δομής τώρα να κατανέμονται σε ένα αδόμητο δίκτυο P2P.

Το *Dwarf* είναι μια δομή για τον υπολογισμό, τη δεικτοδότηση και την αναζήτηση μεγάλου όγκου πολυδιάστατων δεδομένων. Ενώ προσφέρει πολλά πλεονεκτήματα όπως τη συμπίεση δεδομένων και την αποδοτική απάντηση συναθροιστικών ερωτημάτων, παρουσιάζει ορισμένους περιορισμούς που αποθαρρύνουν τη χρήση του σε περιπτώσεις όπως του σεναρίου. Εκτός από

*Το *Brown Dwarf* είναι ένα αντικείμενο με μέγεθος μεταξύ εκείνου ενός γιγάντιου πλανήτη και ενός μικρού αστεριού. Εικάζεται ότι ένα σημαντικό τμήμα της μάζας του σύμπαντος βρίσκεται σε αυτή τη μορφή.

Πίνακας 3: *Πίνακας δεδομένων με τρεις διαστάσεις και ένα ποσό ενδιαφέροντος (measure)*

DIM1	DIM2	DIM3	Measure
S1	C2	P2	$70
S1	C3	P1	$40
S2	C1	P1	$90
S2	C1	P2	$50

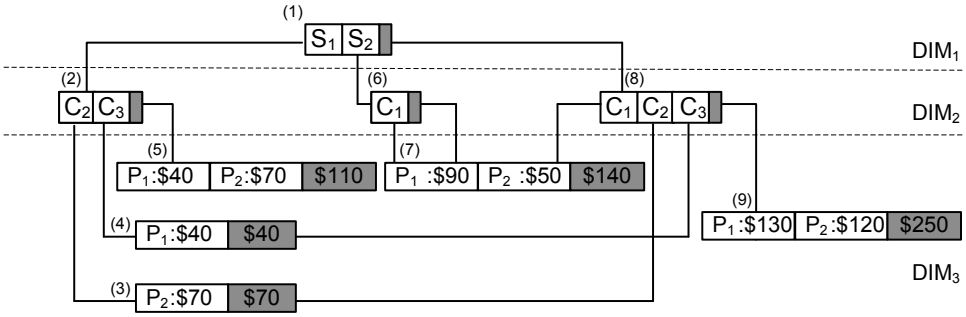
την έλλειψη ανοχής σε σφάλματα, μια δομή *Dwarf* μπορεί να καταλάβει χώρο κατά τάξεις μεγέθους μεγαλύτερο από ότι τα αρχικά δεδομένα [DBS08]. Το σύστημα *Brown Dwarf* χαλαρώνει την απαίτηση για αποθηκευτικό χώρο και διευκολύνει τον υπολογισμό πολύ μεγαλύτερων κύβων. Επιπλέον, επιτρέπει ενημερώσεις πραγματικού χρόνου που προέρχονται από οποιονδήποτε κόμβο του συστήματος. Τέλος, το προτεινόμενο σύστημα μπορεί να χειριστεί σημαντικά μεγαλύτερο ρυθμό ερωτημάτων και μένει ανεπηρέαστο από κόμβους που αποτυγχάνουν ή δε συνεργάζονται, καθώς προσφέρει πολλαπλά σημεία εισόδου και προσαρμοστική αντιγραφή των πιο βεβαρημένων κομματιών του κύβου.

Η συνεισφορά της δουλειάς αυτής είναι η εξής:

- Ένα ολοκληρωμένο σύστημα δεικτοδότησης, επεξεργασίας ερωτημάτων και ενημερώσεων για κύβους δεδομένων σε ένα κατανεμημένο περιβάλλον. Ο κύβος δημιουργείται με ένα μόνο πέρασμα από τα δεδομένα, ενώ οι ενημερώσεις εφαρμόζονται online. Η πρόσβαση στο κατανεμημένο αυτό σύστημα, στο οποίο συμμετέχουν υπολογιστές του εμπορίου, δεν απαιτεί τη χρήση κάποιου εξειδικευμένου εργαλείου.

- Ένας αποδοτικός και εύρωστος μηχανισμός αντιγραφής, που προσαρμόζεται τόσο στο φόρτο εργασίας όσο και στην κινητικότητα (εισαγωγές/εξαγωγές) των κόμβων, χρησιμοποιώντας μόνο τοπικές μετρήσεις και γνώσεις για το δίκτυο.

- Η ανάπτυξη του συστήματος και η λειτουργία του σε ένα πραγματικό δίκτυο φυσικών κόμβων. Η πειραματική αποτίμηση δείχνει ότι το *Brown Dwarf* είναι 5 φορές πιο γρήγορο στην δημιουργία του κύβου και μέχρι 60 φορές πιο γρήγορο στην επίλυση ερωτημάτων σε σύγκριση με την κεντρική εκδοχή. Επιπλέον κατανέμει δίκαια την κεντρική δομή αλλά και το φόρτο ερωτημάτων με μικρό κόστος, το οποίο μοιράζεται ανάμεσα σε πολλούς κόμβους, επιδεικνύει εντυπωσιακά άμεση προσαρμογή σε πολωμένες κατανομές φορτίου και είναι εξαιρετικά ανθεκτικό σε ένα σημαντικό ποσοστό συχνών αστοχιών κόμβων ακόμα και με μικρό βαθμό αντιγραφής.



Σχήμα 11: *Η κεντρική δομή Dwarf για τα δεδομένα του Πίνακα 3*

0.3.2 Η Δομή Dwarf

Η δομή ***Dwarf*** [SDRK02] αποθηκεύει, αναζητά και ενημερώνει υλοποιημένους κύβους δεδομένων. Το κύριο πλεονέκτημα του *Dwarf* είναι το γεγονός ότι εξαλείφει τόσο τις προθεματικές όσο και τις επιθεματικές επαναλήψεις στις διάφορες όψεις, μειώνοντας έτσι το μέγεθος του κύβου. Το Σχήμα 11 δείχνει τη δομή *Dwarf* για τα στοιχεία του Πίνακα 3: Η δομή χωρίζεται σε τόσα επίπεδα, όσες είναι και διαστάσεις των δεδομένων. Ο κόμβος-ρίζα (root) περιέχει όλες τις διαφορετικές τιμές της πρώτης διάστασης. Κάθε κελί (cell) δείχνει σε έναν κόμβο του επόμενου επιπέδου που περιέχει όλες τις διαφορετικές τιμές που σχετίζονται με την τιμή του cell. Τα γκρι κελιά αντιστοιχούν στην τιμή *"ALL"*, που χρησιμοποιείται για τη συνάθροιση σε κάθε διάσταση. Οποιοδήποτε σημειακό ή συναθροιστικό ερώτημα μπορεί να επιλυθεί διατρέχοντας τη δομή και ακολουθώντας τα attributes του ερωτήματος που οδηγούν τελικά σε έναν κόμβο-φύλλο (leaf) που περιέχει την απάντηση. Για παράδειγμα το ερώτημα $\langle S1, C3, P1\rangle$ θα επιστρέψει την τιμή \$40 ενώ το $\langle S2, ALL, ALL\rangle$ την τιμή \$140 ακολουθώντας τους κόμβους (1)→(6)→(7).

Το ***Brown Dwarf*** (ή *BD*) είναι ένα σύστημα που κατανέμει το *Dwarf* σε ένα δίκτυο συνδεδεμένων κόμβων. Στόχος είναι να διατηρηθεί η ευκολία της κατασκευής, της αναζήτησης και της ενημέρωσης της δομής αυτής και όλες οι λειτουργίες να γίνονται με online τρόπο σε ένα δίκτυο κόμβων αντί για έναν κεντρικό υπολογιστή.

Η διαδικασία δημιουργίας του *BD* κατανέμει τη δομή όσο οι πλειάδες διατρέχονται. Το Σχήμα 12 δείχνει πώς οι κόμβοι του δικτύου (1) μέχρι (9) επιλέγονται με αυτή τη σειρά για την αποθήκευση των αντίστοιχων κόμβων της δομής του Σχήματος 11. Έτσι δημιουργείται ένα αδόμητο δίκτυο P2P με βάση τη δεικτοδότηση που επιβάλλει η κεντρική δομή. Τα ερωτήματα και οι ενημερώσεις χρησιμοποιούν την ίδια διαδρομή που θα χρησιμοποιούσαν και στο *Dwarf*, ακολουθώντας τώρα πια δικτυακές συνδέσεις. Αν ένα ερώτημα αφορά το S1 θα προωθηθεί στον κόμβο (2). Από εκεί, ανάλογα με το group-by που ζητείται (ALL, C2 ή C3), θα καταλήξει στον κόμβο (3), (4) ή (5).

Συγκρινόμενο με το παραδοσιακό *Dwarf*, το *BD* προσφέρει τα εξής πλεονεκτήματα:

- Η ύπαρξη περισσότερων του ενός κόμβων επιτρέπει την παραλληλοποίηση των λειτουργιών του κύβου.
- Η κατανομή της δομής καθιστά δυνατό τον υπολογισμό πολύ μεγαλύτερων κύβων.
- Το σύστημα *BD* επιτρέπει online ενημερώσεις που προέρχονται από οποιονδήποτε κόμβο.
- Το προτεινόμενο σύστημα μπορεί να χειριστεί σημαντικά μεγαλύτερο ρυθμό αιτημάτων χωρίς να χρειάζεται να αντιγράψει όλη τη δομή, καθώς προσφέρει πολλαπλά σημεία εισόδου και προσαρμοστική αντιγραφή των πιο φορτωμένων κομματιών του κύβου.

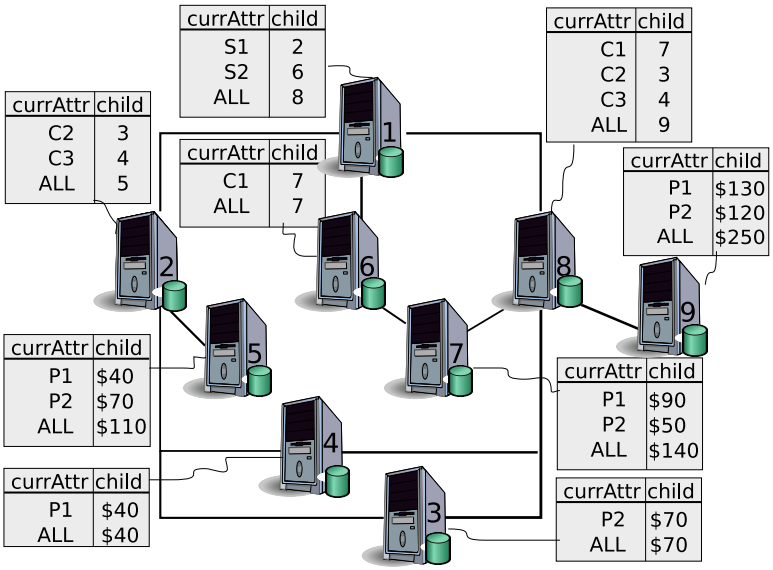
0.3.3 Σχεδίαση

Η γενική προσέγγιση του *BD* είναι η ακόλουθη: Κάθε κορυφή του γράφου (στο εξής *dwarf node*) ορίζεται με ένα μοναδικό ID (UID) και ανατίθεται σε έναν κόμβο του δικτύου. Θεωρούμε ότι κάθε δικτυακός κόμβος n γνωρίζει την ύπαρξη ενός αριθμού άλλων κόμβων, οι οποίοι συγκροτούν το *Σύνολο Γειτόνων* του, ή *Neighbor Set*, NS_n. Γειτονικοί dwarf κόμβοι αποθηκεύονται σε γειτονικούς δικτυακούς κόμβους στο στρώμα P2P προσθέτοντας συνδέσεις επικάλυψης. Έτσι, κάθε ακμή της κεντρικής δομής αντιπροσωπεύει μια δικτυακή σύνδεση ανάμεσα στον κόμβο n και σε έναν κόμβο που ανήκει στο NS_n του. Κάθε δικτυακός κόμβος διατηρεί έναν πίνακα που ονομάζουμε *hint* και είναι απαραίτητος για να καθοδηγεί τα ερωτήματα από τον έναν κόμβο στον άλλον μέχρι να βρεθεί η απάντηση, όπως και μία λίστα με τους κόμβους-γονείς του, την οποία ονομάζουμε *parent list* και απαιτείται στη διαδικασία της αντιγραφής για την αποφυγή ασυνεπειών.

Ο πίνακας *hint* είναι της μορφής (*currAttr, child*), όπου το *currAttr* είναι το τρέχον attribute του προς επίλυση ερωτήματος και το *child* είναι το UID του dwarf node στο οποίο οδηγεί το *currAttr*. Αν ο dwarf node που περιέχει το *currAttr* είναι κόμβος-φύλλο, το *child* είναι η συναθροιστική τιμή. Η parent list περιέχει τα UIDs των γονέων ενός κόμβου μαζί με το *currAttr*, του οποίου το *child* οδηγεί στο συγκεκριμένο κόμβο. Για να δρομολογηθούν τα μηνύματα ανάμεσα στους δικτυακούς κόμβους, κάθε κόμβος διατηρεί έναν πίνακα δρομολόγησης που αντιστοιχίζει UIDs σε *NIDs* (δηλαδή δικτυακά IDs, όπως ο συνδυασμός διεύθυνσης IP και πόρτας).

Εισαγωγή Δεδομένων

Τη δημιουργία του κύβου δεδομένων αναλαμβάνει ένας συγκεκριμένος κόμβος (*creator*), που έχει πρόσβαση στον πίνακα δεδομένων. Ο creator ακολουθεί τον αλγόριθμο κατασκευής του πρωτότυπου *Dwarf*, κατανέμοντας τους dwarf κόμβους εν κινήσει (on-the-fly) κατά την επεξεργασία ανά πλειάδα, αντί να τους κρατά σε δευτερεύον αποθηκευτικό μέσο. Γενικά, η δημιουργία ενός κελιού στον πρωτότυπο *Dwarf* ισοδυναμεί με την εισαγωγή μιας τιμής κάτω από το *currAttr* στον πίνακα hint. Η δημιουργία ενός dwarf node αντιστοιχεί στην εγγραφή μιας τιμής



Σχήμα 12: *Η κατανομή των κόμβων για το σύστημα Brown Dwarf του Πίνακα 3*

στο πεδίο *child*. Έτσι, όλες οι διαφορετικές τιμές των κελιών που ανήκουν σε έναν dwarf node καταγράφονται εν τέλει ως *currAttr*, ενώ τα παιδιά κάθε κόμβου καταγράφονται ως *child*.

Ο προτεινόμενος μηχανισμός εισαγωγής δεν προϋποθέτει την εκ των προτέρων δημιουργία του κεντρικού γράφου. Οι κόμβοι δημιουργούνται και οι πίνακες hint συμπληρώνονται σταδιακά κατά την επεξεργασία των πλειάδων. Η μόνη πληροφορία που χρειάζεται κάθε στιγμή είναι αυτή των d dwarf κόμβων που βρίσκονται στο μονοπάτι που διασχίζει η πλειάδα που υφίσταται επεξεργασία.

Για την πρώτη πλειάδα του Πίνακα 3, οι αντίστοιχοι κόμβοι και κελιά δημιουργούνται σε όλα τα επίπεδα της δομής (Σχήμα 12). Καθένας από τους dwarf κόμβους (1), (2), (3) του Σχήματος 11 ανατίθεται σε δικτυακούς κόμβους. Στον πίνακα hint του κόμβου (1), το S_1 οδηγεί στον κόμβο (2). Ακολουθώντας την ίδια διαδικασία, ο πίνακας hint του (2) συμπληρώνεται με το C_2 και το (3) ενώ ο πίνακας hint του (3) με το P_2 και το \$70. Η διαδικασία προχωρά στην επόμενη πλειάδα, η οποία μοιράζεται με την προηγούμενη το κοινό πρόθεμα S_1. Αυτό σημαίνει ότι το C_3 πρέπει να εισαχθεί στον κόμβο (2) και πρέπει να δημιουργηθεί ο κόμβος (4). Επιπλέον "κλείνει" ο κόμβος (3), οπότε καταγράφεται η τιμή *ALL* μαζί με την συναθροιστική τιμή \$70 στον πίνακα hint του. Σταδιακά, δημιουργούνται όλοι οι απαραίτητοι κόμβοι και οι πίνακες hint συμπληρώνονται με τις κατάλληλες πληροφορίες δρομολόγησης.

Αναζήτηση Δεδομένων

Τα ερωτήματα επιλύονται ακολουθώντας την κατάλληλη διαδρομή κατά μήκος της κατανεμημένης δομής του *BD* πεδίο προς πεδίο. Η τιμή κάθε διάστασης στο ερώτημα ανήκει σε έναν

dwarf node ο οποίος μέσω του πίνακα hint οδηγεί στον κόμβο που είναι υπεύθυνος για την τιμή της επόμενης διάστασης.

Από την παραπάνω περιγραφή είναι εμφανές ότι το σύστημα χρειάζεται ένα σημείο εκκίνησης, δηλαδή κάθε κόμβος που θέτει ένα ερώτημα πρέπει να γνωρίζει τον κόμβο-ρίζα της δομής, από όπου θα ξεκινήσει η επίλυση. Αυτό επιτυγχάνεται μέσω κάποιου μηχανισμού διαφήμισης, τον οποίο καλεί ο κόμβος-ρίζα κατά τη δημιουργία του. Η ύπαρξη ενός μοναδικού σημείου εκκίνησης για το *BD*, που αποτελεί μοναδικό σημείο αποτυχίας (single point of failure), αντιμετωπίζεται από τη στρατηγική αντιγραφής που παρουσιάζεται στη συνέχεια.

Πίσω στο παράδειγμά μας, η αναζήτηση για το ερώτημα $S_1 ALL P_2$ ξεκινά από τον κόμβο (1), του οποίου το child που αντιστοιχεί στην τιμή S_1 οδηγεί στον κόμβο (2). Από εκεί, το ερώτημα ακολουθεί τη διαδρομή που υποδεικνύεται από το τρίτο στοιχείο του πίνακα hint, οπότε επισκέπτεται το (5). Η τιμή P_2 περιορίζει τις πιθανές επιλογές στο δεύτερο στοιχείο του πίνακα, δηλαδή στην τιμή \$70 που είναι και η απάντηση.

Ενημέρωση Δεδομένων

Η διαδικασία της ενημέρωσης είναι παρόμοια με αυτήν της εισαγωγής πλειάδων, μόνο που στην περίπτωση αυτή πρέπει να ανακαλυφθεί το μεγαλύτερο κοινό πρόθεμα μεταξύ της τρέχουσας πλειάδας και της υπάρχουσας δομής, ακολουθώντας τις δικτυακές συνδέσεις. Μόλις ο δικτυακός κόμβος που αποθηκεύει το τελευταίο κοινό πεδίο βρεθεί, οι υφιστάμενοι κόμβοι ενημερώνονται αναδρομικά. Αυτό σημαίνει ότι υπάρχοντες κόμβοι διευρύνονται για να φιλοξενήσουν καινούρια κελιά και νέοι κόμβοι δημιουργούνται όταν αυτό είναι απαραίτητο. Επιπλέον, η εισαγωγή μιας καινούριας πλειάδας σε υπάρχουσα δομή *BD* επηρεάζει και όλα τα κελιά *ALL* που σχετίζονται με τους προς ενημέρωση κόμβους. Επειδή η διαδικασία των ενημερώσεων είναι τόσο δαπανηρή σε εύρος ζώνης, υποθέτουμε ότι καλείται όταν ένα σύνολο από ενημερώσεις έχει συλλεχθεί.

Αντιγραφή Δεδομένων

Για να διασφαλιστεί η διαθεσιμότητα και να αποφευχθεί η ύπαρξη μοναδικού σημείου αποτυχίας, ιδίως στην περίπτωση του κόμβου-ρίζας, θεωρούμε την καθολική παράμετρο αντιγραφής k. Η παράμετρος αυτή καθορίζει το βαθμό πλεονασμού των δεδομένων στο σύστημα: Κατά τη δημιουργία του κύβου, κάθε dwarf node αποθηκεύεται σε $k+1$ δικτυακούς κόμβους αντί σε έναν μόνο. Έτσι, στην αρχική του κατάσταση, το σύστημα φιλοξενεί $k+1$ αντίγραφα (*mirrors*) κάθε dwarf κόμβου. Κατά την επίλυση ενός ερωτήματος ο κόμβος που λαμβάνει το ερώτημα επιλέγει να το προωθήσει τυχαία σε ένα από τα αντίγραφα του κόμβου-παιδιού.

Για να επιτευχθεί η σωστή συμπεριφορά μετά τη διαδικασία αντιγραφής, οι γονείς, τα παιδιά και τα αντίγραφα ενός κόμβου πρέπει να ενημερωθούν για ένα νέο αντίγραφό του. Οι κόμβοι-γονείς πρέπει να γνωρίζουν για το νέο κόμβο ώστε να τον συμπεριλάβουν στη διαδικασία αναζήτησης. Ο καινούριος κόμβος λαμβάνει τον πίνακα hint και την λίστα των γονέων από κάποιο

από τα αντίγραφά του. Τέλος οι κόμβοι-παιδιά πρέπει να ενημερωθούν για τον καινούριο κόμβο-πατέρα.

Από το σημείο αυτό και μετά, το σύστημα είναι υπεύθυνο για τη διατήρηση των αντιγράφων κάθε κόμβου πάνω από k. Για να εξακριβωθεί η διαθεσιμότητα ή όχι ενός dwarf node, οι δικτυακοί κόμβοι στέλνουν περιοδικά μεταξύ τους μηνύματα ping. Εκτός από τη διαδικασία αυτή, η διαθεσιμότητα εξακριβώνεται και μέσω των ερωτημάτων, όταν ένας κόμβος στην πορεία του ερωτήματος δεν απαντήσει. Όταν ένας κόμβος αντιληφθεί την αστοχία κάποιου αντιγράφου του (για παράδειγμα λόγω σφάλματος του δικτύου), τότε επιλέγει κάποιο γειτονικό του κόμβο και αντιγράφει το dwarf node σε αυτόν. Η διαδικασία αυτή δεν επηρεάζει την υπόλοιπη λειτουργία του συστήματος, καθώς όλα τα ερωτήματα συνεχίζουν να επιλύονται κανονικά.

Χειρισμός Αστοχιών Κόμβων και Πόλωσης Φορτίου

Μια βασική απαίτηση κάθε κατανεμημένης εφαρμογής είναι η ανοχή σε σφάλματα. Οι συστοιχίες υπολογιστών αποτελούνται συνήθως από υλικό επιρρεπές σε αστοχίες. Η πόλωση του εισερχόμενου φορτίου είναι άλλος ένας παράγοντας που επηρεάζει την ικανότητα του συστήματος να λειτουργεί σωστά, καθώς μειώνει την απόδοση των υπερφορτωμένων κόμβων, έχοντας επίπτωση στο χρόνο επίλυσης των ερωτημάτων. Για να αντιμετωπιστούν τέτοιες καταστάσεις σε κατανεμημένα συστήματα χρησιμοποιούνται συνήθως τεχνικές αντιγραφής δεδομένων.

Στο σύστημα *BD*, χρησιμοποιούμε μια μέθοδο αντιγραφής που προσαρμόζεται τόσο στην πόλωση του φορτίου όσο και στην κινητικότητα των κόμβων, ώστε να αντιμετωπίσει και τα δύο ζητήματα με ομοιόμορφο τρόπο, διαστέλλοντας δημοφιλή κομμάτια της δομής και συρρικνώνοντας άλλα, που δέχονται λίγα αιτήματα. Με τον τρόπο αυτό, το *BD* είναι σε θέση να αποσπά αυξημένους πόρους για να χειρίζεται εξάρσεις στο φορτίο και να τους απελευθερώνει μόλις αυτές υποχωρούν.

Κινητικότητα Κόμβων

Όταν ένας κόμβος θέλει να αποχωρήσει από το σύστημα, τότε για κάθε dwarf κόμβο που αποθηκεύει, ενημερώνει τους κόμβους-γονείς, τους κόμβους-παιδιά και τους κόμβους-αντίγραφα για να διαγράψουν τις συνδέσεις τους. Όταν ένας κόμβος φύγει ξαφνικά από το δίκτυο, η αποχώρησή του γίνεται αντιληπτή είτε από την περιοδική διαδικασία ping είτε μέσω της δρομολόγησης των ερωτημάτων. Στην πρώτη περίπτωση, το αντίγραφο ενός κόμβου είναι αυτό που αντιλαμβάνεται την αστοχία και ξεκινά τη διαδικασία δημιουργίας ενός καινούριου αντιγράφου σε κάποιο γειτονικό του κόμβο. Στη δεύτερη περίπτωση η αστοχία ανακαλύπτεται από κάποιον κόμβο-γονέα. Ο τελευταίος, εκτός από το ερώτημα, προωθεί σε κάποιο άλλο από τα πιθανά παιδιά (που είναι αντίγραφα του κόμβου που αποχώρησε) και την εντολή να δημιουργήσει αντίγραφο των δεδομένων του. Τελικά, οι γονείς, τα παιδιά αλλά και τα αντίγραφα του κόμβου που απέτυχε ενημερώνονται για το συμβάν και ανανεώνουν τις συνδέσεις τους.

Αντιγραφή Βασισμένη στο Φορτίο

Στο *BD*, οι δικτυακοί κόμβοι χρησιμοποιούν μια μέθοδο αντιγραφής που προσαρμόζεται στο

εισερχόμενο ανά dwarf node φορτίο. Συγκεκριμένα, ένας κόμβος που φιλοξενεί έναν υπερφορτωμένο dwarf node μπορεί να δημιουργήσει πρόσθετα αντίγραφα μέσω της διαδικασίας της *διαστολής (expansion)*. Τα νέα αντίγραφα θα χρησιμοποιηθούν από τους κόμβους-γονείς ώστε να αναλάβουν μέρος του φορτίου ερωτημάτων. Στην αντίθετη περίπτωση, ένας κόμβος που λαμβάνει ελάχιστα ερωτήματα μπορεί να σβηστεί από το σύστημα μέσω της διαδικασίας της *συστολής (shrink)*, αρκεί ο συνολικός αριθμός αντιγράφων του εν λόγω κόμβου να παραμένει τουλάχιστον $k+1$. Η διαγραφή αυτή απελευθερώνει πόρους που μπορούν να χρησιμοποιηθούν για πιο δημοφιλή κομμάτια της δομής.

Οι διαδικασίες αυτές προϋποθέτουν ότι κάθε κόμβος που συμμετέχει στο *BD* ελέγχει το εισερχόμενο φορτίο για καθέναν από τους dwarf κόμβους dn_s, $s \in (i, i+1, \ldots, j)$, που αποθηκεύει. Αν $l_s(t)$ είναι το τρέχον φορτίο για τον dn_s, οι δύο διαδικασίες μπορούν να περιγραφούν ως εξής:

Διαστολή: Καθώς ο φόρτος αυξάνεται λόγω των εισερχόμενων ερωτημάτων, κάποιοι dwarf κόμβοι φτάνουν το όριό τους, το οποίο εκφράζεται από την παράμετρο $Limit^s_{exp}$. Αυτή αντιπροσωπεύει το μέγιστο των αιτημάτων που είναι σε θέση να εξυπηρετήσει κάθε dwarf κόμβος dn_s στη μονάδα του χρόνου. Όταν το όριο αυτό ξεπεραστεί, ο δικτυακός κόμβος επικαλείται τη διαδικασία αντιγραφής. Πιο συγκεκριμένα, κάθε dn_s, με $l_s(t) > Limit^s_{exp}$ θα αντιγραφεί $\lceil l_s(t)/Limit^s_{exp} \rceil$ φορές. Αυτός ο μηχανισμός επιτρέπει τη δυναμική διαστολή των πόρων σύμφωνα με τη ζήτηση και βοηθάει τους υπερφορτωμένους κόμβους να απαλλαγούν από ένα μέρος του φορτίου τους μεταφέροντας το σε άλλους κόμβους.

Συστολή: Οι μεταβολές στο φορτίο μπορούν να επιφέρουν τη δημιουργία αντιγράφων που μακροπρόθεσμα δε χρησιμοποιούνται. Το σύστημα πρέπει να είναι σε θέση να διαγράφει τέτοια αντίγραφα, αρκεί να εξασφαλίσει ότι ο συνολικός αριθμός τους δε θα πέφτει κάτω από $k+1$. Θεωρώντας ότι το $Limit^s_{shr}$ είναι το όριο κάτω από το οποίο ένας κόμβος dn_s υπολειτουργεί και r_s είναι ο αριθμός των αντιγράφων του dn_s, τότε κάθε dn_s με $l_s(t) < Limit^s_{shr}$ και $r_s > k+1$ διαγράφεται. Για να διασφαλιστεί ότι η διαγραφή του dn_s δε θα επιφέρει υπερφόρτωση των αντιγράφων του, ορίζουμε την τιμή του $Limit^s_{shr}$ ίση με $Limit^s_{exp} \cdot r_s/(r_s + c)$, όπου c θετική σταθερά.

0.3.4 Πειραματική Αποτίμηση

Το *BD* έχει υλοποιηθεί σε Java και έχει εγκατασταθεί σε πραγματικό σύστημα αποτίμησης $N = 16$ κόμβων (Quad Core @ 2.0 GHz, 4GB RAM). Η κεντρική έκδοση έχει υλοποιηθεί επίσης, για άμεση σύγκριση.

Στα πειράματά μας χρησιμοποιούμε τόσο συνθετικά όσο και πραγματικά πολυδιάστατα δεδομένα. Τα συνθετικά έχουν παραχθεί από δικό μας γεννήτορα αλλά και από το γεννήτορα του APB-1 benchmark και ακολουθούν διάφορες κατανομές (ομοιόμορφη, αυτο-όμοια (self-similar)

Πίνακας 4: *Οι απαιτήσεις σε αποθηκευτικό χώρο και οι χρόνοι δημιουργίας για κύβους* Dwarf *και* Brown Dwarf *με διαφορετικό αριθμό διαστάσεων*

		Ομοιόμορφη				80-20				Zipf			
d	πίνακας	μέγεθος (MB)		χρόνος (sec)		μέγεθος (MB)		χρόνος (sec)		μέγεθος (MB)		χρόνος (sec)	
	(MB)	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*
5	0.2	1	1	4	4	1	1	8	7	1	1	3	4
10	0.4	4	5	31	13	4	5	28	14	6	7	54	21
15	0.6	7	9	63	29	10	13	96	43	22	27	226	74
20	0.8	13	17	122	50	18	23	352	82	54	69	543	204
25	1.0	18	23	198	88	29	37	729	196	152	195	1206	535

80/20 και Zipf με $\theta = 0.95$). Τα φορτία ερωτημάτων περιλαμβάνουν τόσο σημειακά όσο και συναθροιστικά ερωτήματα με διαφορετικές αναλογίες και κατανομές.

Δημιουργία του Κύβου

Για βαθμό αντιγραφής $k = 0$, κατασκευάζουμε τους *BD* και *Dwarf* κύβους που αποτελούνται από 10k πλειάδες με αριθμό διαστάσεων από 5 έως 25 και ακολουθούν την ομοιόμορφη, την αυτο-όμοια 80-20 και την Zipf ($\theta = 0.95$) κατανομή. Η κατανάλωση αποθηκευτικού χώρου και οι χρόνοι εισαγωγής παρουσιάζονται στον Πίνακα 4.

Το σύστημά μας επιδεικνύει εντυπωσιακά γρηγορότερη δημιουργία του κύβου σε σχέση με την κεντρική μέθοδο, λόγω της επικάλυψης που επιτρέπει στη διαδικασία αποθήκευσης (κάθε κόμβος αποθηκεύει ανεξάρτητα το μέρος του κύβου που του αναλογεί). Η επιτάχυνση είναι πιο εμφανής όσο ο αριθμός των διαστάσεων αυξάνεται και η πόλωση γίνεται εντονότερη, γιατί έτσι δημιουργούνται μεγαλύτεροι κύβοι. Για παράδειγμα, το *BD* εισάγει τον πολωμένο κύβο 25 διαστάσεων μέχρι 3.5 φορές πιο γρήγορα από το *Dwarf*.

Το *BD* επιφέρει ένα μικρό επιπλέον κόστος στον αποθηκευτικό χώρο, το οποίο όμως μοιράζεται ανάμεσα στους συμμετέχοντες κόμβους. Ενδεικτικά για τον κύβο 25 διαστάσεων, παρόλο που το επιπλέον κόστος είναι 43MB, κάθε κόμβος από τους 16 επιφορτίζεται με λιγότερο από 3MB. Έτσι, το μεγαλύτερο πλεονέκτημα του *BD* είναι ότι μπορεί να αποθηκεύσει σχεδόν N φορές τα δεδομένα που αποθηκεύει ο *Dwarf* (για $k = 0$), χρησιμοποιώντας N υπολογιστές με τις ίδιες δυνατότητες.

Ενημερώσεις

Χρησιμοποιώντας τα ίδια σύνολα δεδομένων όπως στο προηγούμενο πείραμα, θέτουμε ενημερώσεις μεγέθους 1% επί του συνόλου των πλειάδων, που ακολουθούν την ομοιόμορφη και την αυτο-όμοια κατανομή (80-20). Τα αποτελέσματα παρουσιάζονται στον Πίνακα 5.

Εκμεταλλευόμενο την παραλληλοποίηση της διαδικασίας, το *BD* αποδεικνύεται μέχρι 2.3 φορές πιο γρήγορο για δεδομένα πολλών διαστάσεων. Ο αριθμός των διαστάσεων παίζει σημαντικό ρόλο τόσο στο χρόνο όσο και στο κόστος των ενημερώσεων. Όσο περισσότερες οι διαστάσεις, τόσο μεγαλύτερος ο κύβος που δημιουργείται, οπότε τόσο περισσότεροι οι dwarf κόμβοι

Πίνακας 5: *Η επίδραση ενημερώσεων σε δεδομένα διαφορετικών διαστάσεων*

d	Ομοιόμορφη			80-20		
	χρόνος (sec)		μηνύματα/ενημ.	χρόνος (sec)		μηνύματα/ενημ.
	Dwarf	*BD*	*BD*	*Dwarf*	*BD*	*BD*
5	7.1	7.2	14.6	7.5	6.4	13.7
10	17.7	14.3	50.8	21.3	14.4	49.8
15	30.8	21.8	111.0	43.4	31.2	120.4
20	48.6	27.9	193.3	104.1	65.8	200.2
25	89.1	39.1	300.7	172.1	103.6	305.7

Πίνακας 6: *Οι χρόνοι επίλυσης και το κόστος επικοινωνίας για διαφορετικά σύνολα 1k ερωτημάτων*

d	Ομοιόμορφη			Zipf		
	χρόνος (sec)		μηνύματα/ερώτημα	χρόνος (sec)		μηνύματα/ερώτημα
	Dwarf	*BD*	*BD*	*Dwarf*	*BD*	*BD*
5	5.2	4.0	5.8	1.9	1.7	5.5
10	30.1	2.6	10.9	29	1.2	10.6
15	65.2	2.9	15.6	55.4	1.2	15.5
20	102.1	3.0	20.8	88.3	1.5	20.3
25	182.5	13.2	25.9	172.1	9.2	25.6

που επηρεάζονται. Επίσης, πολωμένα δεδομένα ενημερώνονται πιο αργά λόγω της ύπαρξης πυκνών σημείων στους αντίστοιχους κύβους.

Επεξεργασία Ερωτημάτων

Χρησιμοποιώντας τα ίδια δεδομένα θέτουμε 1k ερωτήματα που ακολουθούν την ομοιόμορφη κατανομή και την κατανομή Zipf ($\theta = 0.95$), με το ποσοστό των σημειακών ερωτημάτων ίσο με 50%. Επίσης, η πιθανότητα P_d να μη συμμετέχει μια διάσταση στο ερώτημα τίθεται ίση με 0.3. Ο Πίνακας 6 παρουσιάζει τα αποτελέσματα.

Καταρχάς, παρατηρούμε ότι σε όλες τις περιπτώσεις το *BD* επιλύει τα ερωτήματα πολύ πιο γρήγορα από την κεντρική έκδοση. Ενώ ο χρόνος απόκρισης αυξάνεται με τις διαστάσεις στο *Dwarf*, στο *BD* παραμένει σχεδόν σταθερός. Η επίλυση κάθε attribute ενός ερωτήματος μπορεί να γίνει ατομικά, από διαφορετικούς κόμβους. Έτσι, έχοντας 16 κόμβους να εκτελούν λειτουργίες εισόδου/εξόδου παράλληλα, βελτιώνεται αισθητά η απόδοση. Ιδίως στην περίπτωση πολωμένων φορτίων, το *BD* επιδεικνύει εντυπωσιακή επιτάχυνση μέχρι και 60 φορές σε σχέση με το *Dwarf*.

Επιπλέον ο αριθμός των μηνυμάτων ανά ερώτημα σε όλες τις περιπτώσεις έχει άνω όριο το $d + 1$: Το σύστημα χρειάζεται d μηνύματα για να προωθήσει το ερώτημα κατά μήκος του μονοπατιού του μέχρι την απάντηση και ένα μήνυμα για να στείλει την απάντηση πίσω στον κόμβο που την έθεσε.

Πίνακας 7: *Μετρήσεις για διάφορα σύνολα δεδομένων APB*

		μέγεθος (MB)		χρόνος εισαγωγής (sec)		χρόνος αναζήτησης (sec)	
Πυκνότητα	#πλειάδων	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*
0.1	1.2M	17	20	42	16	40	12
0.2	2.5M	35	41	82	32	55	12
0.3	3.7M	51	60	126	53	80	12
0.5	6.2M	74	98	314	93	93	13

Πίνακας 8: *Μετρήσεις για πραγματικά δεδομένα*

	μέγεθος (MB)		χρόνος εισαγωγής (sec)		A (sec)		B (sec)		C (sec)	
Δεδομένα	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*
Weather	9.3	11.4	120	23	234	11	165	12	114	12
Forest	8.0	9.8	66	20	144	11	111	11	70	12

Πραγματικά Δεδομένα και Δεδομένα Benchmark

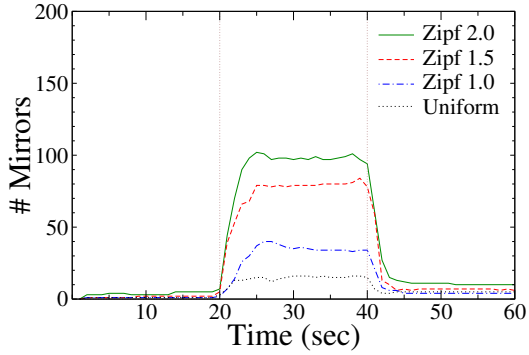
Σε αυτό το σημείο εξετάζεται η συμπεριφορά του *BD* με πιο ρεαλιστικά δεδομένα. Με τη βοήθεια του γεννήτορα του APB-1 benchmark [apb] δημιουργούμε τέσσερα σύνολα δεδομένων με διαφορετικές πυκνότητες. Επίσης χρησιμοποιούνται δύο σύνολα δεδομένων που αποτελούνται από 10k πλειάδες, το *Weather* [wea] με 9 διαστάσεις που αντιστοιχεί σε μετρήσεις καιρικών μεγεθών και το *Forest* [Bla] με 10 διαστάσεις που περιλαμβάνει μετρήσεις μεγεθών που σχετίζονται με δάση. Τα δεδομένα αυτά έχουν χρησιμοποιηθεί σε σχετικές μελέτες στη βιβλιογραφία.

Τα φορτία που τίθενται στα δεδομένα του APB παράγονται από το γεννήτορα του benchmark και αποτελούνται από 1k ερωτήματα, τόσο σημειακά όσο και συναθροιστικά. Για τα πραγματικά δεδομένα παράγουμε φορτία που αποτελούνται από 10k ερωτήματα που ακολουθούν την κατανομή Zipf με θ από 0 έως 2 (A, B και C). Το ποσοστό των σημειακών ερωτημάτων είναι 0.5, ενώ για τα συναθροιστικά ερωτήματα θέτουμε $P_d = 0.3$.

Τα αποτελέσματα παρουσιάζονται στους Πίνακες 7 και 8 και συνάδουν με τα ευρήματα των προηγούμενων πειραμάτων. Όσον αφορά τους χρόνους δημιουργίας του κύβου, το *BD* είναι εμφανώς πιο γρήγορο από το *Dwarf*. Τα αποτελέσματα δείχνουν ότι η κατανεμημένη εκδοχή είναι πάνω από 5 φορές ταχύτερη από την κεντρική, δίνοντας εντυπωσιακούς χρόνους κατασκευής κύβων (περίπου 1.5 λεπτό για 6.2M πλειάδες). Η επίλυση των ερωτημάτων είναι μέχρι 20 φορές γρηγορότερη για το *BD*, το οποίο μπορεί να χειριστεί σχεδόν 1k ερωτήματα το δευτερόλεπτο.

Κλιμακωσιμότητα

Χρησιμοποιώντας τον κύβο των 10 διαστάσεων θέτουμε φορτία που αποτελούνται από 5k ερωτήματα και ακολουθούν την ομοιόμορφη κατανομή και την κατανομή Zipf (με διαφορετικές τιμές του θ) με P_d=0.3. Επιπλέον το όριο διαστολής τίθεται ίσο με 10 $\frac{\text{ερωτήματα}}{\text{sec}}$ ($Limit^s_{exp}$=10).



Σχήμα 13: *Αριθμός αντιγράφων ως προς το χρόνο για παλμικό ρυθμό ερωτημάτων και* $Limit^{s}_{exp} = 10$

Πίνακας 9: *Οι συνέπειες αυξανόμενου αριθμού αστοχιών στην επεξεργασία δεδομένων και ερωτημάτων για διαφορετικές τιμές της παραμέτρου* T_{fail}

$\|N_{fail}\|$	$T_{fail}(sec)$	απώλειες ερωτ. (%)	επαναπροωθήσεις	μηνύματα/ερώτ.	χρόνος/ερώτ. (ms)
0	-	0	0	9.8	57
1	90	0	11	9.8	79
2	90	0	204	10.4	257
4	90	2.9	841	11.1	734
1	60	0	21	9.9	107
2	60	0	258	10.5	304
4	60	4.3	894	11.2	812

Για να εξετάσουμε τη συμπεριφορά του συστήματος κάτω από συνθήκες πίεσης και ξαφνικών αλλαγών στο φορτίο, ξεκινάμε από ένα αρχικό ρυθμό ερωτημάτων ίσο με 10 $\frac{\text{ερωτήματα}}{\text{sec}}$, που ξαφνικά αυξάνει στο δεκαπλάσιο (το λ φτάνει την τιμή 100 $\frac{\text{ερωτήματα}}{\text{sec}}$) μετά από 20 sec. Μετά από άλλα 20 sec, ο ρυθμός επανέρχεται στην αρχική τιμή του. Αποτιμάται η ικανότητα του μηχανισμού της διαστολής (*expansion*) και της συστολής (*shrink*) να αντιλαμβάνεται τις αλλαγές και να προσαρμόζει αναλόγως τον αριθμό των αντιγράφων.

Το Σχήμα 13 παρουσιάζει τον αριθμό των αντιγράφων ως προς το χρόνο. Σχεδόν αμέσως μετά την απότομη αύξηση του λ, αυξάνεται και ο αριθμός των αντιγράφων επί δέκα. Μετά το τέλος του παλμού, ο μηχανισμός διαστολής διαγράφει τους κόμβους dwarf που δε δέχονται ερωτήματα, αποδεσμεύοντας αποθηκευτικό χώρο. Το *BD* καταφέρνει να ανιχνεύσει την αλλαγή στο φορτίο και μέσα σε λίγα δευτερόλεπτα τα αντίγραφα μειώνονται, φτάνοντας στον αρχικό τους αριθμό.

Ανοχή σε Σφάλματα

Το σύστημα στηρίζεται σε υπολογιστές εμπορίου, οπότε είναι πολύ πιθανόν να συμβούν αστοχίες κατά τη διάρκεια της λειτουργίας του. Χρησιμοποιώντας τα δεδομένα 10 διαστάσεων με βαθμό αντιγραφής k=3 και ένα φορτίο με 5K ερωτήματα που καταφθάνουν με ρυθμό 10 $\frac{\text{ερωτήματα}}{\text{sec}}$, προκαλούμε αποτυχίες στους κόμβους ως εξής: Κάθε T_{fail} sec, ένα υποσύνολο n_{fail} των κόμβων

που είναι online αποτυγχάνουν κυκλικά, ενώ οι κόμβοι που προηγουμένως ήταν offline εισάγονται ξανά στο δίκτυο. Ξεκινώντας από $|n_{\text{fail}}|$=1 σταδιακά αυξάνουμε την τιμή μέχρι το 4 (μιας και κάθε κόμβος dwarf υπάρχει σε k+1=4 διαφορετικούς δικτυακούς κόμβους). Ο Πίνακας 9 παρουσιάζει τα αποτελέσματα. Σημειώνεται ότι η στήλη που καταγράφει το χρόνο αναζήτησης αναφέρεται στην απόλυτη τιμή για την επίλυση ενός μεμονωμένου ερωτήματος, όχι το μέσο όρο επίλυσης για ένα σύνολο ερωτημάτων.

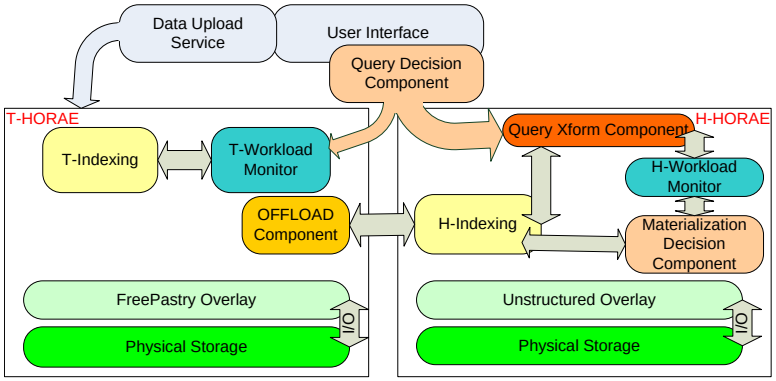
Παρατηρούμε ότι το σύστημα διατηρεί το θεωρητική εγγύηση ότι για οποιοδήποτε επίπεδο σφαλμάτων κάτω από το βαθμό αντιγραφής ($|n_{\text{fail}}| < 4$) δε συμβαίνει καμία απώλεια δεδομένων ή ερωτημάτων. Ακόμα κι όταν το 25% των κόμβων αποτυγχάνει, ένα μικρό ποσοστό των ερωτημάτων πρέπει να τεθούν εκ νέου (λιγότερο από 5% στη χειρότερη περίπτωση). Επίσης, λόγω της αυτόματης αποκατάστασης των αντιγράφων, η απώλεια των ερωτημάτων είναι μικρή. Οι επαναπροωθήσεις και οι καθυστερήσεις αυξάνουν το μέσο χρόνο απόκρισης κατά έναν παράγοντα κοντά στο 13. Οι μετρήσεις δείχνουν επιπλέον ότι ο αριθμός των αντιγράφων παραμένει σταθερός και πολύ κοντά στον αρχικό, παρά τις αναχωρήσεις των κόμβων.

0.3.5 Ανακεφαλαίωση

Το *Brown Dwarf* είναι ένα σύστημα που κατανέμει ένα κύβο δεδομένων στους κόμβους ενός αδόμητου δικτύου P2P. Αυτή είναι μια μοναδική προσέγγιση που δίνει τη δυνατότητα στους χρήστες να θέτουν group-by ερωτήματα και ενημερώσεις σε πολυδιάστατα δεδομένα online, χωρίς τη χρήση κάποιου εξειδικευμένου εργαλείου.

Το προτεινόμενο σύστημα είναι επεκτάσιμο, αφού μπορεί να χρησιμοποιήσει απεριόριστο αριθμό συνεργαζόμενων κόμβων, παρέχει διαθεσιμότητα δεδομένων μέσω της προσαρμοστικής αντιγραφής που βασίζεται τόσο στο φορτίο όσο και στις αστοχίες των κόμβων, απαντά αποδοτικά σημειακά και συναθροιστικά ερωτήματα σε περιορισμένο αριθμό βημάτων και τέλος καταναλώνει όσο αποθηκευτικό χώρο χρειάζεται χάρη στο μηχανισμό διαστολής–συστολής.

Η πειραματική αποτίμηση του συστήματος δείχνει ότι ο κύβος των δεδομένων κατανέμεται δίκαια ανάμεσα στους συνεργαζόμενους κόμβους. Τόσο οι χρόνοι δημιουργίας όσο και οι χρόνοι αναζήτησης μειώνονται δραστικά (πολλές φορές κατά τάξη μεγέθους) χάρη στην παραλληλοποίηση. Επίσης, το *Brown Dwarf* αντιγράφει δημοφιλή κομμάτια της δομής χρησιμοποιώντας μόνο τοπικές μετρήσεις και ελαχιστοποιεί την υπερφόρτωση των κόμβων ακόμα και σε δυναμικά περιβάλλοντα.



Σχήμα 14: *Η αρχιτεκτονική του συστήματος* HORAE

0.4 Το Σύστημα HORAE

0.4.1 Επισκόπηση

Στο σημείο αυτό παρουσιάζεται η πλατφόρμα *HORAE*†, όπου εφαρμόζονται τεχνικές από τον τομέα των κατανεμημένων συστημάτων και των αποθηκών δεδομένων για την αποθήκευση, αναζήτηση και ενημέρωση πολυδιάστατων χρονικών σειρών. Η υλοποίησή του συνδυάζει μια ισχυρή μηχανή δεικτοδότησης για μεγάλο όγκο δεδομένων, τόσο ιστορικών όσο και πραγματικού χρόνου, με μια αρχιτεκτονική shared-nothing που διασφαλίζει κλιμακωσιμότητα και διαθεσιμότητα σε χαμηλό κόστος. Η συνεισφορά του έγκειται στα ακόλουθα:

- Παρουσιάζει ένα ολοκληρωμένο σύστημα δεικτοδότησης, επεξεργασίας ερωτημάτων και ενημέρωσης πολυδιάστατων χρονικών σειρών που παράγονται σε υψηλούς ρυθμούς. Βασίζεται σε μια κατανεμημένη αρχιτεκτονική από υπολογιστές του εμπορίου, χωρίς να απαιτεί τη χρήση κάποιου εξειδικευμένου εργαλείου.

- Προτείνει προηγμένα χαρακτηριστικά που του επιτρέπουν να προσαρμόζει τη συμπεριφορά του στο εισερχόμενο φορτίο. Τόσο το επίπεδο της λεπτομέρειας στο οποίο γίνεται η υλοποίηση του κύβου όσο και το πλήθος των πόρων που συμμετέχουν μεταβάλλονται δυναμικά για καλύτερη απόδοση, βέλτιστη χρήση του αποθηκευτικού χώρου και ανοχή σε σφάλματα.

- Υλοποιείται και συγκρίνεται με το Hive [TSJ+09] τόσο σε συνθετικά όσο και σε πραγματικά δεδομένα, αποδεικνύοντας ότι μειώνει σημαντικά το χρόνο επίλυσης των ερωτημάτων ενώ παρέχει ελαστικότητα και διαθεσιμότητα ανάλογα με τις απαιτήσεις της εφαρμογής.

†Οι Ώρες ήταν οι θεότητες των εποχών και των ωρών στην Ελληνική μυθολογία

0.4.2 Σχεδίαση

Το *HORAE* είναι μια ολοκληρωμένη λύση που χρησιμοποιεί τεχνολογίες αποθηκών δεδομένων και κατανεμημένων συστημάτων για την οργάνωση και ανάλυση πολυδιάστατων δεδομένων με κλιμακώσιμο και αποδοτικό τρόπο. Χειρίζεται δεδομένα χαρακτηριζόμενα από τη διάσταση του χρόνου, που παράγονται με υψηλούς ρυθμούς και εισέρχονται στο σύστημα χρονικά ταξινομημένα ‡. Ως στόχοι του συστήματος τίθενται η αποδοτική και online επεξεργασία των ενημερώσεων που παράγονται με υψηλό ρυθμό, η επίλυση των ερωτημάτων διαφόρων επιπέδων λεπτομέρειας, η κλιμακωσιμότητα, η ανοχή σε σφάλματα και η ευκολία χρήσης.

Η αρχιτεκτονική του συστήματος αποτελείται από δύο συμπληρωματικά υποσυστήματα (Σχήμα 14): Το *T-HORAE*, που αποτελεί το αλληλεπιδραστικό (transactional) κομμάτι, είναι υπεύθυνο για την αποθήκευση και δεικτοδότηση των εισερχόμενων ενημερώσεων, ενώ το *H-HORAE*, το ιστορικό (historical) κομμάτι, αποθηκεύει το μεγάλο όγκο των δεδομένων, καθώς αυτά μεταφέρονται από το T-HORAE. Η λογική πίσω από αυτόν το διαχωρισμό απορρέει από την ανάγκη τόσο για αποδοτική ανάλυση όσο και για online επεξεργασία των ενημερώσεων. Επιπλέον, επειδή στοχεύουμε σε χρονικά δεδομένα, περιμένουμε (αλλά δεν απαιτούμε) ότι τα πρόσφατα δεδομένα θα αναζητούνται με μεγαλύτερη λεπτομέρεια από ότι τα ιστορικά. Έτσι παρέχουμε ένα σύστημα με αλληλεπιδραστική λογική, που όμως υλοποιεί τα περιεχόμενά του ασύγχρονα μέσω ενός συστατικού με τη λογική μιας αποθήκης δεδομένων.

Τα δεδομένα που χειρίζεται το σύστημα είναι d διαστάσεων, με τη διάσταση του χρόνου να θεωρείται η πρωτεύουσα. Για λόγους απλότητας, θεωρούμε ότι υπάρχει ιεραρχία μόνο στη διάσταση αυτή. Ωστόσο, το σύστημά μας μπορεί εύκολα να γενικευθεί ώστε να υποστηρίζει ιεραρχίες σε όλες τις διαστάσεις. Η ιεραρχία του χρόνου εκτείνεται σε L επίπεδα $\ell_i, 0 \leq i \leq L-1$ με το ℓ_0 να αντιστοιχεί στο πιο λεπτομερές και το ℓ_{L-1} στην ειδική τιμή ALL $(*)$. Ορίζουμε ότι ένα επίπεδο ℓ_k είναι υψηλότερα (χαμηλότερα) στην ιεραρχία από το ℓ_l και το συμβολίζουμε $\ell_k > \ell_l$ $(\ell_k < \ell_l)$ όταν και μόνο όταν $k > l$ $(k < l)$, π.χ. *Hour>Second*. Οι πλειάδες της βάσης μας έχουν τη μορφή: $\langle \text{tID}, T_{\ell_{L-1}}, \ldots, T_{\ell_0}, D_1, \ldots, D_{d-1}, \text{fact}_1, \ldots, \text{fact}_s \rangle$, όπου $T_{\ell_i}, 0 \leq i \leq L-1$ είναι η τιμή του ℓ_i επιπέδου του χρόνου, $D_j, 1 \leq j \leq d-1$ η τιμή της διάστασης j και $\text{fact}_m, 1 \leq m \leq s$ τα αριθμητικά ποσά ενδιαφέροντος.

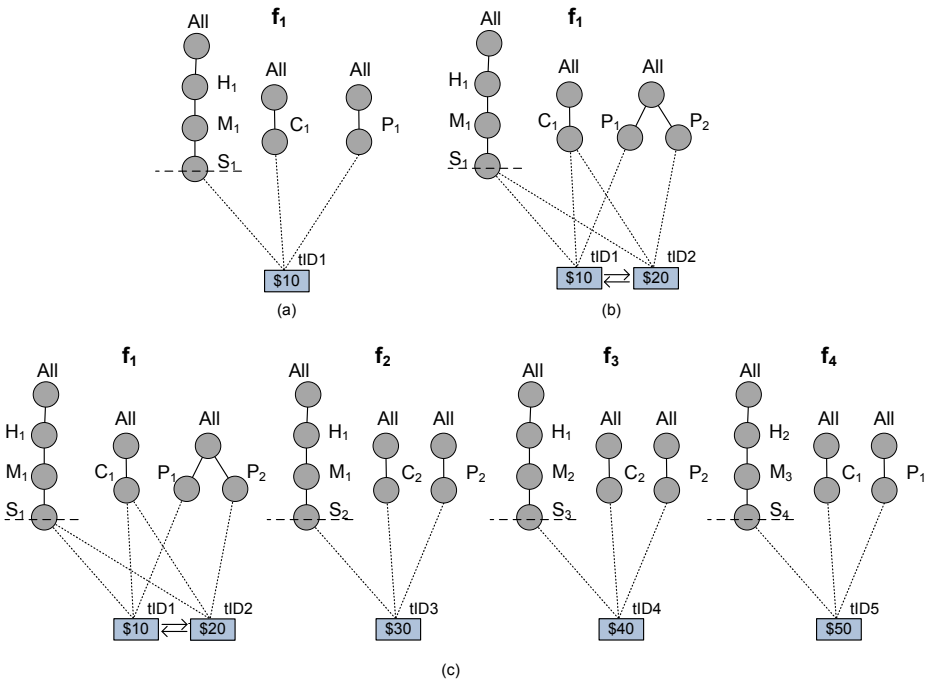
Σκοπός του συστήματος είναι η δεικτοδότηση των συνεχώς εισερχόμενων ενημερώσεων ώστε να επιλύονται ερωτήματα της μορφής: $q = \langle q_t, q_1, q_2, \ldots, q_{d-1} \rangle$, όπου το στοιχείο q_t του ερωτήματος μπορεί να πάρει τιμή από οποιοδήποτε ιεραρχικό επίπεδο του χρόνου και κάθε q_j λαμβάνει οποιαδήποτε τιμή της διάστασης j, συμπεριλαμβανομένης και της ειδικής τιμής $*$.

Ο Πίνακας 10 περιέχει δεδομένα σε τρεις διαστάσεις (`Time`, `Customer`, `Product`) και μία ποσότητα ενδιαφέροντος (`Sales`), καθώς επίσης και την ιεραρχία της διάστασης του χρόνου.

‡Δηλαδή το μέγιστο ποσό καθυστέρησης (*lag*) είναι περιορισμένο.

Πίνακας 10: *Πίνακας δεδομένων και μεταδεδομένα για το σενάριο χρήσης του συστήματος*

ιεραρχία του χρόνου	πίνακας δεδομένων						
	`tupleID`		`Time`		`Customer`	`Product`	`Sales`
Hour	1	H_1	M_1	S_1	C_1	P_1	\$10
↑	2	H_1	M_1	S_1	C_1	P_2	\$20
Minute	3	H_1	M_2	S_2	C_2	P_2	\$30
↑	4	H_1	M_2	S_3	C_2	P_2	\$40
Second	5	H_2	M_3	S_4	C_1	P_1	\$50



Σχήμα 15: *Οι δενδρικές δομές του T-HORAE μετά την εισαγωγή (a) της πρώτης, (b) της δεύτερης και (c) όλων των πλειάδων του Πίνακα 10*

Το Υποσύστημα T-HORAE

Το υποσύστημα T-HORAE είναι υπεύθυνο για την αποθήκευση και δεικτοδότηση των εισερχόμενων πλειάδων από πολλαπλές πηγές, παρέχοντας ενδιάμεση μνήμη (buffering) ανάμεσα σε αυτές και στον κύριο όγκο των ιστορικών δεδομένων που φυλάσσεται στο H-HORAE. Λόγω του απλού μηχανισμού εισαγωγής, μπορεί να χειριστεί αποδοτικά συχνές ενημερώσεις, ενώ ο προσαρμοστικός μηχανισμός δεικτοδότησης διασφαλίζει την αποτελεσματική επίλυση όλων των ερωτημάτων οποιουδήποτε επιπέδου λεπτομέρειας.

Το T-HORAE βασίζεται στο *HiPPIS*, το κατανεμημένο σύστημα για την αποθήκευση και αναζήτηση ιεραρχικών δεδομένων σε δίκτυα DHT που περιγράφηκε πρωτύτερα. Προτείνονται

πολλές σχεδιαστικές αλλαγές για την καλύτερη υποστήριξη χρονοσειρών: Το σχήμα δεικτοδότησης δίνει περισσότερη σημασία στη διάσταση του χρόνου, ώστε να διευκολύνει γρήγορες ενημερώσεις, ενώ η εσωτερική δομή αποθήκευσης των δεδομένων προσφέρει μεγαλύτερη ανεξαρτησία και ευελιξία για αποτελεσματική προσαρμογή στα διάφορα φορτία. Οι λειτουργίες του T-HORAE περιγράφονται λεπτομερώς παρακάτω.

Εισαγωγή/Ενημέρωση Δεδομένων

Κατά την αρχική εισαγωγή των δεδομένων ή κατά την άφιξη ομάδων από ενημερώσεις, καλείται η υπηρεσία *Data Upload*. Η δεικτοδότηση αναλαμβάνεται από την αρχιτεκτονική μονάδα (module) *T-Indexing*: Το ID κάθε πλειάδας προς εισαγωγή είναι το αποτέλεσμα της εφαρμογης της συνάρτησης κατακερματισμού στην τιμή T_{ℓ_0}. Η επιλογή αυτή αντικατοπτρίζει την υπόθεση ότι ο χρόνος είναι η σημαντικότερη διάσταση (παρών στην πλειονότητα των ερωτημάτων) και ότι τα ερωτήματα που αφορούν πρόσφατα γεγονότα (αυτά δηλαδή που φυλάσσονται κατά κύριο λόγο στο T-HORAE) αναζητούνται στην μεγαλύτερη λεπτομέρεια. Το DHT αναθέτει κάθε πλειάδα στον αντίστοιχο κόμβο.

Οι πλειάδες αποθηκεύονται εσωτερικά σε κάθε κόμβο με τη μορφη δενδρικών δομών. Κάθε τέτοια δομή f χαρακτηρίζεται από το *pivot* P_f, το οποίο ορίζει το επίπεδο δεικτοδότησης και μπορεί να είναι οποιοδήποτε από τα επίπεδα της ιεραρχίας του χρόνου εκτός από το $*$, για λόγους εξισορρόπησης του φόρτου (load balancing). Η τιμή στην οποία αντιστοιχεί το P_f ονομάζεται *pivot value* T_f. Στο T-HORAE, κάθε δενδρική δομή f μπορεί να επαναδεικτοδοτεί τα δεδομένα της σε διαφορετικό επίπεδο ανεξάρτητα από τις άλλες, δημιουργώντας καινούριες δενδρικές δομές με διαφορετικά P_f. Αυτή η διαδικασία εξηγείται λεπτομερώς στη συνέχεια.

Για τις ενημερώσεις, θα πρέπει να ανακαλυφθεί σε ποια δενδρική δομή θα επισυναφθεί η καθεμία ώστε να συμπεριληφθεί σε μελλοντικά ερωτήματα. Αυτό μεταφράζεται στην εύρεση της δενδρικής δομής με κοινό μονοπάτι από τη ρίζα μέχρι το pivot της και επιτυγχάνεται με διαδοχικές αναζητήσεις ξεκινώντας από το πιο λεπτομερές επίπεδο στο λιγότερο, μέχρι να ανακαλυφθεί το πρώτο κοινό στοιχείο. Αν δε βρεθεί κανένα, τότε δημιουργείται νέα δενδρική δομή με το ℓ_0 ως pivot. Η διαδικασία κοστίζει $O(L \log n)$ μηνύματα. Λαμβάνοντας υπόψιν ότι ο αριθμός των επιπέδων της ιεραρχίας του χρόνου είναι συνήθως περιορισμένος και ότι ερωτήματα για πρόσφατα δεδομένα τείνουν να είναι πιο λεπτομερή, μπορούμε να υποθέσουμε με ασφάλεια ότι το κόστος είναι κοντά σε μία απλή εισαγωγή σε DHT. Το Σχήμα 15 δείχνει ένα παράδειγμα εισαγωγής δεδομένων χρησιμοποιώντας τον Πίνακα 10.

Αναζήτηση Δεδομένων

Ερωτήματα που αφορούν το P_f μιας δενδρικής δομής f μπορούν να απαντηθούν σε λογαριθμικό αριθμό βημάτων. Ερωτήματα για τιμές του χρόνου που δεν είναι δεικτοδοτημένες δε μπορούν να απαντηθούν παρά μόνο αν προωθηθούν σε όλο το δίκτυο. Με τον τρόπο αυτό ανακαλύπτονται όλοι οι κόμβοι που έχουν πλειάδες που ανήκουν στην απάντηση, οι οποίες επιστρέφονται στον κόμβο που έθεσε το ερώτημα για την τελική συνάθροιση.

Θεωρώντας την αρχική κατάσταση του Σχήματος 15, το ερώτημα $\langle S_1, C_1, P_2 \rangle$ προωθείται στον κόμβο που είναι υπεύθυνος για το S_1 και ακολουθώντας τις συνδέσεις attribute προς attribute, η τιμή \$20 επιστρέφεται. Όταν το ερώτημα είναι το $\langle H_1, *, P_2 \rangle$, το σύστημα ανακαλύπτει ότι το H_1 δεν είναι δεικτοδοτημένο. Έτσι το ερώτημα πλημμυρίζει το δίκτυο και οι κόμβοι που περιέχουν τις δομές f_1, f_2 και f_3 απαντούν, με τον αρχικό κόμβο να κάνει την πρόσθεση και να καταλήγει στην τιμή \$90.

Επαναδεικτοδότηση

Η μονάδα *T-Workload Monitor* του T-HORAE ελέγχει την τρέχουσα τάση του φορτίου των ερωτημάτων διατηρώντας τοπικά στατιστικά για τη δημοφιλία κάθε επιπέδου της ιεραρχίας του χρόνου ανά δενδρική δομή. Αν τα ερωτήματα που αφορούν στο πιο δημοφιλές επίπεδο του f, $\ell_{\max}$, υπερβαίνουν αυτά που αφορούν το P_f κατά *threshold*, τότε ο κόμβος εξετάζει το ενδεχόμενο αναδιοργάνωσης των δεικτών του με βάση το $\ell_{\max}$.

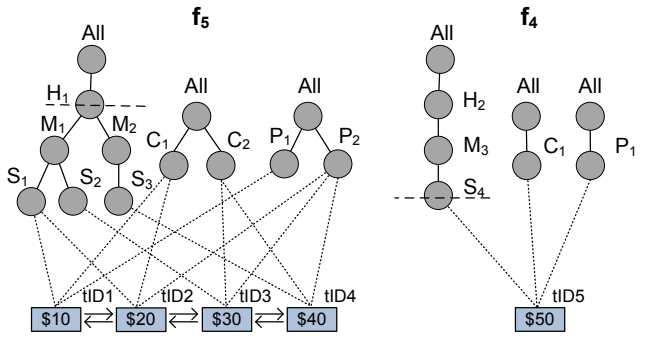
Αν $\ell_{\max} < P_f$, τότε ο κόμβος που φιλοξενεί το f μπορεί να πάρει αυτόνομα απόφαση για επαναδεικτοδότηση, καθώς διαθέτει όλα τα δεδομένα και τα αντίστοιχα στατιστικά των υποδένδρων του f στη διάσταση του χρόνου. Το *T-Indexing* εισάγει εκ νέου όλες τις πλειάδες του f σύμφωνα με το καινούριο pivot, το $\ell_{\max}$. Η διαδικασία αυτή χωρίζει το αρχικό δένδρο και δημιουργει τόσες καινούριες δενδρικές δομές όσες είναι και οι διαφορετικές τιμές του $\ell_{\max}$ που ανήκουν στο f, τις οποίες διαμοιράζει στο δίκτυο.

Στην αντίθετη περίπτωση, ο κόμβος δεν είναι σε θέση να πάρει μόνος του την απόφαση, καθώς η τιμή του $T_{\ell_{\max}}$ υπάρχει και σε άλλες δενδρικές δομές. Ένα μήνυμα *SendStats* σηματοδοτεί την αποστολή στατιστικών από όλους τους κόμβους που περιέχουν την τιμή $T_{\ell_{\max}}$ στις δενδρικές δομές τους. Μετά τη συλλογή των στατιστικών, ο κόμβος που ξεκίνησε τη διαδικασία ελέγχει αν εξακολουθεί να ισχύει η συνθήκη που ίσχυε τοπικά. Σε αυτή την περίπτωση, ένα μήνυμα *Reindex* αποστέλλεται σε όλους τους κόμβους που απάντησαν με στατιστικά, με αποτέλεσμα να συγχωνευθούν οι εμπλεκόμενες δενδρικές δομές σε μία, με το $\ell_{\max}$ ως pivot.

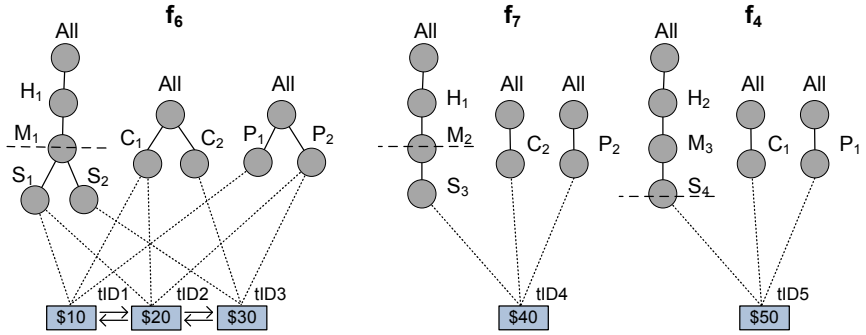
Στο παράδειγμά μας, αν το f_1 παρατηρήσει ότι το H_1 λαμβάνει περισσότερα ερωτήματα από το S_1 και τα στατιστικά των κόμβων που φιλοξενούν το f_2 και το f_3 επιβεβαιώνουν τα τοπικά ευρήματα, τότε η επαναδεικτοδότηση εφαρμόζεται σε όλους τους εμπλεκόμενους κόμβους (Σχήμα 16). Αν κάποια στιγμή τα στατιστικά του f_5 δείξουν ότι το `Minute` είναι το πιο δημοφιλές επίπεδο, ο κόμβος μπορεί μόνος του να αποφασίσει να μετακινηθεί στο επίπεδο αυτό (Σχήμα 17). Για να διασφαλιστεί η ορθότητα των λειτουργιών του συστήματος και να αποφευχθούν ταυτόχρονες επαναδεικτοδοτήσεις, ένα μήνυμα *Lock* πλημμυρίζει το δίκτυο για να σηματοδοτήσει ότι δε μπορεί να γίνει άλλη επαναδεικτοδότηση.

Μεταφορά Δεδομένων

Κατά τη διάρκεια της λειτουργίας του συστήματος, για τη διευκόλυνση μιας πιο ισχυρής ανάλυσης, ένα μέρος των δεδομένων που φυλάσσονται στο T-HORAE μεταφέρονται περιοδικά στο



Σχήμα 16: *Η κατανομή των δεδομένων μετά από επαναδεικτοδότηση στο* H_1

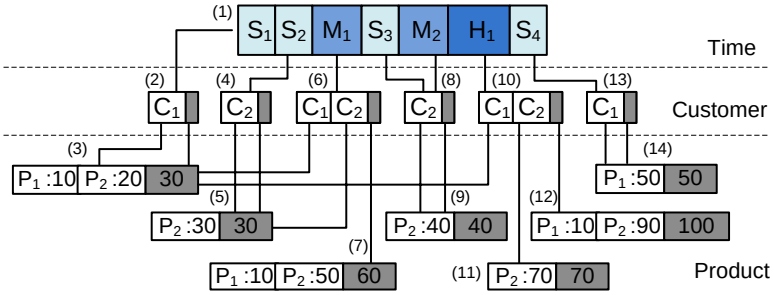


Σχήμα 17: *Η κατανομή των δεδομένων μετά από επαναδεικτοδότηση στο* M_1

H-HORAE μέσω της μονάδας *Offload*. Η μονάδα αυτή ρυθμίζει δυναμικά τη συχνότητα και το μέγεθος των δεδομένων που μεταφέρονται στο H-HORAE.

Νέα γεγονότα προς καταγραφή καταφθάνουν ασύγχρονα και πιθανόν από διαφορετικές πηγές. Το T-HORAE χειρίζεται τις ενημερώσεις αυτές ανά ομάδες (π.χ. ανά λεπτό). Ορίζουμε ως *καθυστέρηση (lag)* το μέγιστο χρόνο ανάμεσα στο πιο πρόσφατο και το πιο παλιό δεδομένο που ανήκει στο ίδιο σύνολο προς εισαγωγή. Πρόκειται για μια σημαντική παράμετρο, καθώς σχετίζεται με το μέγεθος της ενδιάμεσης μνήμης (buffer) που απαιτείται από το T-HORAE ώστε τα μεταφερόμενα δεδομένα να υποστούν σωστή επεξεργασία από την πιο αυστηρή μονάδα δεικτοδότησης του H-HORAE. Μεγάλες τιμές καθυστέρησης αναγκάζουν το T-HORAE να περιμένει περισσότερο πριν μεταφέρει τα δεδομένα του, ενώ μικρές τιμές μπορούν να πυροδοτήσουν μια πιο επιθετική στρατηγική. Τέλος, για να κρατηθεί σχετικά σταθερό το μέγεθος των δεδομένων στο T-HORAE, οι πλειάδες πρέπει να αναχωρούν από το υποσύστημα με ρυθμό ανάλογο του ρυθμού εισόδου. Έτσι, άλλη μια παράμετρος που επηρεάζει τη διαδικασία της μεταφοράς είναι ο ρυθμός ενημερώσεων λ_{upd}.

Επιλέγουμε να καλείται η διαδικασία περιοδικά, κάθε T_{off} χρονικό διάστημα, ενώ ο ακριβής διαχωρισμός των δεδομένων μεταξύ των υποσυστημάτων καθορίζεται από την παράμετρο W_{rem}: Μια τιμή $W_{\text{rem}} = 1$ h σημαίνει ότι το T-HORAE αποθηκεύει πλειάδες με χρονοσφραγίδες της



Σχήμα 18: *Ο πλήρης κύβος H-HORAE για τα δεδομένα του Πίνακα 10*

τελευταίας ώρας. Το σύστημά μας προσαρμόζει δυναμικά τις δύο παραμέτρους ανάλογα με τις εξής σχέσεις: $T_{\text{off}} = c_{\text{off}} \cdot \max\{\frac{1}{\lambda_{up}}, lag\}$ και $W_{\text{rem}} = c_{\text{rem}} \cdot \max\{\frac{1}{\lambda_{\text{upd}}}, lag\}$, όπου $c_{\text{rem}} > c_{\text{off}} > 1$.

Το Υποσύστημα H-HORAE

Το H-HORAE αποθηκεύει το μεγάλο όγκο των δεδομένων καθώς αυτά μεταφέρονται από το T-HORAE. Οργανώνει τη δεικτοδότηση του κύβου με τέτοιο τρόπο ώστε να ελαχιστοποιούνται τόσο το κόστος όσο και ο χρόνος των ενημερώσεων, ενώ η προσαρμοστική μέθοδος υλοποίησης που υιοθετεί συνοψίζει τα δεδομένα ανάλογα με τη μορφή του φορτίου ερωτημάτων. Τέλος, στηρίζεται σε μια στρατηγική αντιγραφής που λαμβάνει υπόψιν το φόρτο, διασφαλίζοντας τη διαθεσιμότητα και την ελαστικότητα του συστήματος με διαφανή τρόπο.

Ο σχεδιασμός του βασίζεται στο *Brown Dwarf*, που, όπως περιγράφηκε σε προηγούμενη ενότητα, κατανέμει μια αποδοτική κεντρική δομή κύβου δεδομένων στους κόμβους ενός αδόμητου δικτύου P2P. Το H-HORAE κάνει τις απαραίτητες αλλαγές στη δομή Brown Dwarf ώστε να διαχειρίζεται χρονικά δεδομένα. Οργανώνοντας τη δομή δεικτοδότησης κυρίως σύμφωνα με τη διάσταση του χρόνου, ευνοεί τις συχνές ενημερώσεις και την προσαρμογή στις τάσεις του φορτίου. Πρόκειται για νέα χαρακτηριστικά που δίνουν τη δυνατότητα στο H-HORAE να εξισορροπεί το κέρδος του ανάμεσα σε αποθηκευτικό χώρο και ακρίβεια με ρυθμιζόμενο τρόπο.

Εισαγωγή Δεδομένων

Ως εισαγωγή θεωρείται η αρχική δημιουργία του κύβου χρησιμοποιώντας τα ιστορικά δεδομένα του παρελθόντος. Αυτήν την αποστολή αναλαμβάνει η μονάδα *H-Indexing*. Η διάσταση του χρόνου επιλέγεται πρώτη στη σειρά φθίνοντος πληθάριθμου των διαστάσεων. Οι πλειάδες υφίστανται επεξεργασία μία προς μία, σύμφωνα με το πιο λεπτομερές επίπεδο ℓ_0. Μόλις όλες οι τιμές του ℓ_0 που ανήκουν στην ίδια τιμή του ℓ_1 έχουν υποβληθεί σε επεξεργασία, ο αλγόριθμος δημιουργεί ένα συναθροιστικό κελί για τη συγκεκριμένη τιμή του ℓ_1. Η ίδια διαδικασία ακολουθείται για όλα τα L επίπεδα της ιεραρχίας.

Για παράδειγμα, όταν εισάγεται η τρίτη πλειάδα του Πίνακα 10, το σύστημα συνειδητοποιεί ότι όλες οι πλειάδες που ανήκουν στο M_1 έχουν εισαχθεί, οπότε δημιουργεί ένα συναθροιστικό κελί και τον αντίστοιχο υπο-κύβο για το M_1. Ομοίως, με τη λήψη της τελευταίας πλειάδας, το σύστημα δημιουργεί τον υπο-κύβο του Y_1. Ο τελικός γράφος φαίνεται στο Σχήμα 18. Αξίζει να

σημειωθεί ότι το υψηλότερο επίπεδο συνάθροισης καθορίζεται από το υψηλότερο επίπεδο της ιεραρχίας και ότι δεν υπάρχει κελί ALL για την ιεραρχία του χρόνου.

Αναζήτηση Δεδομένων

Τα ερωτήματα απαντώνται ακολουθώντας το μονοπάτι τους κατά μήκος του συστήματος στοιχείο προς στοιχείο. Ένας κόμβος που ξεκινά ένα ερώτημα $q = \langle q_t, q_1 \ldots q_{d-1} \rangle$, το προωθεί στον κόμβο-ρίζα της κατανεμημένης δομής. Αν το q_t ανήκει στο επίπεδο ℓ_0, τότε το ερώτημα οδηγείται στον επόμενο κόμβο που πρέπει να επισκεφθεί. Η διαδικασία επαναλαμβάνεται έως ότου βρεθεί η απάντηση. Επειδή γειτονικοί κόμβοι στην κατανεμημένη δομή ανήκουν σε γειτονικούς κόμβους στο δίκτυο επικάλυψης, οποιοδήποτε σημειακό ή συναθροιστικό ερώτημα επιλύεται το πολύ μέσα σε d βήματα. Το ίδιο ισχύει και όταν το q_t ανήκει σε επίπεδο διάφορο του ℓ_0, αρκεί να υπάρχει η συναθροιστική τιμή του εν λόγω επιπέδου. Αν η τιμή αυτή δεν έχει δημιουργηθεί ακόμα, το αρχικό ερώτημα πρέπει να αντικατασταθεί από πολλαπλά ερωτήματα χαμηλότερου ιεραρχικού επιπέδου. Η διαδικασία αναζήτησης ενορχηστρώνεται από τη μονάδα *Query Xform*, μέσω της οποίας περνούν όλα τα ερωτήματα που αφορούν το H-HORAE.

Στο παράδειγμά μας, ένα ερώτημα για το $\langle M_1, ALL, P_1 \rangle$ ακολουθεί το μονοπάτι (1)→(6)→(7) και επιστρέφει \$10, ενώ το $\langle M_3, C_1, P_1 \rangle$ μεταφράζεται στο $\langle S_4, C_1, P_1 \rangle$, το οποίο αφού επισκεφθεί τους κόμβους (1), (13) και (14) φτάνει στην απάντηση \$50.

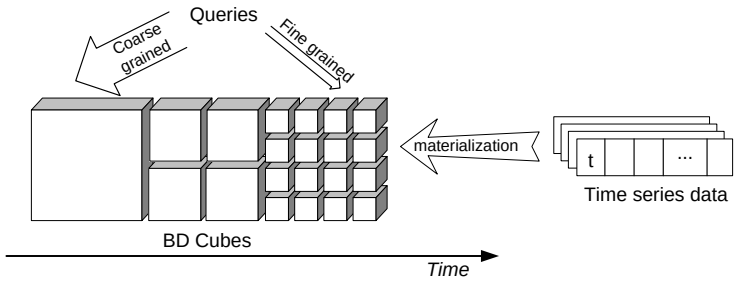
Ενημέρωση Δεδομένων

Η διαδικασία της ενημέρωσης μεταφράζεται στην εισαγωγή καινούριων πλειάδων στην υπάρχουσα δομή (read-only δεδομένα). Δεν είναι δυνατή εφαρμογή παλαιών ενημερώσεων στο H-HORAE. Αυτό διασφαλίζεται από το μηχανισμό μεταφοράς δεδομένων του T-HORAE, το οποίο συγκεντρώνει σύνολα πλειάδων, μέσα στα οποία οι ενημερώσεις ταξινομούνται πριν ενσωματωθούν στην υλοποιημένη δομή του H-HORAE.

Ακολουθώντας δικτυακές συνδέσεις, ανακαλύπτεται το μεγαλύτερο κοινό πρόθεμα της καινούριας πλειάδας με τις υπάρχουσες του συστήματος και οι υποκείμενοι κόμβοι ενημερώνονται αναδρομικά. Όπως συμβαίνει και στην εισαγωγή, οι πλειάδες εισάγονται αρχικά στο πιο λεπτομερές επίπεδο ℓ_0 και σταδιακά δημιουργούνται τα συναθροιστικά κελιά της δομής.

Ένα σημαντικό πλεονέκτημα του H-HORAE συγκριτικά με το *Brown Dwarf* είναι ότι μειώνει σημαντικά το κόστος των ενημερώσεων λόγω της εισαγωγής των πλειάδων με χρονική σειρά, σε συνδυασμό με την απουσία κελιού *ALL* στην πρώτη διάσταση. Η χρονική σειρά διασφαλίζει ότι το πρώτο πεδίο της καινούριας πλειάδας είτε θα δημιουργήσει νέο κελί στον κόμβο-ρίζα είτε θα συμπέσει με το τελευταίο του κελί (δεν επηρεάζεται κανένα συναθροιστικό κελί για τη διάσταση του χρόνου).

Η άφιξη της πλειάδας $\langle H_2, M_3, S_5, C_2, P_1$,\$60$\rangle$ στο παράδειγμά μας δημιουργεί ένα καινούριο κελί στον κόμβο (1) για το S_5 και δύο νέους κόμβους dwarf για τις εναπομείνασες τιμές, ενώ οι υπόλοιποι κόμβοι μένουν ανεπηρέαστοι. Αντιθέτως, στην αυθεντική δομή *Dwarf* θα προσπελάζονταν 12 κόμβοι.



Σχήμα 19: *Υλοποίηση με βάση το χρόνο στο H-HORAE*

Προσαρμοστική Υλοποίηση

Το προτεινόμενο σύστημα, όπως περιγράφηκε παραπάνω, ακολουθεί μια στατική στρατηγική για την υλοποίηση: Μια roll-up όψη στην ιεραρχία του χρόνου δημιουργείται μόλις τα απαιτούμενα δεδομένα είναι διαθέσιμα, χωρίς να καταστρέφονται οι drill-down όψεις. Αντ' αυτής, μπορούν να υιοθετηθούν πιο προσαρμοστικές προσεγγίσεις.

Υλοποίηση με βάση το χρόνο Σε εφαρμογές αποθηκών δεδομένων που αφορούν χρονοσειρές, συχνά τα πρόσφατα δεδομένα αναζητούνται σε μεγαλύτερη λεπτομέρεια από ότι τα ιστορικά. Με το σκεπτικό αυτό, δημιουργείται μια μέθοδος υλοποίησης όπου μια διεργασία daemon δημιουργεί περιοδικά roll-up όψεις και διαγράφει τις αντίστοιχες drill-down. Η περίοδος της διαδικασίας αυτής επιλέγεται λαμβάνοντας υπόψιν τα χαρακτηριστικά της εφαρμογής. Έτσι, η υλοποίηση ακολουθεί σταδιακά τη roll-up διαδρομή, όπως φαίνεται στο Σχήμα 19.

Τα χρονικά όρια πέρα από τα οποία δημιουργούνται roll-up όψεις και σβήνονται οι drill-down ορίζονται ως $(T_{\ell_1}, T_{\ell_2} \ldots T_{\ell_{N-1}})$. Αυτό πρακτικά σημαίνει ότι για τις εγγραφές που βρίσκονται αποθηκευμένες στο σύστημα για περισσότερο από T_{ℓ_i} το σύστημα κατασκευάζει και διατηρεί μόνο τη συναθροιστική όψη που ανήκει στο ℓ_i.

Υλοποίηση με βάση το φορτίο Ανάλογα με την εφαρμογή, δεν αναζητούνται όλα τα δεδομένα στο ίδιο επίπεδο λεπτομέρειας. Ο κόμβος-ρίζα διατηρεί στατιστικά για το φορτίο κάθε επιπέδου λεπτομέρειας μέσω της μονάδας *H-Workload Monitor*, ενώ η μονάδα *Materialization Decision* δημιουργεί ασύγχρονα συναθροιστικές όψεις για δημοφιλείς τιμές και διαγράφει όψεις που δεν προσπελάζονται συχνά. Το σύνολο των ορίων του φορτίου πάνω από το οποίο δημιουργείται μια roll-up όψη και το σύνολο των ορίων κάτω από το οποίο διαγράφεται μια drill-down όψη ορίζονται ως $(TMat_{\ell_1}, TMat_{\ell_2}, \ldots, TMat_{\ell_{L-1}})$ και $(TDel_{\ell_0}, TDel_{\ell_2}, \ldots, TDel_{\ell_{L-2}})$ αντίστοιχα και τίθενται ανάλογα με τις ανάγκες της εφαρμογής.

Στην περίπτωση της προσαρμοστικής υλοποίησης, υπάρχει ένα αντιστάθμισμα ανάμεσα στο μέγεθος και την πολυπλοκότητα του κύβου, που επηρεάζει τους χρόνους απόκρισης αλλά και

την ακρίβεια των αποτελεσμάτων. Η περιοδική δημιουργία και διαγραφή όψεων από τη μία κάνει οικονομία σε αποθηκευτικό χώρο, από την άλλη επιφέρει μεγάλο επικοινωνιακό κόστος και χαμηλή ακρίβεια αποτελεσμάτων όταν τα ερωτήματα δεν ακολουθούν την αναμενόμενη κατανομή. Επιπλέον, η διαγραφή όψεων μπορεί να οδηγήσει σε μη αναστρέψιμη απώλεια δεδομένων, με αποτέλεσμα λεπτομερή ερωτήματα για δεδομένα που είναι υλοποιημένα σε πιο υψηλά επίπεδα της ιεραρχίας να μπορούν να απαντηθούν μόνο προσεγγιστικά (με μεθόδους όπως οι [DGR07, Gar06, GKMS03] κλπ.).

Αντιγραφή

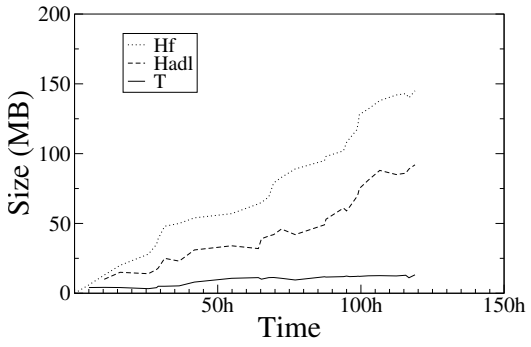
Η αντιγραφή είναι ένας σημαντικός μηχανισμός για την επίτευξη κλιμακώσιμης απόδοσης, ιδίως υπό μεγάλο φόρτο, και ανοχής σε σφάλματα. Το HORAE παρέχει δυνατότητα αντιγραφής και στα δύο του υποσυστήματα. Στο T-HORAE, η αντιγραφή αναλαμβάνεται από το υφιστάμενο DHT, ενώ το H-HORAE υιοθετεί μια μέθοδο αντιγραφής που προσαρμόζεται τόσο στην πόλωση του φορτίου όσο και στην κινητικότητα των κόμβων. Οι κόμβοι ελέγχουν περιοδικά τους γείτονες τους και αντικαθιστούν τα αντίγραφα που φιλοξενούνταν σε κόμβους που αστόχησαν, διατηρώντας το βαθμό πλεονασμού των δεδομένων πάνω από k. Επιπλέον, εποπτεύοντας το εισερχόμενο φορτίο ανά dwarf node, Το H-HORAE δημιουργεί επιπλέον αντίγραφα υπερφορτωμένων κόμβων και διαγράφει κόμβους με μικρό φόρτο μέσω του μηχανισμού της *διαστολής (expansion)* και *συστολής (shrink)*.

0.4.3 Πειραματική Αποτίμηση

Το σύστημα HORAE έχει υλοποιηθεί και αποτιμηθεί πειραματικά σε ένα πραγματικό σύστημα 16 κόμβων (Quad Core @ 2.0 GHz, 4GB RAM). Το Hive [TSJ+09] (version 0.5.0) έχει εγκατασταθεί στο ίδιο σύστημα δοκιμών για άμεση σύγκριση: 15 κόμβοι-εργάτες (με 2 Mappers και 2 Reducers χρησιμοποιώντας 512 MB RAM ο καθένας) και ένα μηχάνημα στο ρόλο του HDFS, MapReduce και HBase master. Για δίκαιη σύγκριση, δε χρησιμοποιείται αντιγραφή στο HDFS.

Χρησιμοποιώντας τον γεννήτορα του APB-1 benchmark [apb] παρήχθησαν τρία σύνολα δεδομένων με 4 διαστάσεις (A, B και C με πυκνότητες 0.1, 0.2 και 0.3 αντίστοιχα) και πληθαρίθμους 24, 9000, 900 και 9. Η ιεραρχία του χρόνου αποτελείται από τα επίπεδα `Month`<`Quarter`<`Year` και καλύπτει την περίοδο από τον Ιανουάριο του 1995 μέχρι τον Ιούνιο του 1996. Επίσης χρησιμοποιούνται πραγματικά δεδομένα από το πρόγραμμα αποτίμησης της ανίχνευσης εισβολών του DARPA (Intrusion Detection Evaluation Program) [dar98]. Περιλαμβάνει 1.1 εκατομμύριο εγγραφές, που συνελλέγησαν σε μια περίοδο 6 εβδομάδων και οργανώνονται σε 7 διαστάσεις. Η ιεραρχία του χρόνου είναι η εξής: `Second`<`Minute`<`Hour`<`Day`.

Τα πειράματα διεξήχθησαν με τη στατική αλλά και την προσαρμοστική έκδοση του *H-HORAE*. Στην πρώτη περίπτωση, στην οποία αναφερόμαστε στα πειράματα με το αναγνωριστικό H_f, η



Σχήμα 20: *Η κατανομή των δεδομένων του DARPA ως προς το χρόνο στο σύστημα HORAE*

Πίνακας 11: *Μετρήσεις για πραγματικά δεδομένα και δεδομένα benchmark*

		κατανομή ερωτ.(%)		χρόνος/ερώτ.(ms)			μέσος χρόνος/ερώτ.(ms)	
δεδομένα	#πλειάδων	T	H	T	H_f	H_{ad_l}	H_f	H_{ad_l}
APB-A	1.2M	10	90	502	14	17	56	56
APB-B	2.5M	9	91	510	17	18	59	60
APB-C	3.7M	9	91	514	20	24	62	64
DARPA	3.0M	62	38	607	52	87	357	374

υλοποίηση γίνεται σύγχρονα για όλα τα ιεραρχικά επίπεδα. Στη δεύτερη περίπτωση, παραθέτουμε πειράματα για τη μέθοδο υλοποίησης που βασίζεται στο φορτίο (H_{ad_l}).

Τα ερωτήματα που αφορούν τα δεδομένα του APB παράγονται από τον γεννήτορα του benchmark. Αυτά που αφορούν τα δεδομένα του DARPA δημιουργούνται ως εξής: Πρώτα επιλέγεται μια πλειάδα σύμφωνα με την κατανομή Zipf με $\theta = 1$, που ευνοεί τα πιο πρόσφατα δεδομένα. Έπειτα, το επίπεδο της ιεραρχίας του χρόνου επιλέγεται με την ίδια πολωμένη κατανομή. Έτσι, τα πρόσφατα δεδομένα αναζητούνται σε μεγαλύτερη λεπτομέρεια.

Μετά την εισαγωγή των πρώτων 100K πλειάδων στο T-HORAE, αρχίζουν να τίθενται ερωτήματα με $\lambda_q = 100 \frac{\text{ερωτηματα}}{\text{sec}}$. Ταυτόχρονα, πλειάδες συνεχίζουν να εισάγονται στο σύστημα σύμφωνα με τις χρονοσφραγίδες τους. Ο ρυθμός δεν είναι σταθερός, αλλά περιλαμβάνει και εξάρσεις. Το Σχήμα 20 παρουσιάζει το μέγεθος των αποθηκευμένων δεδομένων ως προς το χρόνο, τόσο στο T-HORAE όσο και στο H-HORAE, για τα δεδομένα του DARPA. Αν και το λ_{up} μεταβάλλεται στο χρόνο, το T-HORAE διατηρεί σχεδόν σταθερό μέγεθος δεδομένων χάρη στην προσαρμογή του T_{off} και του W_{rem}. Όσον αφορά το H-HORAE, το μέγεθός του αυξάνεται καθώς μεταφέρονται πλειάδες περιοδικά από το T-HORAE, με την αύξηση να είναι ομαλότερη για το H_{ad_l}.

Ο πίνακας 11 περιέχει μετρήσεις που αφορούν στην αναζήτηση. Καταρχάς, σημειώνουμε την κατανομή των ερωτημάτων ανάμεσα στα δύο υποσυστήματα. Για το DARPA, περισσότερο από το 60% των ερωτημάτων κατευθύνονται στο T-HORAE, αφού το φορτίο που δημιουργήθηκε

ευνοεί τα πρόσφατα δεδομένα. Για την περίπτωση του APB, όπου δεν έχουμε κανέναν έλεγχο στα φορτία που παράχθηκαν από το γεννήτορα, η πλειονότητα των ερωτημάτων επιλύονται από το H-HORAE. Ωστόσο παρατηρείται ότι σχεδόν το 99% των ερωτημάτων στοχεύουν στο πιο λεπτομερές επίπεδο της ιεραρχίας του χρόνου, οπότε η επίλυσή τους γίνεται γρηγορότερα από αυτήν του DARPA. Ενώ το T-HORAE είναι εν γένει πιο αργό από το H-HORAE, η διαφορά καλύπτεται από την ενσωμάτωση των υποσυστημάτων. Επιπλέον, καθώς τα δύο υποσυστήματα λειτουργούν ταυτόχρονα, επιτυγχάνεται παραλληλοποίηση που "κρύβει" ακόμα περισσότερο την επιβράδυνση του T-HORAE. Το Hive είναι τάξεις μεγέθους πιο αργό στην επίλυση ερωτημάτων, με χρόνο απόκρισης 20 sec κατά μέσο όρο.

0.4.4 Ανακεφαλαίωση

Περιγράφηκε το HORAE, ένα σύστημα αποθήκης δεδομένων ανεπτυγμένο σε μία αρχιτεκτονική shared-nothing και ειδικά σχεδιασμένο ώστε να χειρίζεται χρονικά δεδομένα που παράγονται με υψηλό ρυθμό. Το HORAE συνδυάζει την ταχύτητα και την ευρωστία ενός στρώματος DHT, που χρησιμοποιείται για την αποδοτική επεξεργασία ενημερώσεων, με τη δύναμη μιας δομής κύβου δεδομένων, κατανεμημένης σε αδόμητο δίκτυο P2P, που χρησιμοποιείται για το χειρισμό συναθροιστικών ερωτημάτων. Τα πλεονεκτήματά του περιλαμβάνουν υψηλούς ρυθμούς εξυπηρέτησης, online ενημερώσεις και ερωτήματα, ελαστικότητα των πόρων ανάλογα με τη ζήτηση και σημαντικό κέρδος σε αποθηκευτικό χώρο και σε χρόνο απόκρισης. Η πειραματική αποτίμηση του HORAE σε πραγματικό δίκτυο υπολογιστών δείχνει ότι είναι πιο γρήγορο από το Hive σε οποιοδήποτε συνδυασμό σημειακών και συναθροιστικών δεδομένων, με συγκρίσιμους χρόνους ενημερώσεων. Επιπλέον, το HORAE επιτρέπει προσαρμοστική εκχώρηση πόρων ανάλογα με τις απαιτήσεις της εφαρμογής, υλοποίηση που εξαρτάται από το φορτίο, ισορροπώντας ανάμεσα στο αποθηκευτικό κέρδος και στην ακρίβεια και ανοχή σε σημαντικό ποσοστό σφαλμάτων.

0.5 Συμπέρασμα

Στην εποχή της έκρηξης των δεδομένων, όπου σχεδόν κάθε ενέργεια καταγράφεται σε αρχεία, η ανάγκη να διαχειριστούμε αυτή την τεράστια ποσότητα πληροφορίας, να εξάγουμε από αυτήν χρήσιμα συμπεράσματα και να την εκμεταλλευτούμε για την ανίχνευση τάσεων, την κατανόηση φαινομένων και συμπεριφορών, την πρόβλεψη μελλοντικών γεγονότων και εν κατακλείδι τη λήψη αποφάσεων βασισμένων σε αδιάσειστα στοιχεία είναι πιο επιτακτική από ποτέ. Οι απαιτήσεις της σημερινής εποχής επιτάσσουν την ύπαρξη ενός συστήματος αποφάσεων πραγματικού χρόνου, που θα προσφέρει πρόσβαση σε μεγάλο όγκο δεδομένων και θα εξυπηρετεί ταχείς ρυθμούς ερωτήσεων χωρίς οποιαδήποτε επίπτωση στο χρόνο απόκρισης.

Μέχρι σήμερα, τα συμβατικά εργαλεία ανάλυσης δεδομένων είναι κεντρικά και παρόλο που απαντούν αποδοτικά πολύπλοκα ερωτήματα σε μεγάλο όγκο ιστορικών δεδομένων, αποτυγχάνουν να καλύψουν τις ολοένα αυξανόμενες ανάγκες για αποθηκευτικό χώρο και επεξεργαστική ισχύ. Τεχνικές από την περιοχή των κατανεμημένων συστημάτων προτάθηκαν πρόσφατα για να καλύψουν το κενό αυτό. Το Υπολογιστικό Νέφος, που αποτελεί την τελευταία τάση στο χώρο αυτό, έχει προσελκύσει το ενδιαφέρον επιστημόνων και επιχειρηματιών παγκοσμίως καθώς προσφέρει φαινομενικά άπειρους πόρους κατ' απαίτηση. Ωστόσο, η νέα κλάση μηχανών ανάλυσης που αναπτύχθηκαν πάνω στην πλατφόρμα αυτή, παρά την κλιμακωσιμότητα, διαθεσιμότητα και ανοχή σε σφάλματα που προσφέρουν, αδυνατούν να παρέχουν επεξεργασία ανά πλειάδα (per-tuple) σε πραγματικό χρόνο μιας και αποσκοπούν κυρίως σε μαζική επίλυση προβλημάτων.

Η έρευνά μου εστιάζει στην κατανομή και διαχείριση μεγάλου όγκου πολυδιάστατων δεδομένων που μπορούν να χρησιμοποιηθούν σε εφαρμογές αναλυτικής επεξεργασίας. Αρχικά προτάθηκαν δύο συστήματα, το *HiPPIS* και το *Brown Dwarf*, τα οποία στοχεύουν στην κάλυψη της ίδια ανάγκης: Τη δημιουργία μιας αποθήκης δεδομένων, ανεπτυγμένης σε υπολογιστές του εμπορίου, που θα είναι σε θέση να παρέχει πρόσβαση σε δεδομένα και να υποστηρίζει online επεξεργασία σε πραγματικό χρόνο. Για να εξασφαλιστεί η κλιμακωσιμότητα, η ανοχή σε σφάλματα και η δικαιοσύνη στη χρήση των πόρων επιστρατεύτηκαν τεχνικές από τον τομέα των P2P δικτύων.

Τόσο το *HiPPIS* όσο και το *Brown Dwarf* βασίζονται σε μια αρχιτεκτονική χωρίς κοινόχρηστους πόρους, επενδύοντας σε επεκτασιμότητα και διαθεσιμότητα σε χαμηλό κόστος. Φυσικοί ή εικονικοί πόροι μπορούν να εισέρχονται στο σύστημα εύκολα και με διαφανή τρόπο για να ανακουφίζουν από το φόρτο και να βοηθούν ώστε το σύστημα να ανταποκρίνεται στις αυξανόμενες απαιτήσεις σε αποθηκευτικό χώρο και υπολογιστική ισχύ. Η ανοχή σε σφάλματα είναι άλλη μία απαίτηση που ικανοποιούν και τα δύο συστήματα. Οι εργασίες ανάλυσης είναι ιδιαίτερα ευαίσθητες σε αστοχίες κόμβων λόγω του χρόνου που χρειάζονται για να ολοκληρωθούν. Η επανεκτέλεση του συνόλου των ερωτημάτων μιας εργασίας στην περίπτωση που ένας κόμβος αποτύχει δεν είναι βιώσιμη λύση, ιδίως όταν οι αποφάσεις πρέπει να ληφθούν άμεσα. Ζητήματα ανοχής σε σφάλματα αλλά και διαθεσιμότητας των δεδομένων αντιμετωπίζονται μέσω της έμφυτης μεθόδου αντιγραφής δεδομένων που προσφέρουν τα DHTs στην περίπτωση του *HiPPIS* και μέσω του ευφυή μηχανισμού αντιγραφής που προτείνουμε για το *Brown Dwarf*, που προσαρμόζεται τόσο στο εισερχόμενο φορτίο όσο και στην κινητικότητα των κόμβων.

Παρολαυτά, κάθε σύστημα προσεγγίζει τα ίδια θέματα από διαφορετική οπτική γωνία, θέτοντας άλλες προτεραιότητες:

Το *HiPPIS* εστιάζει στη διαχείριση ιεραρχικών δεδομένων, επιτρέποντας ερωτήματα σε διάφορα επίπεδα λεπτομέρειας μέσω λειτουργιών roll-up και drill-down. Αυτό το γεγονός καθιστά το *HiPPIS* κατάλληλο για σενάρια όπου είναι απαραίτητη μια πιο λεπτομερής αναπαράσταση

των δεδομένων. Η απλότητα της δομής του *HiPPIS* επιτρέπει την ταχεία εισαγωγή του αρχικού πίνακα δεδομένων χωρίς καμία προεπεξεργασία. Ωστόσο, επειδή καμία υλοποίηση του κύβου δε γίνεται εκ των προτέρων, τα ερωτήματα group-by απαιτούν περαιτέρω επεξεργασία μετά τη συλλογή των πλειάδων που σχετίζονται με αυτά. Οι ενημερώσεις είναι τόσο γρήγορες και απλές όσο και οι εισαγωγές, επιφέροντας επιβάρυνση στην επικοινωνία που εξαρτάται από το επίπεδο συνέπειας που απαιτεί η εκάστοτε εφαρμογή. Έτσι, σε περιπτώσεις όπου τα δεδομένα ενημερώνονται συνεχώς και μάλιστα με υψηλό ρυθμό, το *HiPPIS* καταφέρνει να αντεπεξέλθει με αποδοτικό τρόπο.

Το *Brown Dwarf* κατανέμει μια ευρέως διαδεδομένη δομή που υλοποιεί έναν κύβο δεδομένων, επιτυγχάνοντας σε πολλές περιπτώσεις ένα σημαντικό βαθμό συμπίεσης. Πληρώνοντας το κόστος της προεπεξεργασίας άπαξ, επιλύει τα συναθροιστικά ερωτήματα τόσο εύκολα και φυσικά όσο και τα σημειακά. Όμως οι συναρτήσεις συνάθροισης πρέπει να έχουν καθοριστεί εκ των προτέρων. Επίσης, οι ενημερώσεις στο *Brown Dwarf* είναι αρκετά δαπανηρές, καθώς η εισαγωγή μιας και μόνο καινούριας πλειάδας πυροδοτεί πολλαπλές αλλαγές σε συναθροιστικές τιμές σε όλη τη δομή. Έτσι, το σύστημα *Brown Dwarf* είναι πιο αποδοτικό σε περιβάλλοντα όπου ο ρυθμός των ενημερώσεων δεν είναι τόσο υψηλός σε σχέση με το ρυθμό των ερωτημάτων ή όταν οι ενημερώσεις μπορούν να εφαρμοστούν σε ομάδες.

Υπάρχει ξεκάθαρα μια αντιστάθμιση: Μη υλοποιημένα δεδομένα καταλαμβάνουν λιγότερο χώρο και προσφέρουν ευκολότερη λειτουργία ενημέρωσης με κόστος την αυξημένα επεξεργασία στην πλευρά του πελάτη. Αντιθέτως, όσο πιο επεξεργασμένα είναι τα δεδομένα τόσο περισσότερο χώρο καταλαμβάνουν αλλά και τόσο λιγότερη εκ των υστέρων επεξεργασία χρειάζονται Επομένως, από τη μία πλευρά το *HiPPIS* προσφέρει γρήγορη εισαγωγή και ενημέρωση των δεδομένων που αναπαριστώνται με περισσότερη λεπτομέρεια μέσω της χρήσης εννοιολογικών ιεραρχιών, αλλά επιδεικνύει πιο χρονοβόρα επεξεργασία ερωτημάτων. Από την άλλη πλευρά, το *Brown Dwarf* απαντά αποδοτικά όλα τα σημειακά αλλά και συναθροιστικά ερωτήματα σε φραγμένο αριθμό βημάτων, αλλά αντιμετωπίζει πιο δαπανηρές ενημερώσεις λόγω της υλοποίησης του κύβου.

Έχοντας διακρίνει τις περιπτώσεις στις οποίες ταιριάζει καλύτερα καθένα από τα προτεινόμενα συστήματα, το σύστημα *HORAE* γεφυρώνει το χάσμα και προσφέρει μια ολοκληρωμένη λύση που συνδυάζει τα πλεονεκτήματα του καθενός για τη διαχείριση χρονοσειρών: Μια ισχυρή μηχανή δεικτοδότησης για τεράστιο όγκο δεδομένων τόσο ιστορικών όσο και πραγματικού χρόνου με μια αρχιτεκτονική χωρίς κοινόχρηστους πόρους που εξασφαλίζει κλιμακωσιμότητα και διαθεσιμότητα σε χαμηλό κόστος. Ένα υποσύστημα βασισμένο σε DHT που μοιάζει με τη δομή του *HiPPIS* χειρίζεται τις ενημερώσεις που καταφθάνουν με υψηλό ρυθμό και τις ερωτήσεις που αφορούν τα πιο πρόσφατα δεδομένα. Τον κύριο όγκο των δεδομένων αναλαμβάνει ένα υποσύστημα με τη δομή του *Brown Dwarf* που υλοποιεί και αντιγράφει κατ' απαίτηση. Τα δύο

συστατικά αυτά ενσωματώνονται για να προσφέρουν τα πλεονεκτήματα της ισχυρής επεξεργασίας δεδομένων σε συνδυασμό με κλιμακωσιμότητα και ελαστικότητα των πόρων.

Η διατριβή μέχρι τώρα έχει ασχοληθεί με την διαχείριση δομημένων δεδομένων. Μέρος των μελλοντικών ερευνητικών μου στόχων είναι η χαλάρωση των περιορισμών του συστήματος για ένα καθολικό σχήμα. Ο αποδοτικός χειρισμός ημι-δομημένων δεδομένων, όπως για παράδειγμα XML αρχείων, αλλά και η υποστήριξη δυναμικών αλλαγών στα σχήματα δομημένων δεδομένων ενέχουν μεγάλες ερευνητικές προκλήσεις.

Επιπλέον, στα μελλοντικά σχέδια βρίσκεται η έρευνα για το πώς οι τεχνολογίες MapReduce και τα συστήματα που προτάθηκαν στη διατριβή αυτή μπορούν να αλληλοσυμπληρωθούν στον τομέα της ευρείας κλίμακας ανάλυσης δεδομένων. Πράγματι, όπως υποδεικνύει και η πειραματική αποτίμηση των προτεινόμενων συστημάτων, οι πλατφόρμες ανάλυσης MapReduce είναι ιδιαίτερα αποδοτικές σε ETL διεργασίες, όμως αποτυγχάνουν σε αυξητική επεξεργασία και διαδραστικότητα, οι οποίες είναι επιθυμητές σε εφαρμογές όπως ο έλεγχος, η εξυπηρέτηση πελατών, η διόρθωση σφαλμάτων κλπ. Από την άλλη, όλα τα συστήματα που προτάθηκαν στη διατριβή αυτή προσφέρουν επεξεργασία ανά πλειάδα. Αυτή η παρατήρηση καταδεικνύει την ανάγκη για συνεργασία μεταξύ των δύο κλάσεων συστημάτων που θα επιτρέπει την εκμετάλλευση των πλεονεκτημάτων τους.

Σε εξέλιξη βρίσκεται έρευνα που αφορά στη χρήση του συστήματος *HiPPIS* για την διατήρηση της ανωνυμίας οριζόντια κατανεμημένων σχεσιακών δεδομένων (βλέπε Παράρτημα Α). Η ιδιωτικότητα κατανεμημένων δεδομένων είναι ιδιαίτερα σημαντική, μιας και η ανάλυσή τους σε συνδυασμό με άλλα σχετικά δεδομένα που παράγονται συνήθως από διαφορετικές πηγές μπορεί να αποκαλύψει προσωπικές και ευαίσθητες πληροφορίες. Για την αποκατάσταση τέτοιων λαθών χρησιμοποιείται συχνά η γενίκευση τομέα (domain generalization): Η αντιστοίχηση τιμών ενός attribute σε τιμές που ανήκουν σε έναν πιο γενικό τομέα ανεβαίνοντας ιεραρχικά επίπεδα μπορεί να βοηθήσει στο να μη μπορεί κανείς να εξάγει ευαίσθητη πληροφορία για ένα συγκεκριμένο άτομο. Το *HiPPIS*, που εγγενώς χειρίζεται ιεραρχικά και κατανεμημένα δεδομένα βελτιώνεται στον μηχανισμό δεικτοδότησής του με τέτοιον τρόπο ώστε να διατηρεί την ανωνυμία τους υπό το καθεστώς συνεχών ενημερώσεων. Η επιδίωξή μας είναι η περαιτέρω διερεύνηση τομέων και εφαρμογών όπου τα προταθέντα συστήματα θα μπορέσουν να χρησιμοποιηθούν επιτυχώς.

CHAPTER 1

Introduction

1.1 Motivation

In the last decade we have witnessed an enormous data explosion, which is still in progress. As Information Technology (IT) becomes ever more prevalent in nearly every aspect of our lives, the amount of data generated and stored continues to grow at an astounding rate: According to IBM [CCMR06], worldwide data volumes are currently doubling every two years, having already crossed the zettabyte limit [IDC10]. Scientists and computer engineers have coined a new term for the phenomenon: *big data**. This growth is attributed to the evolution of data itself as well as its production and manipulation processes.

This fact has become apparent even in our everyday lives, where the emergence of new technologies and mainly the appearance of Web 2.0 allow users to do more than just retrieve information: They can be content producers besides content consumers. As hardware (from storage devices to high-tech digital cameras) becomes a cheaper commodity by the day, the amount of data an average person produces in the form of e-mails, images, video albums, personal records, etc., is rapidly increasing. Moreover, with domestic Internet connections gaining ground, connection speeds being on the rise and web services becoming more and more accessible, "large"

*"Big data are datasets that grow so large that they become awkward to work with using on-hand database management tools. [. . .] Though a moving target, current limits are on the order of terabytes, exabytes and zettabytes of data", Wikipedia.

data objects such as photos, audio and video files can easily be uploaded and shared over the Internet. Social network statistics prove this tendency. Facebook [fac] for instance, with over 500 million active users is responsible for the upload of 20 million videos and over 2 billion photos per month [fac11] while Twitter [twia] counts more than 160 million users producing over 90 million tweets per day [twi10].

Similarly, several data-intensive applications in the scientific field, such as bioinformatics, physics or astronomy, rely, to a great extent, on the analysis of data produced at a tremendous rate and volume by geographically disperse scientific devices such as sensors, satellites, digital cameras, etc. For example, the Large Hadron Collider (LHC) project at CERN [lhc] generates tens of terabytes of raw data per day that have to be transferred to academic institutions around the world, in seek of the Higgs boson [atl, ali]. Another example is the Laser Interferometer Gravitational Wave Observatory (LIGO) [lig], a multi-site research facility whose objective is the detection of gravitational waves, producing 1 TB of data per day. Due to the growing size of such data sets, management platforms are needed to ensure fast and reliable access to users in remote and distributed locations.

In the business domain, organizations are investing in more sophisticated business intelligence and analytics in order to base decisions on solid and reliable management information. One of the most popular tools for data analysis is *Data Warehousing*. In data warehousing, vast amounts of historical data along with data from multiple operational databases (in the form of multidimensional cubes) need to be stored and analyzed in order to identify behavioral patterns and discover useful associations, pushing the size of data warehouses over the petabyte barrier [Mon]. Market globalization, business process automation, the growing use of sensors and other data-producing devices, along with the increasing affordability of hardware have contributed to this continuous trend [Sie08]. Indeed, recent research [Gro10] has indicated that large organizations are experiencing an average 32% annual growth in data volume.

Besides the well-documented need for offline analytics, the requirement to immediately detect interesting trends is ever-growing [Kno09, AFG+09], rendering real-time analytics a necessity [dat10]. For instance, Denial of Service (DoS) attacks or intrusions should be detected by Internet Service Providers (ISPs) the moment they occur, so that appropriate measures can be taken to restore functionality with minimal service unavailability [SJ06]. Another example of the power of real-time analytics is their use in the forecasting of a hurricane's path, intensity and wind field hours or days in advance [PHAML98], sparing valuable time for evacuation and preparation that can save lives and properties. In everyday business environments, real-time analytics can provide up-to-the-minute information about an enterprise's customers so that better and faster business decisions can be made – perhaps even within the time span of a customer interaction.

Real-time web analytics is a category per se, with a plethora of commercial products (e.g., Clicky [cli], Woopra [woo], Chartbeat [cha], etc.) claiming to offer monitoring of web page visitors as well as tracking of social media shares in real time, unlike Google Analytics [goob], their free adversary. Moreover, analyzing the so-called *real-time Web*, i.e., content pulled by Twitter, blogs and news websites within minutes of its generation, valuable information about behavior and sentiment can be derived: A public figure can estimate its reputation and popularity (Twittercounter [twib]), market researchers can measure the impact of a product's launch (WebTrends [webb], WebAbacus [weba], etc.), even political candidates can derive the orientation of the electoral body and predict the winner of an election [DS10, MM10]. In recognition of the power of real-time analytics over a system like Twitter, which is real-time itself, the Twitter analytics team has recently (February 2011) presented Rainbird [Wei11], a high-volume analytics system that scales horizontally in addition to being real-time.

As a result, data processing applications that extract, store and process useful information in near real-time are taking center stage in the enterprise Information System (IS) infrastructure. In such applications, data are usually determined by a temporal aspect (e.g., time-stamps of router data or dates of purchases) presented at different levels of granularity through the use of *concept hierarchies* (e.g., `Day`$<$`Month`$<$`Quarter`$<$`Year`). Thousands or millions of such records are produced per second and modern systems are expected to be able to both incorporate and process them. It is clear that the efficient and effective management of this enormous volume of data is of utmost importance, in order to make the most out of the available information and to ensure that the decisions are made and actions are taken based on accurate, complete and up-to-date facts.

Another important aspect relates to the fact that information environments themselves are distributed. Business groups consist of multiple companies around the world, which, although operating autonomously, still need to provide the headquarters with summarized information for decision making. Moreover, it is a fact that IT is moving towards environments, where resources are provided as services over the Internet and business applications are delivered online and accessed from a web browser [Eco10]. *Cloud Computing* is the most recent such paradigm that offers resources as a service and has drawn attention from the research as well as business community, with major IT companies (e.g., Microsoft [mic], Amazon [ec2], Google [gap], etc.) getting involved in the provision of infrastructure, on-demand analytics solution vendors (e.g., Vertica [ver], Terradata [ter], GoodData [gooa], etc.) supporting it and numerous enterprises using it. It is not surprising that major content providers and social network sites have already moved towards cloud-based solutions: Twitter [twia], Digg [dig], Reddit [red] and many others use Apache Cassandra [cas] to store their large, active data sets, the message infrastructure of Facebook [fac] relies on HBase [hba], while LinkedIn [lin] has launched its own project, Voldemort [vol], to handle the amount of stored data as well as the rate of operations. It is estimated

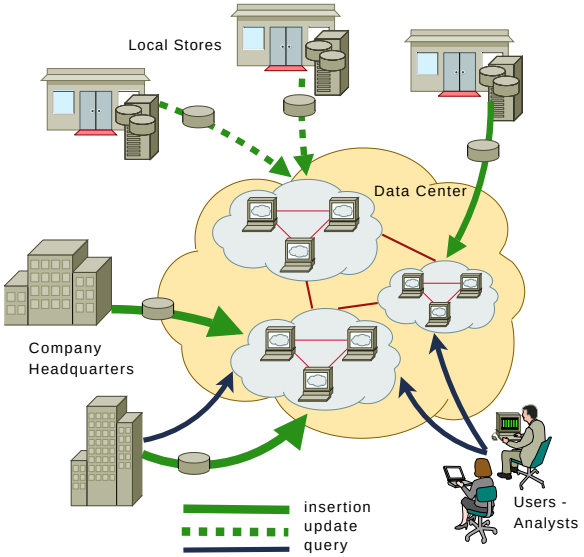


Figure 1.1: *Motivating scenario of distributing a datawarehouse*

that at least 15% of the Digital Universe by 2020 will be managed or stored in the Cloud and at least a third of all this data will pass through it at some point in their life cycle [IDC10]. In such environments, the software as well as the data are stored on servers, often geographically dispersed, and therefore their manipulation requires sophisticated, distributed techniques.

The above requirements imply the need for an always-on, real-time data access and support system for concurrent processing of queries without significant deterioration in response times. As a motivating scenario, let us consider a business establishment that maintains records of its operations. These records could well be security, network or system event logs. The search and analysis of such data constitutes an essential part of managing, securing and auditing the usage of this company's technology infrastructure. Instead of creating a centralized data warehouse on-site with a large upfront and maintenance cost, the management chooses to distribute data and computation to possibly multiple location-transparent facilities of commodity nodes and access it more easily and ubiquitously. In this manner, the establishment significantly lowers maintenance and hardware costs while enjoying a scalable, real-time decision support system. Figure 1.1 depicts this scenario, where multiple establishments of a business insert, update and query such a distributed warehouse.

1.2 Relative solutions

Up till recently, data management has mainly been translated to mediating access to centralized, application-specific databases. However, such conventional practices are unable to keep

pace with the ever growing needs of today's data-intensive applications. Particularly in the field of analytics, existing systems inadvertently fail to achieve both powerful data processing and high-rate updates: Conventional data warehousing solutions (e.g., [LPZ03, SDRK02, WLFY02]), while highly efficient on complex queries upon large volumes of historical data, present a strictly centralized and offline approach in terms of data location and processing: Views are usually calculated on a daily or weekly basis, after the operational data have been transferred from various locations and, surprisingly, this practice is still considered to be state-of-the-art.

Distributed variations (e.g., [AAC+08, ABJ+03]) essentially just interconnect conventional warehouses, while maintaining the main functionality, i.e., aggregation, update and querying, centralized. On the other hand, there has been considerable work in sharing relational data using both structured (i.e., *Distributed Hash Tables* or *DHTs* for short) and unstructured (i.e., Gnutella-style) Peer-to-Peer (P2P) overlays, combining the advantages of a distributed and resilient solution with the performance of storing large volumes of data in database systems. In Peer Database Management Systems (e.g., [KTSR09, HHL+03, NOTZ03]) peers maintain databases with different schemas and communicate with each other in a distributed, fault-tolerant manner, using query reformulation in order to translate a query form one schema to another. Nevertheless, no special consideration has been given to multidimensional data supporting hierarchies nor to temporal data and, to date, Peer Databases that rely on DHT functionality are unable to directly support queries on multiple dimension hierarchies. Moreover, the slow query reformulation process renders them unsuitable for real-time application scenarios such as the ones involving the management of temporal data.

A new class of analytics engines (e.g., [ABPA+09, TSJ+09]) that leverage the recent innovation in the industry around large-scale data management has emerged to fill this gap. These engines are deployed on shared-nothing, commodity hardware architectures, covering the newly added requirement for scalability, robustness and availability at low cost. Yet, even the new platforms pose some limitations: Based upon the MapReduce programming model, they mostly target batch-mode analytics jobs rather than real-time, "per-tuple" processing. This drawback has been recently identified by Google and alternative approaches for incremental processing and interactive response times have been proposed [DP10, MGL+10].

To conclude with relative solutions, parallel databases (e.g., [ora, ter]) offer great efficiency at the cost of elasticity and robustness in failures [PPR+09]. Although scalability is theoretically promised, actual systems operate on relatively small clusters of 100 or less homogeneous nodes. Overcoming the cost of purchase and support for such systems, the difficulty of their installation and proper configuration prevents the automatic and transparent expansion of the system to handle increased demand.

1.3 Contribution

This thesis deals with the issue of storing, indexing and querying multidimensional data used for analytical processing in large scale distributed systems. It explores ways to create an always-on, real-time data access and support system, applying techniques from the field of distributed data management and data-warehousing in order to disseminate, query and update high volumes of multidimensional data. The goal is to maintain the best of both worlds: Powerful indexing/analytics engine for immense volumes of data both over historical and real-time incoming updates and a shared-nothing architecture that ensures scalability and availability at low cost. Geographically spanned users, without the use of any proprietary tool, can share information that arrives from distributed locations at a high rate and query it in different levels of granularity.

The research process towards this goal starts with an attempt, the first to the best of our knowledge, to support concept hierarchies in DHTs, in order to store historical data in various levels of granularity. The resulting system, *HiPPIS*, greatly simplifies the insertion and update operations due to the lack of data pre-processing. However, it increases the post-processing on the client side.

In an attempt to include an a priori consideration for group-by queries, as well as to explicitly deal with the query performance versus variable data availability or load skew, a well known, highly effective centralized structure, the *Dwarf* [SDRK02], is distributed over a network of interconnected commodity nodes on-the-fly, reducing cube creation and querying times by enforcing parallelization.

To improve the processing of data determined by a temporal aspect (henceforth termed as *time series data* †), the special requirements of this specific data type are identified and the appropriate modifications in the proposed systems are applied.

By reviewing and evaluating the proposed systems, their strengths and weaknesses are revealed. There exists a trade-off between the ease of operations in one hand and the storage consumption as well as the efficiency of query resolution on the other. As a conclusion to my research, the best of the two worlds are combined, creating *HORAE*, a hybrid solution for data storage and processing.

Hence, the work of this dissertation can be divided in 3 major parts:

The Hierarchical Peer-to-Peer Indexing System (HiPPIS)

HiPPIS [DATK08, DTK08, DTK11] is a distributed system designed to efficiently store, query and update multidimensional data organized into concept hierarchies and dispersed over a network. *HiPPIS* employs an adaptive scheme that automatically adjusts the level of indexing according to the granularity of the incoming queries, without assuming any

†We will be using the terms *temporal data* and *time series data* interchangeably to refer to the data determined by a temporal aspect.

prior knowledge of the workload. Efficient roll-up and drill-down operations take place in order to maximize the performance by minimizing query flooding. Updates are performed online, with minimal communication overhead, depending on the level of consistency needed. Extensive experimental evaluations show that, on top of the advantages that a distributed storage offers, *HiPPIS* answers the large majority of incoming queries, both point and aggregate ones, without flooding the network and without causing significant storage or load imbalance. *HiPPIS* proves to be especially efficient in cases of skewed workloads, even when these change dynamically with time. At the same time, it manages to preserve the hierarchical nature of data. To the best of our knowledge, this is the first attempt towards the support of concept hierarchies in DHTs.

The Brown Dwarf System

Brown Dwarf [DTK10a,DTK10c,DTK10b] is a data analytics system that distributes multidimensional data over commodity network nodes, without the use of any proprietary tool. Brown Dwarf distributes a centralized indexing structure, the Dwarf [SDRK02], among peers on-the-fly, reducing cube creation and query times by enforcing parallelization. Analytical queries as well as updates are naturally performed online through cooperating nodes that form an unstructured P2P overlay. Updates are also performed online, eliminating the usually costly over-night process. *Elasticity* and *content availability* are indispensable features: The system employs an adaptive replication scheme that adjusts to sudden shifts in workload skew as well as network churn by expanding or shrinking the units of the distributed data structure. These characteristics along with the cost-effectiveness both over the required hardware and software components render the *Brown Dwarf* an ideal candidate application to be deployed in large scale distributed environments such as the Cloud [AFG+09]. Experimental evaluation on an actual testbed has shown that it manages to accelerate cube creation up to 5 times and querying up to several tens of times compared to the centralized solution by exploiting the capabilities of the available network nodes working in parallel. It also manages to quickly adapt even after sudden bursts in load and remains unaffected with a considerable fraction of frequent node failures. These advantages are even more apparent for dense and skewed data cubes and workloads.

The HORAE System

HORAE is a hybrid system focused on the management of time series data in a fully distributed manner. Time series data is an important class of data that typically contain a time attribute, such as the date in a stream of sales data or the time of a credit card purchase. Time series analysis can extract meaningful statistics and other characteristics of data that have a natural temporal ordering and is widely used in forecasting. After examining the behavior of *HiPPIS* as well as *Brown Dwarf* with respect to time series data, a complete

system that employs a hybrid solution for data storage and processing is designed: Recent data, which are bound to be updated rapidly and queried in finer granularity, are stored in a DHT, *HiPPIS*-like system, that enables fast insertion times and multidimensional indexing. The large bulk of the data is handled through *Brown Dwarf* cubes that adaptively materialize and replicate according to demand.

The two components seamlessly integrate to offer the advantages of powerful aggregate data processing along with scalability and elasticity of commodity resources. The prototype implementation over an actual testbed proves that HORAE is able to efficiently handle large rates of both updates and queries, tolerate high failure ratios and expand or contract its resources according to demand. A direct comparison with a state-of-the-art warehousing solution demonstrates HORAE's advantages in both performance and elasticity under variable workloads: HORAE accelerates query resolution by orders of magnitude, manages to quickly adapt to the incoming load and tolerates a considerable fraction of frequent node failures.

The contribution of this thesis as a whole is summarized in the following:

- It studies and identifies the requirements of a large scale analytics platform.
- It proposes a system that provides an efficient and cost-effective way to handle concept hierarchies in DHTs. Taking into account user preferences and sensing potential overall tendencies, the proposed system allows reorganization of the indexing structure in favor of resolving queries for the most popular data, preserving at the same time the useful hierarchy-specific information that hashing destroys.To our knowledge, this is the first attempt towards this direction.
- It creates a fully distributed data-warehouse-like system that offers indexing, query processing and update operations for data cubes over a distributed environment. The cube is created with just one pass over the data, while updates are processed online. Commodity PCs can participate in this distributed data store, while users need no proprietary tool to access it. Giving special consideration to elasticity and fault tolerance, a robust and efficient adaptive replication scheme is designed, perceptive both to workload skew as well as node churn using only local load measurements and overlay knowledge.
- It specifically deals with the special requirements of time series data, produced at a high rate from distributed sources, presenting a complete system that stores, indexes and processes them. Combining the advantages of both structured and unstructured P2P overlays, the system enables fast insertions and efficient aggregate query resolutions.

- All of the proposed systems have been implemented and deployed either on a well known simulator or on an actual LAN testbed of commodity PCs. Their extensive experimental evaluation under a variety of datasets, workload distributions and network setups demonstrates their ability to efficiently handle large rates of both updates and queries, tolerate high failure ratios and adjust their indexing structure and their available resources according to demand. A direct comparison with centralized as well as distributed state-of-the-art warehousing solutions proves the advantages of the proposed systems in both performance and elasticity: Query resolution is accelerated, updates are performed online, the load is handled efficiently even after sudden bursts and the functionality remains unaffected with a considerable fraction of frequent node failures.

1.4 Outline

This dissertation is organized as follows:

Chapter 2 briefly introduces some basic notions, upon which the work of this thesis is based, for self-containment reasons. On a higher abstraction level, references are made to the basic structures and common practices of data warehousing. Moving towards lower levels, the chapter presents Distributed System platforms, where our solutions can be applied and concludes with the description of the underlying overlays, structure- and operation-wise, that were utilized in our work.

Chapter 3 presents our work in the field of distributed management of multidimensional, hierarchical data, describing *HiPPIS*, discussing its requirements together with protocol enhancements and providing a cost/benefit analysis as well as a thorough evaluation of it.

Chapter 4, describes the *Brown Dwarf* system, which aims to serve as a distributed data-warehouse-like system. Besides the evolution from the centralized Dwarf structure to the fully distributed *Brown Dwarf* system, the chapter presents extensions that allow for dynamic replication as well as fault tolerance and discuss its potential for deployment in the Cloud.

Chapter 5 sums up the strengths and weaknesses of each of the previously proposed systems and motivates the creation of a hybrid approach that combines the best of the two worlds to target the efficient storage of time series data. The prototype system is evaluated in parts and as a whole against a state-of-the-art large scale data warehousing solution.

Chapter 6 compares our approach to related work in the literature, while Chapter 7 summarizes our conclusions and provides directions for future work. Appendix A presents a different use case scenario for the *HiPPIS* system: The core idea of the hierarchy support in DHTs is exploited for the anonymization of distributed, sensitive data.

CHAPTER 2

Background

In this chapter we introduce some basic terms used throughout the dissertation and elaborate on notions necessary to guide us through the rest of the text. First, we cover the basic concepts of data warehousing, which is the target application for the systems we propose. Then we introduce the challenges and potentials of large scale distributed systems, presenting new paradigms that are adopted in our proposed work.

2.1 Data Warehousing

Data Warehousing is a vital component of every organization in the scientific as well as the business domain, as it provides tools for data analysis, summarization and prediction of future trends in areas such as retail, finance, network/Web services, etc. A *Data Warehouse (DW)* is a central repository that hosts immense volumes of historical data from multiple sources and provides tools for their aggregation and management at different levels of granularity. Thus, unlike operational database systems that cover day-to-day operations of an organization through *Online Transaction Processing (OLTP)*, data warehouses serve users and knowledge workers in the role of data analysis and decision making through *Online Analytical Processing (OLAP)*.

The basic abstraction in data warehousing is the *data cube* [GCB+97], a multidimensional array, in the form of which data are usually viewed. Data cubes are characterized by their *dimensions*, which represent the notions that are important to an organization for managing its data (e.g., time, location, product, customer, etc.) and the *facts*, which are the numerical quantities

Table 2.1: *Sales data for electronics, according to* `time`*,* `location` *and* `product`*. The measure is dollars sold (in thousands).*

		Time								
		2008			2009			2010		
		Location								
		Athens	Patra	Lamia	Athens	Patra	Lamia	Athens	Patra	Lamia
Product	Apple	$280	$120	$65	$290	$56	$36	$320	$ 38	$24
	Sony	$250	$140	$48	$220	$59	$45	$255	$46	$37
	Intel	$180	$42	$39	$200	$39	$22	$210	$46	$24

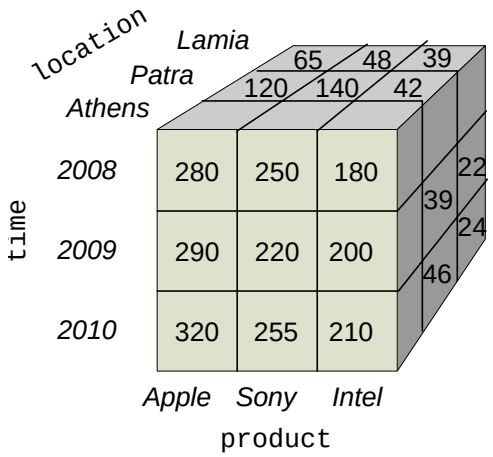


Figure 2.1: *The cube representing the data of Table 2.1*

to be analyzed (e.g., sales, profit, etc.). Figure 2.1 depicts the 3-D cube representing the data in Table 2.1 according to the dimensions `time`, `location` and `product` [HK06]. They allow for efficient summarization of data by reducing the dimensions and producing aggregate views of the data. However, data can be presented in an even more fine-grained manner through the use of *concept hierarchies*.

A *concept hierarchy* defines a sequence of mappings from more general to lower-level concepts. Figure 2.2 shows a simple hierarchy for the `location` dimension, where `Address` $<$ `ZipNo` $<$ `City` $<$ `Country` and one for `time`, where a *partial* order is defined. Concept hierarchies are important because they allow the structuring of information into categories, thus enabling its search and reuse. The mappings of a concept hierarchy are usually provided by application or domain experts.

The typical OLAP operations for the multidimensional data model, where data are organized into multiple dimensions and each dimension contains several abstraction levels defined by a concept hierarchy, offer users the flexibility to view data from different perspectives and at different levels of granularity. These are the following:

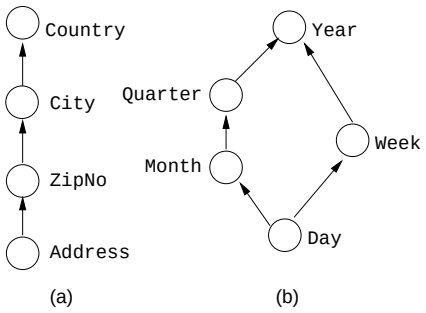


Figure 2.2: *A concept hierarchy for dimension (a)* `Location` *(b)* `Time` *(lattice)*

Roll-up The *roll-up* operation performs aggregation in a data cube either by climbing up to a more summarized level of the hierarchy or by dimension reduction. For instance, by ascending the `location` hierarchy by one level, the resulting cube groups data by `Country` rather than `City`.

Drill-down The *drill-down* operation navigates to lower levels of increased detail. For example, stepping down one level in the `time` hierarchy, the resulting cube details the sales per quarter rather than summarizing them by year.

Slice and Dice The slice operation performs a selection on one dimension of a given cube, "slicing" it and resulting in a subcube. When the selection concerns more than one dimension, the operation is called dice. As an example, the subcube for `Year`=2009 is a slice while the sales for `Year`=2009 and `City`=Athens form a dice.

Rotate This operation rotates the axis of the cube to present different views of the data.

In this thesis we focus on providing simple data warehousing functionality in a distributed environment. By simple data warehousing functionality we mean the ability to answer roll-up, drill-down, slice and dice queries. Using shared-nothing architectures we aim to store high volumes of multidimensional, hierarchical data and accommodate high rate of update and queries concerning any combination of dimensions at any granularity.

2.2 Large-Scale Distributed Environments

In the last few years, *Distributed Systems* have drawn much attention from the research as well as the business community. They are mainly defined by their common properties: Several autonomous computational entities (*nodes*), communicate with each other by message passing to achieve a common goal – from a large computational problem to the coordination of the use of

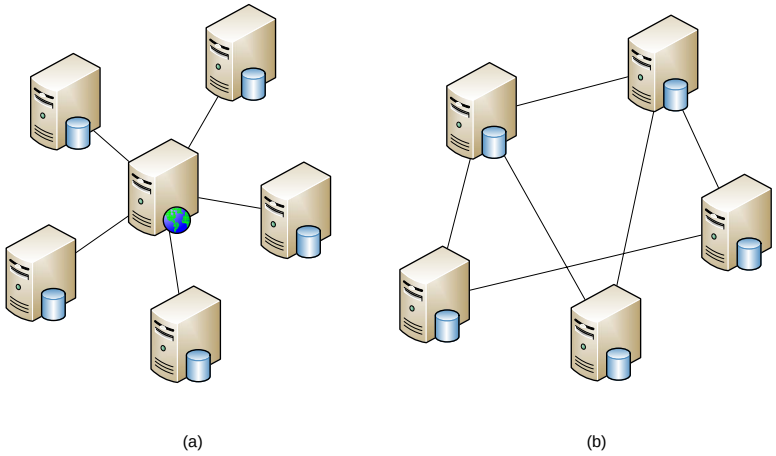


Figure 2.3: *(a)The client-server model (b)The P2P model*

shared resources. Other typical properties of distributed systems include fault tolerance of individual computers, heterogeneity of the participating entities and limited, incomplete view of the system from each node. A typical example of a distributed system is the Internet itself: Computers connected to it communicate with each other through well defined protocols (e.g., TCP/IP), make use of common services (e.g., the World Wide Web, e-mail and file transfer services) and share resources (e.g., files, printers etc.).

A long-standing tenet of distributed systems is that the strength of them can grow as more hosts participate in them. Each participant may contribute data and computing resources (such as unused CPU cycles and storage) to the overall system, thus the wealth of the community can scale with the number of participants. On the other hand, the construction of distributed systems faces numerous challenges, such as dealing with the heterogeneity of resources in terms of hardware as well as software, guaranteeing the security of the shared information and the protection of the participating resources, handling failures, ensuring transparency and many others.

2.2.1 The Peer-to-Peer System Architecture

A new paradigm for the construction of distributed systems that has become extremely popular not only in the research community, but also in the general public is that of *Peer-to-Peer* (*P2P*) systems. The P2P model dictates a fully-distributed, cooperative network design, where nodes contribute data and computational resources and collectively form a system that provides a uniform service without any supervision. Most importantly, they operate in a symmetric manner, running the same protocols and communicating freely and equally with each other. Unlike the dominant client/server model, which decomposes the system into clients that consume services and a limited number of well known, powerful sites that provide them, the P2P model consists of

nodes that serve both as clients and as servers (Figure 2.3). Its advantages, although application-dependent in many cases, are [CDK05]:

- *Decentralization*: P2P overlays have no central control over their main operations, and their correct behavior does not depend on the existence of any centrally administered part. Thus there is no single point of failure in the system.
- *Autonomy*: Peers are autonomous in every role they are assigned in a P2P system. The only key issue for their efficient operation is the choice of an algorithm for the placement of data among hosts and the subsequent access to it.
- *Network dynamics*: Peers join and leave the overlay very easily, with routing tables being updated synchronously or asynchronously with fraction of a second delays.
- *Scalability*: P2P systems have the ability to handle growing amounts of work in a graceful manner as well as enlarge their capabilities when resources (typically hardware) are added. This is a very important property, as P2P systems are meant to be used massively.
- *Robustness*: It is essential for a P2P system to be able to cope with errors during execution and to continue to operate despite abnormalities. Routes and object references are replicated n-fold ensuring tolerance of n failures of nodes or connections.
- *Self-Organization*: Because of the decentralization and scalability, there is no need for a central administration. Each participating node is responsible for its own resilience and maintenance of data as well as metadata.
- *Transparency*: One of the goals of P2P applications is to transparently locate and access data. Transparency is also aimed for replication of data as well as mobility and security of peers.

Undoubtedly, P2P systems have gained their enormous popularity because of the file sharing applications (e.g., Kazaa [kaz], eMule [emu], BitTorrent [Coh03], etc.). It is indicative that the bandwidth consumption attributed to popular such applications amounts to a considerable fraction (up to 60%) of the total Internet traffic ([san]). However, many other examples of P2P systems have emerged, most of which are wide-area, large-scale systems that provide storage [KBC+00], telephony [sky], audio and video streaming [joo], instant messaging [sky] and many other services. The most exciting possibility of P2P computing is that the desirable properties of the system can become amplified as new peers join: Because of its decentralization, the system's robustness, availability and performance grows with the number of peers. Moreover, the need for administration is diminished, since there is no dedicated infrastructure to manage.

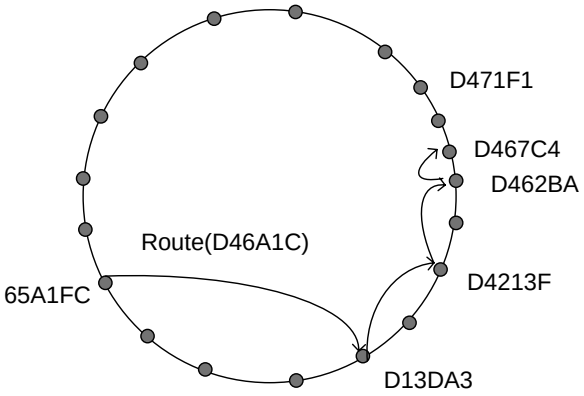


Figure 2.4: *Routing of message with GUID D46A1C in a Pastry ring*

In general, a P2P system is an *overlay network*, meaning a computer network which is built on top of another network (often the Internet itself). Nodes in the overlay can be thought of as being connected by logical links, each of which corresponds to a path, perhaps through many physical links in the underlying network. Several P2P overlays have been proposed by both the academia and the industry in the last few years in order to permit routing of messages to destinations not specified by an IP address.

We can roughly classify P2P architectures into two categories according to the degree of control over the topology and routing infrastructure they provide: *Structured* P2P systems, which follow strict rules for file placement and object discovery and *unstructured* ones that offer arbitrary network topology, file placement and search.

Structured P2P Systems Structured P2P networks employ a globally consistent protocol to guarantee the location and the retrieval of any data, if stored in the overlay, in a satisfying time complexity, usually $O(\log N)$ with N being the number of peers. The information is indexed and distributed using a global algorithm known to all participants in the system. By far the most common type of structured P2P network is the *Distributed Hash Table (DHT)*, in which a variant of consistent hashing is used to assign ownership of each file to a particular peer, in a way analogous to a traditional hash table's assignment of each key to a particular array slot.

In DHTs, (key, value) pairs are stored and any participating node can efficiently retrieve the value associated with a given key. The responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows DHTs to scale to large numbers of nodes and to handle continuous node arrivals, departures and failures. Some prominent research projects include Chord [SMK+01], Pastry [RD01], Kademlia [MM02], P-Grid [ACMD+03] and others.

In a nutshell, in DHTs resources as well as data are identified by globally unique identifiers (GUIDs), usually derived as a secure hash from some or all of the resource's state. Each node is responsible for processing requests addressed to all objects in its numerical neighborhood. The simplest API used to manipulate data is:

put(GUID, data) The data is stored at all nodes responsible for the object identified by the GUID.

get(GUID) The data associated with GUID is retrieved from one of the nodes responsible for it.

If the GUID identifies a node that is currently active, the message is delivered to it, otherwise it is forwarded to the node whose GUID is numerically closest to it (according to some proximity metric). As a use case, we present the Pastry routing algorithm, since Pastry is used as a substrate for *HiPPIS*, the first system proposed in this dissertation. All other DHTs operate in a similar manner, mainly differing in the proximity metric utilized.

In Pastry, the GUID space is treated as circular. Each node maintains a *leaf set*, a vector of size 2ℓ containing the GUIDs and IP addresses of the nodes numerically closest on either side (ℓ above and ℓ below). Moreover, each pastry node holds a tree structured routing table, which maps GUIDs to IPs of a set of nodes spread throughout the entire range of the possible GUID values with increased density of coverage for GUIDs close to its own. When routing a message m, the node n is able to scan both the leaf set and the routing table and forward it to the node whose GUID has the most matching prefix digits with m's GUID but is closest to n's GUID. Figure 2.4 depicts the routing steps required for a message starting from a node to reach its destination.

For further reading, there exists a plethora of works overviewing and comparing the most popular DHT overlays and their operations in terms of performance and cost [BKK+03, LSM+05, KK07, JMW03, LCP+04].

Unstructured P2P Systems An unstructured P2P network is formed when the overlay links are established arbitrarily. The simplicity of its construction is outweighed by the lack of guarantees about the location and retrieval of any data stored in the system, especially with low time complexity, since searching is performed through flooding. In practice, popular content is likely to be available at several peers, thus it is feasible to retrieve it efficiently. The lack of correlation between a peer and the content managed by it alleviates the burden of the high cost of the necessary maintenance of indices and other such structures. This has led to the success of applied P2P unstructured overlays (e.g. Kazaa [kaz], Gnutella [gnu03], Freenet [CSWH01] etc.). A survey and comparison of popular unstructured overlays can be found in various studies in the relative bibliography [TR03, LCP+04, LCC+02].

Consequently, DHTs guarantee the discovery of a data item as long as it exists in a bounded number of hops at the cost of maintenance overhead. Indeed, structured graphs are more expensive to maintain than unstructured ones due to the constraints imposed by the structure. On the other hand, unstructured overlays can perform complex (like range or boolean) queries more efficiently than structured overlays but may prove inefficient in discovering (if at all) unpopular data items since the lookup service relies on flooding. While both classes of P2P systems offer tolerance in node churn, unstructured overlays can cope better in environments with extremely transient peers due to the lack of strict rules for data placement [CCR04].

In this dissertation the potentials of both structured and unstructured P2P overlays are exploited. First, a system has been built, that relies on a DHT to enable fast operations on the data with minimal communication cost, while additional design choices have been made for the handling of more complex structured data (multidimensional, hierarchical). Next, in our efforts to distribute a complex data structure (data cube) we create a system that benefits from the simplicity of an unstructured overlay for efficient indexing, with special effort given to algorithms that ensure availability and load balancing. In the last proposed system the advantages of both are combine to offer an integrated solution for the efficient handling of temporal data.

2.2.2 Cloud Computing

The current trend in distributed computing dictates the exploitation of multiple commodity machines rather than the construction of conventional supercomputers to tackle the increased need for CPU power as well as disk storage. The *Grid* [FKT01, FKNT03] was a pioneering effort to create wide-area, large-scale distributed computing systems, in which remotely located, disjoint and diverse processing and data storage facilities are integrated. However, it remained mostly targeted on the scientific world, unlike its latest descendant, the *Cloud* [AFG+09], which gains ground both in the academic as well as the commercial world.

Cloud Computing represents a computing paradigm where computation and storage alike move towards network-centric data centers hosted by large infrastructure companies. The new aspects that distinguish Cloud Computing from other distributed architectures is the typically elastic resource availability, which gives the illusion of infinite computing power and storage available on demand and the particularly appealing pricing model based on direct storage use and/or number of CPU cycles consumed, which largely alleviates companies from the cumbersome and expensive maintenance costs.

In such environments, many remotely located users are able to share data to produce useful results, thus there is an urgent requirement to obtain solutions to manage, distribute and access large sets of raw and processed data efficiently and effectively. Currently, there exists a large interest in cloud-based data management, with big companies such as Amazon, Google, Yahoo,

Oracle, Sun, etc., providing either the computational resources or the application platforms. For an application to be deployed in the Cloud, it should provide some architectural characteristics, such as [BvET08]:

- *Cost-efficiency*: Data volumes and transfers as well as computational costs should be minimized, as billing is always relative to the resource usage. Hence, we require efficient data-compression combined with high-performance operations.
- *Elasticity*: Computing power and storage should be adaptively allocated according to demand, giving the users the impression of infinite resources at their disposal. This requirement can be made possible through the use of a shared-nothing architecture.
- *Content availability*: More precisely, a data management application should replicate data automatically across the nodes in the Cloud, be able to continue running in the event of multiple node failures and be capable of restoring data on recovered nodes automatically.

In this thesis, the proposed systems are based on a shared-nothing architecture, providing most of the above characteristics. As such, they constitute ideal candidates for deployment in distributed environments like the Cloud.

CHAPTER 3

The Hierarchical Peer-to-Peer Indexing System

This chapter describes a distributed system designed to efficiently store, query and update multi-dimensional data organized into concept hierarchies and dispersed over a network. Our system employs an adaptive scheme that automatically adjusts the level of indexing according to the granularity of the incoming queries, without assuming any prior knowledge of the workload. Efficient roll-up and drill-down operations take place in order to maximize the performance by minimizing query flooding. Updates are performed online, with minimal communication overhead, depending on the level of consistency needed. Extensive experimental evaluations show that, on top of the advantages that a distributed storage offers, our method answers the large majority of incoming queries, both point and aggregate ones, without flooding the network and without causing significant storage or load imbalance. Our scheme proves to be especially efficient in cases of skewed workloads, even when these change dynamically with time. At the same time, it manages to preserve the hierarchical nature of data. To the best of our knowledge, this is the first attempt towards the support of concept hierarchies in DHTs.

3.1 Overview

As a motivating scenario, let us consider a geographically dispersed business or application that produces immense amounts of data, e.g., a multinational sales corporation or a data-collection

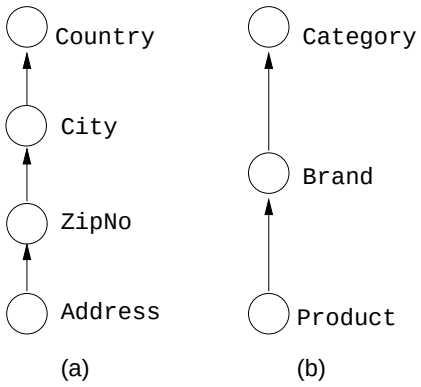


Figure 3.1: *A concept hierarchy for dimension (a)* `location` *(b)* `product`

facility that processes data from Internet routers. We argue for a completely decentralized approach, where users can perform *online* queries on the multiple dimensions, simple yet important mining operations (such as roll-up and drill-down on the defined hierarchies) and calculate aggregate views that return important data summaries. Such an application, besides eliminating the central storage and processing bottleneck and minimizing human coordination, enables querying the data in real time, even if some of the resources are unavailable.

Let us assume that the company's database contains data organized along the `location` and `product` dimensions (see Figure 3.1). In a plain DHT system, one would have to choose a level of the suggested hierarchy in order to hash all tuples to be inserted to the system and repeat this for each dimension. Assuming the tuples are hashed according to the `city` and `category` attributes, there will be a node responsible for tuples containing the value *Athens*, one for *Milan*, etc., as well as nodes responsible for *Electronics*, *Household*, etc. This structure can be very effective when answering queries referring to the chosen levels of insertion (and even so, intersection of tuples will be necessary), whereas queries concerning other hierarchy levels demand global processing.

The solution of multiple insertion of each tuple by hashing every hierarchy value of each dimension is not viable: As the number of dimensions and levels increase, so does the redundancy of data and the storage sacrificed for this purpose. Furthermore, while point queries would be answered without global processing, this scheme fails to encapsulate the hierarchy relationships: One cannot answer simple queries, such as "*Which country is Patras part of*" or "*What is the total revenue for Electronics products sold anywhere*".

This chapter investigates the problem of indexing and querying hierarchical data in DHTs in a way that preserves the semantics of the hierarchies and is efficient in retrieving the requested values for *both* point and aggregate queries. To that end, we propose the *Hierarchical Peer-to-Peer Indexing System (HiPPIS)*, a DHT-based system that stores and indexes bulk data in the form of a fact table (e.g., Table 3.1) to multiple sites over the network. It also enables efficient

querying over multiple dimensions characterized by specific hierarchies. Thus the system benefits from the inherent characteristics of the P2P architecture, such as scalability, fault tolerance and availability relying solely on commodity nodes. *HiPPIS* nodes actively monitor the granularity of posed queries in order to adjust the indexing level to the most beneficial one. Combined with soft-state indices which are dynamically created after query misses, our system manages to minimize the number of flooding operations necessary to provide exact answers. Furthermore, *HiPPIS* does not invalidate the semantics of the stored hierarchies and allows for distributed knowledge mining.

Peers initially index at a default (*pivot*) level combination. Inserted tuples are internally stored in a hierarchy-preserving manner. Query misses are followed by soft-state pointer creations so that future queries can be served without re-flooding the network. Peers maintain local statistics which are used in order to decide if a reindexing (to a different combination of hierarchy levels) is necessary, according to the current query trend. For instance, if the ratio of queries for ⟨`country, brand`⟩ exceeds a threshold (assuming the pivot level is ⟨`city, category`⟩), data would be reindexed according to that level combination so that most requests would be directly answered. Besides answering point queries at different granularity, *HiPPIS* can answer group-by queries, such as "*Give me the sales registered for Greece for ALL products*".

It has been widely observed that most Internet-scale applications, including P2P ones, exhibit highly skewed workloads (e.g., [CKR+07, RFI02], etc). *HiPPIS* indexes popular levels and uses indices to answer the less popular requests. It adapts to the incoming workload as a whole, without assuming any prior knowledge of the data or workload distributions and without any precomputations on the data. The contribution of our work can be summarized in the following:

- It addresses the problem of hierarchical data search in DHT systems. Even though DHTs bind the number of query hops to the logarithm of the size of the overlay, they are unable to directly support queries on dimension hierarchies, since they perform exact match lookups. Any other case would require message and time-consuming query flooding over the whole network. Our technique, taking into account user preferences and sensing potential overall tendencies, allows reorganization of the indexing structure in favor of resolving queries for the most popular data. It also manages to preserve the useful hierarchy-specific information that hashing destroys. Either through hashing on a single or multiple levels of the hierarchy, a naive data insertion would fail to preserve the associations between the stored keys. By using a tree-like data structure to store data and maintain indices to related keys, our system is able to respond to more complex, hierarchy-based queries.
- It allows for online updates, unlike the conventional update technique in data warehousing, which dictates an offline update application on a daily or weekly basis. The communication overhead depends on the level of consistency needed by the application.

Table 3.1: *Sample fact table*

	Location			*Product*		Fact
TupleID	Country	City	Zip	Category	Brand	Sales
ID2	Greece	Athens	16674	Electronics	Apple	11,500
ID5	Greece	Athens	15341	Electronics	Sony	1,900
ID51	Greece	Athens	15341	Electronics	Philips	22,900
ID31	Greece	Athens	16732	Household	AEG	2,450
ID55	Greece	Larissa	20100	Electronics	Sony	12,100
ID190	Greece	Patras	19712	Household	Unilever	1,990
ID324	Greece	Athens	17732	Electronics	Philips	2,450
ID501	Greece	Athens	17843	Electronics	Sony	12,000
ID712	Greece	Athens	17843	Electronics	Apple	32,000

- It presents a thorough experimental section where we clearly identify the advantages of our proposed system in a variety of workloads (variable levels of skew, dynamic changes, etc.), datasets and update setups. We also register the induced data and load distributions across the nodes of the overlay. *HiPPIS* achieves a high ratio of exact-match queries in a variety of workloads, even when these change dynamically with time. We show that our scheme is particularly efficient with highly skewed data distributions which are frequently documented in the majority of applications, without inducing significant load or storage imbalance among the network nodes. Moreover, even under high update rates, the freshness of the query responses remains acceptable.

3.2 HiPPIS Design

3.2.1 Necessary Notation

Our data spawn the d-dimensional space. Each dimension i is organized along $L_i + 1$ hierarchy levels: $H_{i0}, H_{i1}, \ldots, H_{iL_i}$, with H_{i0} being the special *ALL (∗)* value. We assume that our database comprises of fact table tuples of the form:
$\langle tupleID, D_{11} \ldots D_{1L_1}, \ldots, D_{d1} \ldots D_{dL_d}, fact_1, \ldots, fact_k \rangle$, where $D_{ij}, 1 \leq i \leq d$ and $1 \leq j \leq L_i$ is the value of the j^{th} level of the i^{th} dimension of this tuple and $fact_i, 0 \leq i \leq k$ are the numerical facts that correspond to it (we assume that the numeric values correspond to the more detailed level of the cube). Our goal is to efficiently insert and index these tuples so that we can answer queries of the form: $q = \langle q_1, q_2, \ldots, q_d \rangle$, where each query element q_i can be a value from a valid hierarchy level of the i^{th} dimension, including the $*$ value (dimensionality reduction): $q_i = D_{ix}, 0 \leq x \leq L_i$.

3.2.2 Data Insertion

The insertion of a data tuple (or a pointer to the real location of it) is performed as follows: Upon creation of the database, a combination of levels is globally selected. This is called *pivot* $P = \langle p_1, p_2, \ldots, p_d \rangle$, where each pivot element p_i can be a valid hierarchy level of the i^{th} dimension (including the special $*$ value): $p_i = H_{iy}, 0 \leq y \leq L_i$. The ID of each tuple to be inserted is the hashed value combination corresponding to the pivot. The DHT then assigns each tuple to the node with ID numerically closest to this value. For tuples inserted at a later stage, nodes can be informed of P from one of their neighbors in the overlay.

Inserted data are stored in the form of trees that preserves their hierarchical nature. Nodes store multiple forests, one for each d-valued combination it is responsible for. As a consequence, each distinct value of the pivot level combination corresponds to a forest that reveals part of the hierarchy. Each forest consists of d rooted trees, one for each dimension. To see this pictorially, let us refer to the example depicted in Figure 3.2. Let us assume the data contained in Table 3.1 and the hierarchy of figure 3.1 (without the last level of each dimension) with $\langle city, category \rangle$ as the globally defined pivot. The first tuple to be inserted is assigned an ID that derives from applying our hash over the value *Athens*||*Electronics* and forms a forest with two plain lists (Figure 3.2(a)). As data items with the same ID keep arriving at this node, different values at levels lower in the hierarchy than the pivot levels create branches, thus forming a tree structure (Figure 3.2(b) and (c)). The trees of a forest are connected (in order to retrieve the corresponding facts) through the tuple IDs, depicted as a linked list in Figure 3.2.

3.2.3 Data Lookup and Indexing Mechanism

Queries concerning P are defined as *exact match* queries and can be answered within O(log N) forwarding steps. Since we have included the $*$ as the top level of the hierarchy of each dimension, P may include $*$ in any of its d possible values. Therefore, assuming the query elements $q_i = D_{ix}$ and the respective pivot level elements $p_i = H_{iy}$, the query is an *exact match* one if $x = y$, in the case it comprises of exact values, or if $p_i = *$. Queries on any of the other level combinations cannot be answered unless flooded across the DHT. In order to amortize the cost of this operation and facilitate such requests, we introduce *soft-state indices* to our proposed structure. These indices are created on demand, as soon as a query for non-pivot level data is answered. After the answers from the corresponding nodes are received through overlay flooding, the query initiator hashes the value of the requested key and sends the IDs of the nodes that answered the query to the node responsible for that key. So, essentially, we term indices the pointers from a node that should hold the answer to a query, had the pivot been the queried level combination, to the node or nodes that actually store the answer.

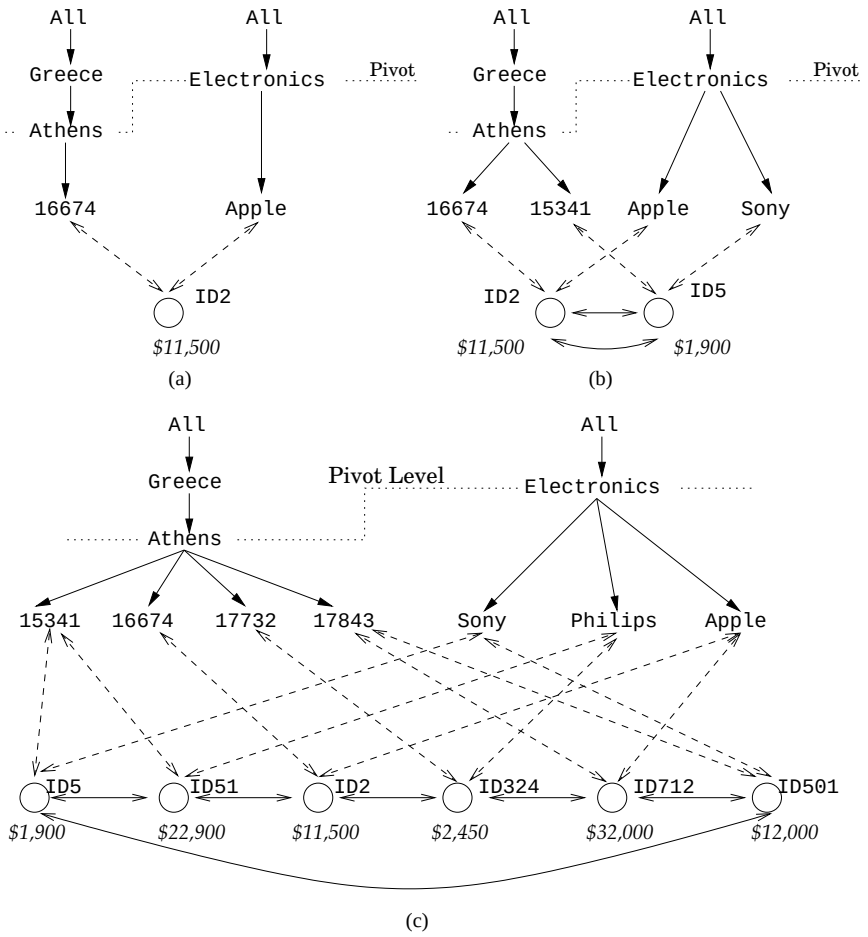


Figure 3.2: *The forest structure at node responsible for* `Athens,Electronics` *after the insertion of (a) the first tuple, (b) the second tuple and (c) all tuples of Table 3.1*

Soft-state indices give users the illusion that the queried values are actually hashed and retrieved in a fast manner. In reality, O(log N) steps are required to locate the indices which are then used to retrieve the multiple tuples required to compute the correct result set. The number of indices followed depends on the query and P: If the query attributes are of equal/smaller level than the respective pivot level elements, only a single pointer will exist. Otherwise multiple (the exact number depends on the data) pointers need be followed.

The created indices are soft-state, in order to minimize the redundant information. This means that they expire after a predefined period of time (Time-to-Live or TTL), unless a new query for that specific value is initiated, in which case, the index is renewed. This mechanism ensures that changes in the system (e.g., data location, node unavailabilities, etc.) will not result in stale indices, affecting its performance. Apparently, in cases of very large datasets and uniform query distributions the index size can grow large. While memory becomes a cheaper commodity

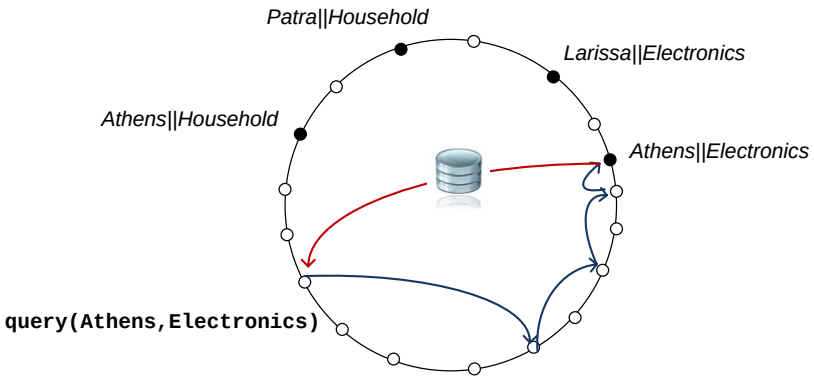


Figure 3.3: *Lookup for* ⟨*Athens,Electronics*⟩

by the day, the plain size of data discourages an "infinite" memory allocation for indices. After the number of created indices per node has reached the limit I_{max}, the creation of a new index results in the deletion of the oldest one. Calibrating I_{max} for performance without increasing it uncontrollably entails knowledge of our data (e.g., how skewed each hierarchy is). Thus, the system tends to preserve the most "useful" indices, namely the ones that refer to the most frequently used data items. The HiPPIS lookup and indexing algorithm is presented in Algorithm 3.1.

As an example, let us assume the same hierarchy as before, with ⟨`city`, `category`⟩ as P. A query for ⟨*Athens,Electronics*⟩ is an exact match one and translates to a simple DHT lookup operation (Figure 3.3). When querying for ⟨*16674,Apple*⟩, we discover that no such key exists in the DHT. Flooding is performed and the node *Athens*||*Electronics* answers with the corresponding tuple. The initiator, which now knows the ID of the node that answered the query, forwards it to the node responsible for the value *16674*||*Apple*, which now has an index pointing to the node *Athens*||*Electronics*. Thus, in case of another query referring to the same value, the time and bandwidth consuming flooding is avoided and the response can be provided quickly and efficiently, within logN+C hops.

The same procedure takes place when the query concerns a value that lies higher in the hierarchy than the pivot. The query for ⟨*Greece,* $*$⟩ is routed to the node responsible, where no answer is available. Flooding is performed and the nodes that contain relevant tuples are discovered. Finally, the data satisfying the query are returned to the initiator and multiple indices are built. Both these cases are shown pictorially in Figures 3.4 and 3.5 respectively, where the black nodes are the ones that store the actual data, whereas the nodes holding pointers are depicted in gray. The pointers themselves are represented as dashed arrows.

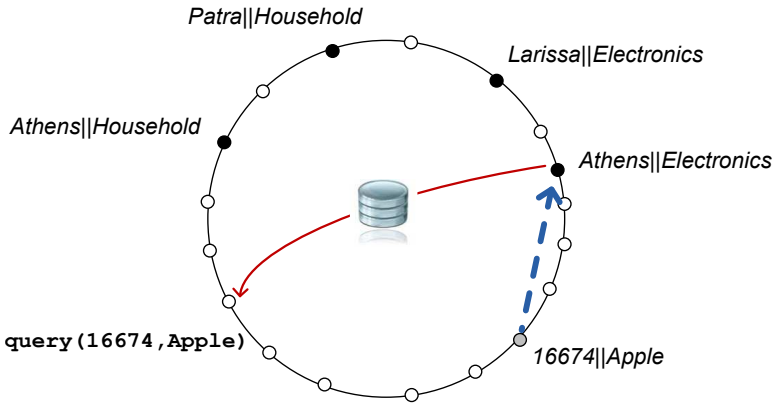


Figure 3.4: *Lookup and index creation for ⟨16674,Apple⟩*

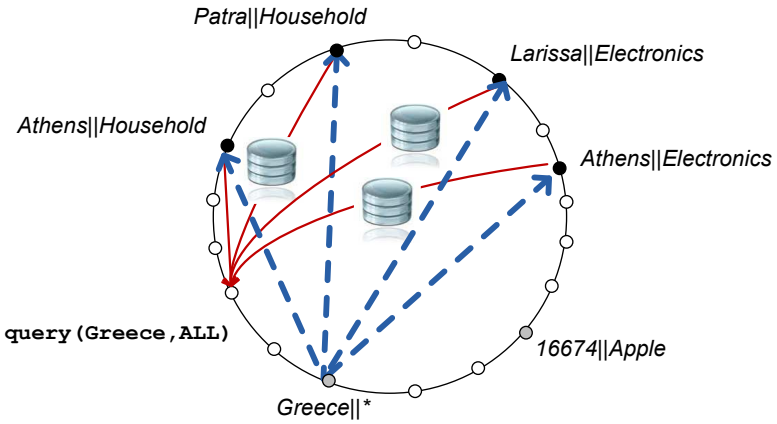


Figure 3.5: *Lookup and index creation for ⟨Greece, ∗⟩*

3.2.4 Reindexing Operation

In a data warehouse, the distribution of data or queries may vary over time. Thus, it is possible that the choice of P, which is done once at the beginning, does not favor performance. *HiPPIS* is adaptive to the query distribution, supporting dynamic changes in the pivot, without assuming any prior knowledge, being solely based on locally maintained statistics. By shifting to a different level combination we aim at increasing the ratio of exact match queries, reducing floodings and boosting performance. The exact procedure is presented in Algorithm 3.2.

If the number of queries initiated by a node regarding level combinations different than P exceeds the number of queries for P by some *threshold*, this node considers the possibility of a new partitioning. Each node determines the *popularity* of each level combination ($\prod_{i=0}^{d} L_i$ exist) by measuring the number of queries it has locally initiated within the most recent time-frame W.

This time-frame should be properly selected to perceive variations of query distributions and, at the same time, stay immune to instant surges in load.

Algorithm 3.1: HiPPIS Lookup and Indexing Algorithm

Data: $q = \langle q_1, q_2, \ldots, q_d \rangle$: the query to be resolved
$P = \langle p_1, p_2, \ldots, p_d \rangle$: the pivot level combination
r: remote node
$K_{r,exact}$, $K_{r,ind}$: set of keys held and indexed by remote node respectively
Result: Tuples returned to the query initiator

if $\exists P_s \subseteq P : \forall p_i \in P_s, p_i = * \wedge q_i \neq *$ *and the rest of the attributes* $(P - P_s)$ *are of the same level* **then**
    $ID_q \leftarrow$ hash(q) where q_i is replaced by $*$;
    `DHT_route` (LookupMessage) to r responsible for ID_q;
    local processing by r and possible answers returned;
end
else
    $ID_q \leftarrow hash(q)$;
    `DHT_route` (LookupMessage(ID_q));
    local processing by r;
    if $ID_q \notin K_{r,exact}$ **then**
        if $ID_q \notin K_{r,ind}$ **then**
            `flood` (q), local processing by each r ;
            answers returned by set of nodes R ;
            `DHT_route` (IndexMessage($ID_q \rightarrow R$));
            Receiver nodes add ID_q to $K_{r,ind}$;
        end
        else
            local processing, tuples returned
        end
    end
    else
        tuples returned
    end
end

If the percentage of the queries on the most popular level combination c_{max} is more than *threshold* of the respective pivot popularity, the node is positive to the potential of adopting another pivot. If this is the case, reindexing enters its second phase, in which the local intuition

Algorithm 3.2: HiPPIS Reindexing Algorithm

Data: P: current pivot level combination
$popularity_{c_i}$: popularity of level combination c_i
$C_{local} : c_0 < c_1 < \ldots < c_{max}$ ranked level combinations according to local popularity
Result: Reindexing of data

if $popularity_{c_{max}}$ - $popularity_P >$ threshold **then**
  `flood` (SendStatsMessage) and collect global statistics;
  $C_{global} : c_0 < c_1 < \ldots < c_{max}$ ranked level combinations according to global popularity;
  calculate *threshold*;
  if $popularity_{c_{max}}$-$popularity_P >$ threshold **then**
    determine new pivot level P_{new};
    if $P_{new} \neq P$ **then**
      `flood` (ReindexingMessage(P_{new}));
      $P \leftarrow P_{new}$, rehashing of tuples;
    end
  end
end

must be confirmed (or not) using global statistics. The node whose local information indicates a possible shift of P sends a *SendStats* message to all system nodes. The initiator, after collecting the statistics from all nodes, redefines c_{max} and repeats the aforementioned procedure, enhanced with a strategy for the optimal pivot selection, thoroughly described in the next section. In the case of a new P selection, reindexing is performed respectively by all nodes.

It should be noted here that the first phase of the reindexing process is not decisive for the selection of the new potential pivot; it is rather used as an indication of an imbalance that should be further investigated. Thus, we assume that nodes act altruistically, not only by reporting their true statistics, but also in the sense that they may trigger a change of pivot that may not reflect their personal preferences.

The initiating node floods a *Reindex* message to force all nodes to change their pivot. Each node that receives this message traverses its tuples, finds all the values of the level combination that will constitute the new reference point and rehashes them one by one, sending the tuples to the corresponding nodes. Assuming that the size of the dataset $|D| \gg N^2$, N being the size of the network, the preferred method to perform this is to send at most $N - 1$ messages per node, grouping the tuples by recipient. After the node completes the procedure, it erases all its data and indices.

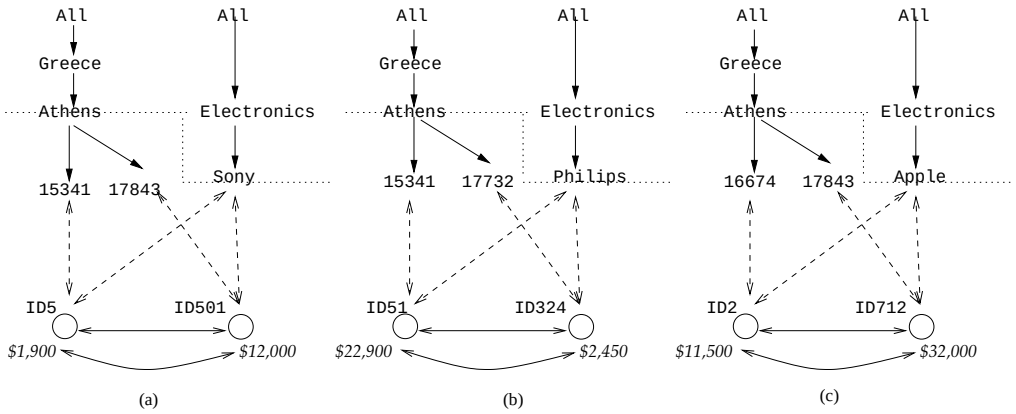


Figure 3.6: *The produced tree structures after Reindexing*

Back to our example, if the node *Athens*||*Electronics* receives a *Reindex* message for ⟨`city`, `brand`⟩, it runs through its tuples and discovers that the values corresponding to that level combination are *Athens*||*Sony*, *Athens*||*Philips* and *Athens*||*Apple*. The values are hashed and the corresponding nodes are now responsible for the tuples containing these values (Figure 3.6).

3.2.5 Locking

In order to ensure the correctness of the answers during the reindexing process and to avoid simultaneous reindexings by multiple nodes, we introduce a locking mechanism. After a node decides to perform reindexing according to the global statistics, it first sends a *Lock* message to all system nodes and then proceeds to it. Once a node receives the *Lock* message, it changes its state to LOCKED and maintains it for a predefined period of time (related to the network size), which we assume is adequate to cover the time needed for the whole system to finish reindexing and to reach a stable state. During this time, locked nodes continue answering queries through flooding. Therefore the system constantly remains online.

To cope with the issue of possible concurrent locks, we adopt a simple resolution mechanism: Since each *Lock* message, upon creation by the initiator, is identified by a (local) timestamp and the initiator's ID, nodes receiving more than one *Lock* messages within a small time frame may assume as valid the one with the earliest timestamp (or the one coming from the lowest ID) and accept *Reindex* messages only by its initiator. If the newly received *Lock* is not valid, the node stops forwarding it. An already LOCKED node is not allowed to initiate another locking.

We should note here that since each node collects global statistics before a new pivot decision, it is impossible that two non-malicious nodes come to a different decision. This would only be possible (with low probability) if sampling was used for global statistics collection. Even so, the locking mechanism makes sure that only one node at a time can instruct reindexing.

3.2.6 Updates

Tuple updates are normally performed through an update of the tuple's measures at the corresponding node. One open issue relates to the insertion of new tuples in the system. While hashing according to the current pivot and storing the new item is trivial, there may exist indices that need to be updated since the new tuple must be included in the result set of various queries. As an example, consider an inserted tuple that documents sales of electronics in a new Greek city. An existing index for $q = \langle$*Greece, Electronics*$\rangle$ should now include the ID of the node responsible for the new tuple. It should be noted that inconsistencies may arise only by tuples that contain new pivot level combinations and thus create new forests. Since the creation of an index may be followed by one or more index deletions at the creating node (due to space constraints), the inserting node cannot know of the existence or not of an index relative to the new tuple a priori. This can be resolved in a variety of ways, according to the level of consistency that we require from our system. We identify the following two cases:

- *Strong consistency:* For applications that rely on constant data analysis and immediate detection of changes in trends, it is crucial that, at any time, any query to the data warehouse returns the complete and most up-to-date answers. For instance, in case of an intrusion detection application which analyzes data created by geographically dispersed routers, denial-of-service (DoS) attacks must be tracked immediately to protect the routers from collapsing. To achieve strong consistency, after each tuple insertion, the node performs $\prod_{i=0}^{d} L_i - 1$ lookups to identify the existence of all possible index combinations. Each node that holds a corresponding combination will update its value. Thus, consistency is guaranteed in exchange of a higher communication cost, which depends on the rate λ_{upd} at which updates are being performed.

- *Weak consistency:* When the application can afford some "staleness" in the data, a weak consistency scheme can be applied. Nodes append the inserted tuples to a globally known location. Index-holding peers can then, asynchronously, retrieve this directory and update the required indices. During the time period between the new tuple insertion and the asynchronous index update, it is possible that some answers are not 100% up-to-date. The freshness of the responses depends on λ_{upd}, as well as on the rate λ_{index} at which each node contacts the central directory and updates its indices. The communication cost is smaller than that of the strong consistency scheme, since $\lambda_{index} < \lambda_{upd}$. Therefore, this approach is recommended in cases where bandwidth resources are limited and not 100% accuracy is required.

3.3 Discussion - Enhancements

In this section we discuss some important aspects of *HiPPIS* that relate to its parameters as well as optimization issues.

3.3.1 Memory requirements

A node running *HiPPIS* requires space for the combination statistics ($O(\prod_{i=0}^{d} L_i)$ modulo the window W) plus the storage required for the soft state indices. Each created index for a specific key holds, besides the key itself and its time of creation, the IDs of the nodes that hold the relative tuples. The number of different IDs is bound by the size of the network N. Hence, if K_{max} is the maximum number of non-pivot keys held by a node, each node requires $O(NK_{max})$ bytes. Note here that in this calculation we have not included the amount of space reserved for the data at each node (usually not stored in main memory). Nodes can either physically store the data or pointers to their original locations. Whichever the case, the amount of space per forest depends on P (besides the data distribution of course): The more coarse grained the hierarchy levels in P, the larger the number of tuples that correspond to each tree.

3.3.2 Parameter Selection

A careful choice of the *TTL, W,* K_{max} parameters plays an important role in the performance of the system. A small TTL degrades the success ratio of the search mechanism, invalidating indices unnecessarily. Assuming the rate at which participating peers delete their data or disconnect is small (a reasonable assumption for our motivating application), a large value for TTL will not create a stale image that fails to reflect the infrequent changes.

The window parameter W represents the number of previous statistics that each node stores and uses in order to decide a pivot change. A large value for W will fail to perceive load variations, whereas a very small value will possibly lead to frequent erroneous or conflicting reindexing decisions. In order to estimate its value, we set $W = O(1/\lambda)$, i.e., we connect the size of the window with the query inter-arrival time. The more frequent the requests, the smaller W can be and vice versa.

In order to estimate λ, we need the 0^{th} and 1^{st} frequency moment (F_0 and F_1 respectively) of the request sequence arriving at a node. F_0 is the number of distinct IDs that appear in the sequence, while F_1 is the length of the sequence (number of requests). Nodes can easily monitor the number of incoming requests inside a time interval. Many efficient schemes to estimate F_0 within a factor of $1 \pm \epsilon$ have been proposed (e.g., [BYJK$^+$02, AMS99]). We use one of the schemes in [BYJK$^+$02], which requires only $O(1/\epsilon^2 + \log(m))$ memory bits, where m is the number of

distinct node IDs. In reality, m is in the order of the network size, since all nodes may possibly reach it in the DHT.

Finally, regarding the total amount of memory dedicated per node, this is dominated by the maximum number of non-pivot keys K_{max} that a node is responsible for. Assuming a value of $N = 1K$ nodes for our application and that IDs and keys need 20 bytes (as outputs of the SHA1 hash function), a node that is responsible for 1K different keys will need at most 20MB of memory while for 10K keys a node will need at most 200MB of memory (certainly affordable by most modern desktop PCs).

3.3.3 Reindexing Cost and Load Balancing

Reindexing is a costly procedure, as it requires network flooding for the collection of statistics and the consecutive re-insertion of tuples. Instead of crawling the entire network, the global statistics collection could be based on uniform sampling, thus decreasing the number of required messages. Random sampling in DHTs can be achieved simply by generating identifiers at random and finding the peers closest to them. Because peer identifiers are generated uniformly, we know they are uncorrelated with any other property. This technique is simple and effective, as long as there is little variation in the amount of identifier space that each peer is responsible for. Such a sampling technique was used in various studies of widely deployed DHTs (e.g., [SR06]). However, the re-insertion of tuples is the operation that dominates the complexity of the reindexing process, requiring $\Omega(N^2)$ messages. Therefore, it is important to ensure that our gains from reducing query floodings outweigh this cost.

More formally, let us assume that x and y are the pivot level combinations before and after reindexing respectively. As $Gain_{x \to y}(t)$ we denote the gain in messages after reindexing as a function of time and as $Cost_{x \to y}$ the cost of reindexing in messages. To conclude that a reindexing was indeed beneficial for the system, the following statement should be true:

$$
\begin{aligned}
&\text{Cost}_{x \to y} < \text{Gain}_{x \to y}(t) \Rightarrow \\
&\text{Cost}_{x \to y} < \text{EM}_x \cdot \log(N) + \text{Fl}_x \cdot N - \text{EM}_y \cdot \log(N) - \text{Fl}_y \cdot N \Rightarrow \\
&\text{Cost}_{x \to y} < \lambda_x \cdot t \cdot \log(N) + (\lambda - \lambda_x) \cdot t \cdot N - \lambda_y \cdot t \cdot \log(N) - (\lambda - \lambda_y) \cdot t \cdot N \Rightarrow \\
&\text{Cost}_{x \to y} < (N - \log(N))(\lambda_y - \lambda_x) \cdot t
\end{aligned}
$$

where EM_i and Fl_i represent the exact match and flooded queries respectively for level combination i. Moreover, we assume no soft-state indices, a steady query arrival rate λ and steady query rates λ_x and λ_y targeted towards x and y respectively. $Cost_{x \to y}$ is bound by N^2, since the size of

the dataset $|D| \gg N^2$ and thus messages are grouped by recipient. So, in the worst case:

$$N^2 \cdot \log(N) < (N - \log(N))(\lambda_y - \lambda_x) \cdot t \Rightarrow$$
$$(\lambda_y - \lambda_x) \cdot t > \frac{N^2 \cdot \log(N)}{N - \log(N)}$$

and for large N values, $N - \log(N) \cong N$

$$(\lambda_y - \lambda_x) \cdot t > N \cdot \log(N) \tag{3.1}$$

From Equation 3.1 we derive that a reindexing is beneficial in terms of messages when the difference in the number of exact matches before and after reindexing is greater than a number depending on the network size. This can be achieved either when there is a reasonable difference between the query arrival rates of the two level combinations or when adequate time has elapsed before a new reindexing takes place. We must stress out that this formula represents the worst case scenario for *HiPPIS*, since we have assumed that soft-state indices do not contribute to the system's gain. However we expect that, depending on the posed workload, soft-state indices can significantly decrease the number of messages exchanged and thus lead to a balance between reindexing *Cost* and *Gain* more quickly.

Furthermore, following our previous discussion, there is a clear trade-off between the amount of space per forest (via the choice of the pivot) and the amount of processing corresponding to each node: The higher the pivot levels, the more requests are handled through a single node. In this work, we do not explicitly deal with the load-balancing problem (caused either by uneven load or data distribution), as this is orthogonal and can be handled in a variety of well-documented ways in a DHT (e.g., [PNT06, GSBK04], etc.). Nevertheless, for our target applications, we believe and prove in our experimental section that an uneven data distribution is unlikely: The number of participating peers is not expected to be very high so that a uniform hashing of the existing combinations even at the highest levels will result in a uniform data distribution.

3.3.4 Minimize Global Statistics Collection

In order to minimize the number of occasions where global statistics are collected due to nodes interested in suboptimal levels or malicious users, we define the $interval_{n,t}$ parameter for each node n at the t^{th} time it checks its statistics. This parameter defines the minimum time-stretch between two consequent checks that can be initiated by n and coincides with the frequency of n checking its statistics. Its initial value T_s is the same for all nodes: $interval_{n,0} = T_s$. In order to discourage consecutive reindexing attempts from the same node, this parameter is multiplicatively increased when the processing of global statistics concludes in different results or in a

no-change decision and reset to T_s otherwise. Specifically:

$$interval_{n,t} = \begin{cases} 2 \times interval_{n,t-1} & \text{if conflict between} \\ & \text{local \& global stats} \\ T_s & \text{otherwise} \end{cases}$$

3.3.5 Threshold Selection

The *threshold* parameter is of vital importance for the efficiency of the system, and should therefore be carefully determined in order to avoid unnecessary reindexing decisions. Frequent index reorganizations should be discouraged, yet beneficial reindexing should not be prevented. The node having initiated the collection of global statistics calculates the *popularity* of each level combination, that is, the percentage of queries concerning that specific level combination, and ranks them according to this metric ($C : c_0 < c_1 < \ldots < c_{max}$). The overall query distribution should be taken into account as well, since it is possible that the system profits by choosing some less popular combination than c_{max}. This conclusion derives from the following observations:

- Remaining at the current P spares the reindexing process as well as the invalidation of the so far created indices.
- A $*$ subsumes all levels of a dimension's hierarchy, since queries for other levels can be answered from the ALL data stored. For example, for a pivot level $P = \langle H_{11}, * \rangle$ all queries $q = \langle D_{11}, q_2 \rangle$ can be answered (with q_2 being any possible value from any level of dimension 2).

The pivot choice is shaped as follows: The level combinations that lie within *threshold* from c_{max} are considered as *pivot candidates*. More formally,

$$\{\forall c_i \in C,\ 0 \leq i \leq max |\ \textit{popularity}_{c_{max}} - \textit{popularity}_{c_i} < \textit{threshold} \Rightarrow c_i \in C_{cand}\}$$

where C_{cand} is the set of candidate level combinations. The threshold value is proportional to the Mean Difference (Δ) of the popularity values, in particular *threshold*=$k \cdot \Delta$, $k \geq 1$. The parameter Δ, which equals the average absolute difference of two independent values, is chosen as a measure of statistical dispersion:

$$\Delta = \frac{1}{max \cdot (max+1)} \sum_{i=0}^{max} \sum_{j=0}^{max} |c_i - c_j|$$

Among all $c_i \in C_{cand}$, the new pivot level is chosen through the following strategy:

Table 3.2: *Percentage of queries directed towards the 27 level combinations of our initial simulation*

θ	% most popular	% least popular	#combs
0	3.7	3.7	27
0.5	11.1	2.1	27
1.5	44.8	0.3	27
2.5	74.9	0.01	27
3.5	88.8	0.01	12

1. If the current level $P \in C_{cand}$, the system takes no action.

2. Otherwise, from all $c \in C_{cand}$ containing $*$ in one or more dimensions, we consider only combinations that include up to $\lceil \frac{D}{2} \rceil$ ones and exclude the rest. This is in order to ensure that no excessive local processing will be needed for incoming queries. For each of the remaining combinations containing $*$, we recalculate their *popularity* adding the *popularity* of other candidate combinations that are subsumed by it. For instance, let us assume $\langle$`Country`, `Brand`$\rangle$, $\langle$`City`, $*\rangle$ and $\langle *$, `Brand`$\rangle$ are the candidate pivot levels, with *popularities* of 10%, 20% and 15% respectively. Comparing the two levels with $*$, $\langle *$, `Brand`$\rangle$ can answer $\langle$`Country`, `Brand`$\rangle$ queries, thus its *popularity* rises to 25% and is therefore chosen over $\langle$`City`, $*\rangle$ as the new pivot combination.

3. If none of the above holds, the system shifts to the level combination with the highest popularity (c_{max}).

3.4 Experimental Results

We now present a comprehensive simulation-based evaluation of *HiPPIS*. Our performance results are based on a heavily modified version of the FreePastry simulator [fre], although any DHT implementation could be used as a substrate. By default, we assume a network size of 256 nodes, but results are collected with up to 8K nodes. In our simulations, we use synthetically generated data, produced by our own as well as the APB-1 benchmark generator [apb]. In the former case, each dimension is represented as a tree, with each value having a single parent and mul children in the next level. The tuples of the fact table to be stored are created from combinations of the leaf values of each dimension tree plus a randomly generated numerical fact (`sales`). By default, our data comprise of 22k tuples, organized in a 3-dimensional, 3-level hierarchy. The number of distinct values of the top level is $|H_1| = 20$ and mul=2. The initial pivot is, by default, $\langle H_{12}, H_{22}, H_{32} \rangle$. The APB-1 generated datasets are described in the corresponding subsection.

For our query workloads, we consider a two-stage approach: We first identify the probability of querying each level combination according to the *levelDist* distribution; a query is then chosen

from that combination following the *valueDist* distribution. In our experiments, we order the different combinations lexicographically, i.e., combination $\langle H_{13}, H_{21}, H_{31}\rangle > \langle H_{11}, H_{23}, H_{33}\rangle$ and we use the Zipfian distribution for *levelDist* where the number of queries for combination i is proportional to $1/i^{\theta}$. We vary the value of θ as well as the direction of the ordering to control the amount and target of skew of our workloads. For *valueDist* we use the 80/20 rule by default, unless stated otherwise. Table 3.2 gives an overview of the workloads we frequently use in this section. We document the percentage of queries directed towards the most and least popular combination, as well as the number of combinations that receive at least one query (out of the total 27 existing).

Our default workload comprises of 35k queries which arrive at an average rate λ_{query} of 10 queries per simulated time unit. For simplicity reasons we have set the time unit equal to 1 sec, therefore λ_{query} = 10 $\frac{queries}{sec}$. For our experiments, W is set to 50 sec and TTL is given a practically infinite value (indices never expire). Finally,the value of I_{max}, which is heavily data and query-dependent, has been experimented on and set to 2k (see section 3.4.10). This practically means that each node dedicates at most 100KB of memory on soft-state indices.

In this section, we intend to demonstrate the performance and adaptability of *HiPPIS* under various conditions. To that direction, we measure the percentage of queries which are answered directly, i.e., without flooding (*precision*) and we trace the average number of exchanged messages per query, as well as the overhead of control messages needed by our protocol. We compare *HiPPIS* with the naive protocol (referred to as *Naive*), where precision equals the ratio of queries on the initial pivot, and a special case of *HiPPIS*, where only the indices are utilized and no reindexing occurs (referred to as *HiPPIS(N/R)* or plain *N/R*).

3.4.1 Performance with Varying Query Distributions

In this initial set of simulations, we vary the θ parameter for levelDist as well as the direction of skew, using the default parameters otherwise.

In the first graph of Figure 3.7, data are skewed towards the "lowest" level ($\langle H_{13}, H_{23}, H_{33}\rangle$). As θ increases, the workload becomes more skewed and the performance of *HiPPIS* improves: Reindexing is performed sooner, as the ratio of popular queries increases, resulting in a rise of the exact matches due to the chosen combination. For uniform distributions, the number of distinct queries does not allow our method to capitalize on the indexing scheme.

The next graph shows results where our workload favors $\langle H_{11}, H_{21}, H_{31}\rangle$. Again, we notice a similar trend in performance as the values for θ increase. Nevertheless, *HiPPIS* is slightly more effective than before, with its difference from *N/R* increasing as θ increases. This is due to the limited number of distinct values of $\langle H_{11}, H_{21}, H_{31}\rangle$, which facilitates the maintenance of indices, favoring *N/R* against *HiPPIS*. The latter erases all created indices during the reindexing

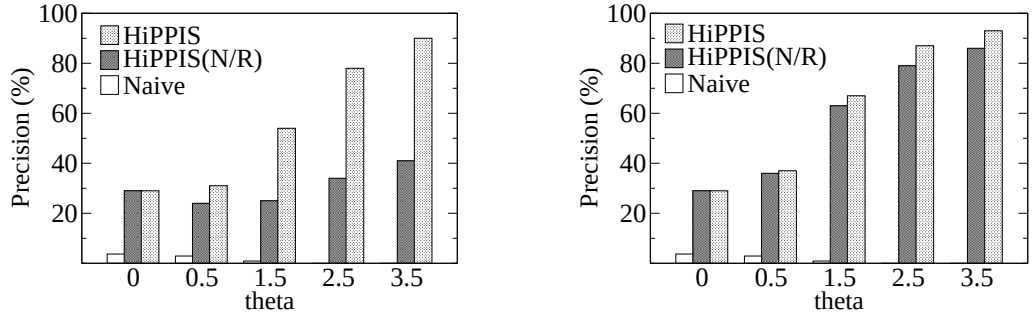


Figure 3.7: *Precision for varying levels of skew (most popular combination is $\langle H_{13}, H_{23}, H_{33}\rangle$ and $\langle H_{11}, H_{21}, H_{31}\rangle$ respectively)*

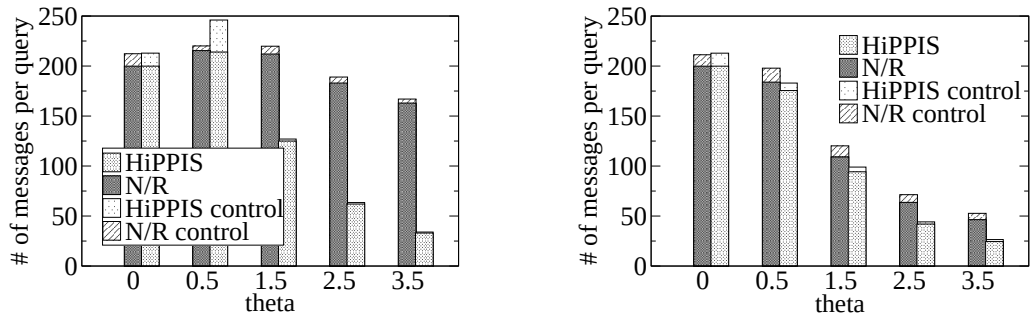


Figure 3.8: *Average number of messages required to answer a query for varying levels of skew (most popular combination is $\langle H_{13}, H_{23}, H_{33}\rangle$ and $\langle H_{11}, H_{21}, H_{31}\rangle$ respectively)*

process. However, *HiPPIS* naturally outperforms its competition in the steady state, as it can increase its performance with time.

Figure 3.8 depicts the number of messages exchanged per query in the system, indicating a measure of bandwidth consumption. Messages regarding query resolution (including requests as well as responses) and control messages, which include those needed to build indices, collect statistics, notify of a reindexing and reinsert tuples, are presented separately. Qualitatively, the total number of messages per query is inversely proportional to the system's precision. As observed in all experiments, the overhead of control messages is small and outweighed by the gains in precision (less than 8% over the total number of messages). This is due to the fact that *HiPPIS* carries out the minimum required reindexing rounds, which translates to one reindexing process per direction of skew. We also notice that the overhead of the control messages decreases as the workload becomes more skewed (almost negligible for $\theta > 1.5$). This can be explained by the fact that *HiPPIS* becomes more confident in the level of reindexing it chooses as θ increases.

Table 3.3: *Statistics for various datasets*

direction of skew	θ value	#global stats	#reind.	#reinsertions	sim. time (sec)	BW (KB)
up	1.5	5	1	11746	5.1	755
up	2.5	4	1	11678	5.5	746
up	3.5	2	1	11521	5.5	730
down	1.5	5	1	16824	4.2	743
down	2.5	4	1	16933	4.4	735
down	3.5	3	1	16701	4.3	740

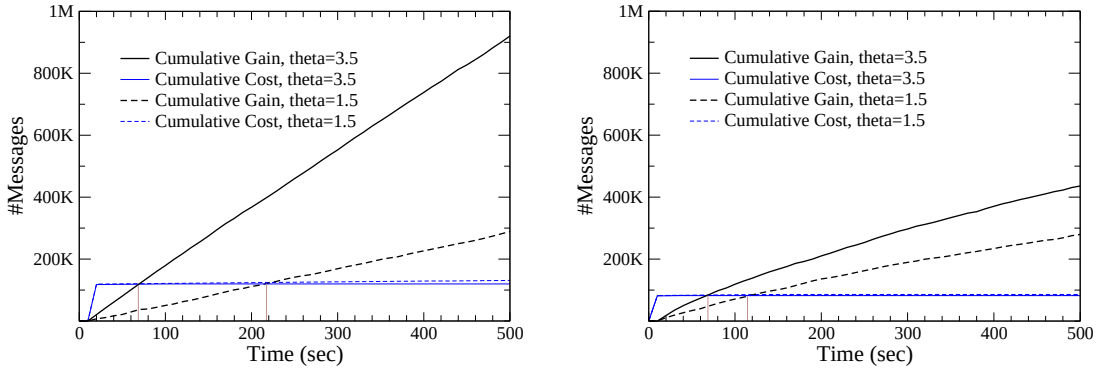


Figure 3.9: *Balance between reindexing cost and gain in messages over time (skew towards* $\langle H_{13}, H_{23}, H_{33}\rangle$ *and* $\langle H_{11}, H_{21}, H_{31}\rangle$ *respectively)*

3.4.2 Reindexing Cost

Table 3.3 presents statistics concerning the reindexing process during the workloads of the previous experiment. The workloads directed towards $\langle H_{13}, H_{23}, H_{33}\rangle$ are denoted as *down* and the ones towards $\langle H_{11}, H_{21}, H_{31}\rangle$ as *up*. As aforementioned, the cost of reindexing is non-negligible. Hence, it is very important that the system performs the minimum required reindexing rounds. *HiPPIS* proves extremely efficient to that end: Only one reindexing process is carried out per direction of skew and less than 5 *SendStats* requests are produced per simulation, thanks to the *interval* selection strategy presented in Section 3.3.4. Thus, our method makes near-optimal use of its bandwidth-intensive operations. It is also worth noting that reindexings towards the lowest hierarchy levels cause more reinsertions than those directed towards the upper ones. This is due to the fact that the dataset used has a limited number of tuples. For a large number of tuples, reinsertions for all possible pivots converge to N^2. However, the total consumed bandwidth (denoted as *BW*) remains the same regardless the skew and its direction, since, in all cases, the initial dataset is reinserted. The time measurements may not adequately reflect reality due to the fact that the experimental evaluation is based on a simulation rather than a real system deployment. In a real system of N nodes, we would expect a significant acceleration in computation (almost N-fold) and a communication cost depending on the topology of the underlying network.

Trying to identify the circumstances under which our system benefits from the reindexing process, we plot the *Cumulative Gain* and the *Cumulative Cost* of reindexing in messages for various datasets and workloads (Figure 3.9). By *Cumulative Gain* we signify the total number of messages spared when using *HiPPIS* instead of *N/R* and in *Cumulative Cost* we include messages for global statistics collection, locking and reinsertion of tuples. Our first observation is that the *Cumulative Gain* increases more rapidly with the increase in skew. This is natural, since highly skewed workloads translate to bigger differences between the most popular and the rest level combinations. Moreover, the workloads directed towards $\langle H_{13}, H_{23}, H_{33}\rangle$ exhibit a higher increase rate in *Cumulative Gain* compared to the ones towards $\langle H_{11}, H_{21}, H_{31}\rangle$. This is due to the fact that the soft-state indexing mechanism of *N/R* is more effective in the latter case (less distinct values for the specific level combination). However, for highly biased workloads, regardless the direction of skew, our system manages to outweigh the reindexing cost in less than 100 sec.

The results of this experiment conform to the conclusions derived by our cost benefit analysis of Section 3.3.3: The highest the workload skew, the more quickly *HiPPIS* starts gaining benefit from a reindexing.

3.4.3 Storage and Load Distribution

This set of experiments aims to evaluate *HiPPIS* in terms of storage and load distribution among the participating network nodes. Using the default dataset, we utilize three of the workloads generated for the previous experiments, the one with levelDist of $\theta = 0$ (denoted as *uni*), and the ones with $\theta = 3.5$, directed towards $\langle H_{11}, H_{21}, H_{31}\rangle$ and $\langle H_{13}, H_{23}, H_{33}\rangle$ (denoted as *up* and *down* respectively).

Figure 3.10 depicts the space dedicated by each node for storing the actual data (in the form of the forest-like structures presented in Section 3.2.2) after the end of the simulation. The measured quantities for each of the 256 nodes are sorted in ascending order. After the necessary reindexings have occurred, the final pivot level combinations are $\langle H_{12}, H_{22}, H_{32}\rangle$, $\langle H_{11}, H_{21}, H_{31}\rangle$ and $\langle H_{13}, H_{23}, H_{33}\rangle$ for *uni*, *up* and *down* respectively. The more numerous the different values of P, the more balanced is the storage distribution among the nodes. In the case of the *down* workload, the majority of nodes host similar quantities of storage space. However, even for the *up* workload, no major differences are documented, since the number of different value combinations is still much larger than the size of the network. This leaves the fairness of the distribution mainly on the hash function.

P affects the total disk space needed to store the distributed data structure: The same dataset requires more space when stored under $\langle H_{13}, H_{23}, H_{33}\rangle$ than under $\langle H_{11}, H_{21}, H_{31}\rangle$. This is due to the fact that a P close to the root of the forest eliminates redundancies in all levels lying above

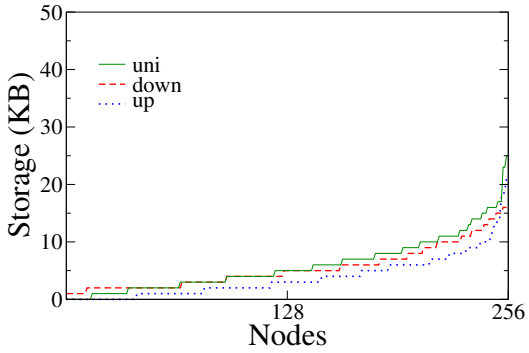


Figure 3.10: *Storage distribution over the network nodes*

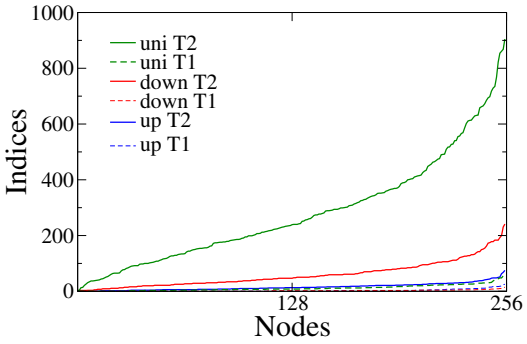


Figure 3.11: *Index distribution over the network nodes*

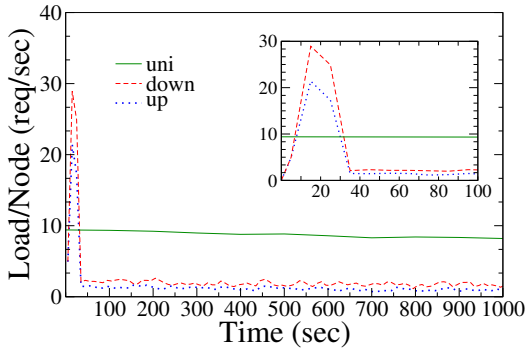


Figure 3.12: *Average load per network node over time*

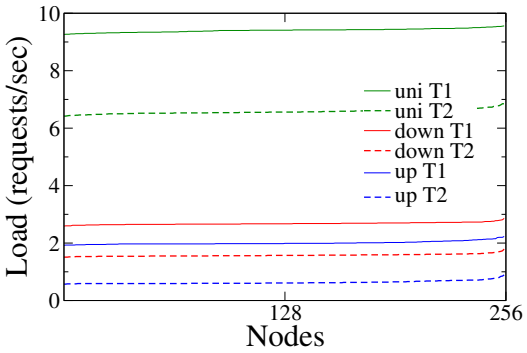


Figure 3.13: *Load distribution over the network nodes*

it in the hierarchy. This can be clarified by observing Figure 3.2(c) and 3.6. While the value *Athens* is stored just once for ⟨`city`, `category`⟩, it needs to be stored 3 times for ⟨`city`, `brand`⟩.

Figure 3.11 shows the amount of indices stored by each network node in ascending order at two distinct points in time, at T_1 = 100 sec and T_2 = 3000 sec. T_1 corresponds to an initial point before any reindexing has occurred, whereas T_2 to a moment close to the end of the simulation. In all cases (except *uni*), indices are distributed pretty evenly among the nodes, with more skewed loads registering the most balanced results. *Uni* exhibits a remarkable increase in indices over time (almost 18 times as many indices in T_2 than in T_1). Since the queries can contain any value with equal probability and no reindexing is performed, very few queries are being repeated and thus indices are constantly being built. The smallest increase is documented for the *up* workload, since it is the workload with the least possible value combinations that can be queried.

Figure 3.12 depicts the average number of messages per second handled by each network node over time, including control messages. For a uniform workload, the simulation starts with an average load almost as high as the query arrival rate, since the majority of the queries are

Table 3.4: *Statistics for various network sizes*

#nodes	#global stats	#reind.	avg.load/node (msg/s)	reind.load/node (msg/s)	#msg/query	precision (%)
128	5	1	2.2	95	28	84.7
256	4	1	2.0	54	51	84.4
512	4	1	1.9	31	97	83.9
1024	2	1	1.8	19	182	83.5
2048	2	1	1.7	12	348	83.3
4096	2	1	1.6	9	655	83.3
8192	2	1	1.6	6	1202	83.3

Table 3.5: *Statistics for various dataset sizes*

#tuples	#global stats	#reind.	avg.load/node (msg/s)	reind.load/node (msg/s)	BW (MB)	#msg/ query	precision (%)
100K	3	1	2	204	4	52	83
1M	4	1	2	235	40	49	83
10M	3	1	2	254	400	50	83
100M	3	1	2	255	4000	49	83

answered though flooding. As time progresses and indices are being created, this measure decreases almost linearly. For the skewed workloads, we observe a spike in load shortly after the simulation starts (see embedded graph). This is due to the reindexing process and mainly to the reinsertion of the dataset according to the new P. However, the average load per node remains within acceptable limits (less than 30 msg/sec) and can easily be handled by the network nodes. Moreover, the decrease is more abrupt during the first 100 sec, while after that point, no significant improvement is documented. This can be explained by the fact that reindexing occurs very quickly, leaving little room for refinement through index creation. Finally, as seen in Figure 3.13, the individual load (sorted in ascending order) is very evenly distributed among the network nodes at all times.

3.4.4 Scaling the Network and Dataset Size

In this set of experiments we aim to examine how well our system scales with regard to the number of participating nodes and the number of tuples in the dataset. First, having inserted the default dataset, we vary the network size from 128 to 8192 nodes. We believe that for a data warehousing application, a system consisting of 8K nodes is an already exaggerated scenario. Also, we vary the dataset size from 100k to 100M tuples and insert it in our default system of 256 nodes. In all cases we pose workloads with levelDist of $\theta = 3$, directed towards $\langle H_{13}, H_{23}, H_{33} \rangle$.

As Table 3.4 proves, *HiPPIS* manages to maintain a steadily high precision, performing only one reindexing and collecting global statistics less than 5 times throughout the simulation, regardless of the network size. Of course, the average number of messages required to resolve a query increases with the increase in network nodes, as floodings become more costly. However, this number is scattered over the network nodes, resulting in a decreasing average load per node. The same is true for the load caused by the reindexing process, since the number of reinsertions remains the same in all cases due to the dataset size ($|D| < N^2$).

When the number of tuples increases by orders of magnitude, only the bandwidth consumed during the reindexing process shows a proportional increase, due to the reinsertion of the dataset. The rest of the statistics presented in Table 3.5 remain stable: Invariably high precision, steady average load per node and number of messages per query and a reindexing load per node converging to N.

3.4.5 The Effect of Recurring Queries

We plan to identify the effectiveness of our system's indexing mechanism under workloads with varying ratio of recurring queries. We believe that this will be the case for the majority of workloads for our target applications, with users temporarily interested in a small number (or set) of (aggregate) data. We consider two different scenarios for the distribution of the duplicate queries. In the first case, for two levels of skew ($\theta = \{1.0, 3.0\}$), we vary the percentage of unique queries by increasing duplicate ones, following the same distribution. In the second case, for three different values of θ for *levelDist*, namely 0.0, 1.0 and 3.0, the *valueDist* distribution varies from uniform to 99/1, creating within each level combination the same amount of skew. The documented precision for both cases is depicted in Figures 3.14 and 3.15 respectively.

In both cases we notice that, as more queries recur in the workload, the performance increases. In the first case, recurring queries follow the *levelDist* distribution, meaning that duplicate queries primarily concern the most popular level combinations. Since *HiPPIS* reindexes to the most beneficial level, it naturally increases its exact answers compared to *N/R*. Nevertheless, the gains decrease as replication increases, unlike *N/R*, which shows almost linear improvement. This is due to the fact that there exists less room for *HiPPIS* to take advantage of the indexed queries, since it has already moved to the best P which takes up significantly more requests. As θ increases, we normally expect an increase in performance.

In the second case (Figure 3.15), as the bias of queried values within a level combination increases, we observe that our system benefits even more from the soft-state indices, exploited by duplicate queries. Small replication in queries results in significant differences in precision for the various θ values, as for these kinds of workloads precision is dominated by exact matches.

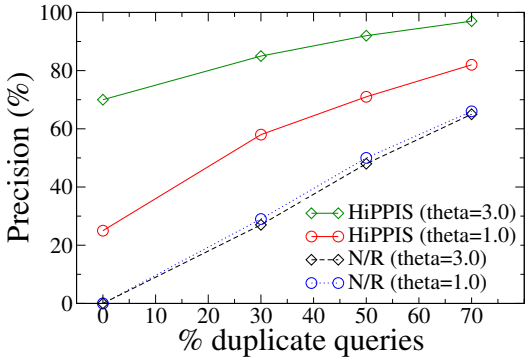


Figure 3.14: *Precision over variable percentage of duplicate queries*

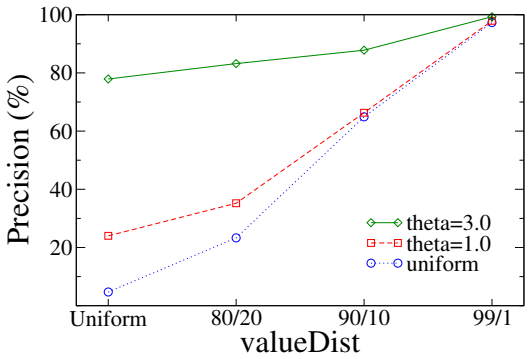


Figure 3.15: *Precision for varying distributions of* valueDist

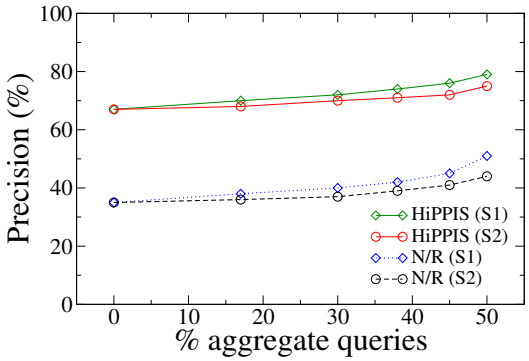


Figure 3.16: *Precision over variable number and skew of aggregate queries*

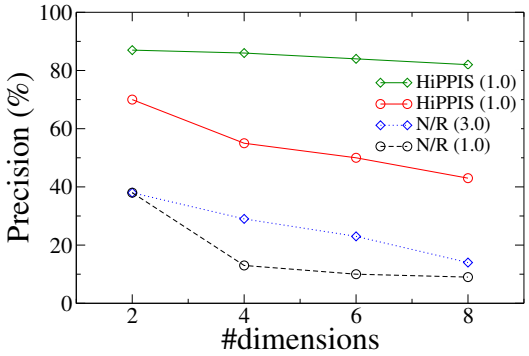


Figure 3.17: *Precision over variable dimensionality datasets*

Nevertheless, all three distributions seem to converge to very high precision levels as the ratio of duplicate queries augments.

3.4.6 The Effect of Aggregate Queries

In this experiment we intend to examine how our system behaves when we inject an increasing number of aggregate queries (from zero up to 50% of the total number of queries). We assume two different distributions as to how $*$ are distributed in those queries: In the first scenario (S1), a $*$ appears in the three dimensions with probabilities (0.73, 0.18, 0.09) respectively (i.e., we heavily favor an aggregate view on the first dimension). In the second one (S2), each dimension is given an equal probability. The workload skew is set to $\theta = 2.0$. Results are presented in Figure 3.16.

We notice that both methods increase in performance as the percentage of aggregate queries increases in both distributions. This is due to the fact that the different combinations that these queries can produce are less than those of point queries. Therefore, increasing their ratio enables the indexing mechanism to store and answer a larger amount of requests without flooding. This

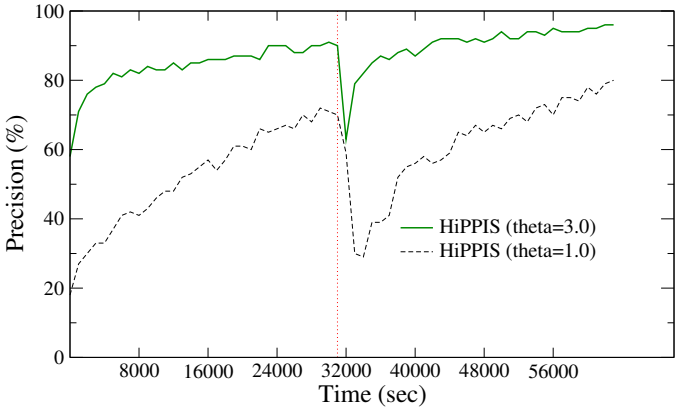


Figure 3.18: *Precision over time for various workloads when a sudden shift in skew occurs in* $t_c = 31000sec$

is evident from *N/R*'s precision increase. In the latter case, the reindexing process invalidates all created indices, thus mitigating the beneficial effect we described before. Furthermore, the skew in the star distribution affects, although slightly, the system's precision. Greater skew leads to greater probability of duplicate queries, favoring the indexing mechanism. Since *HiPPIS* is less dependent on this mechanism, the increase in precision is less noticeable than in the case of *HiPPIS(N/R)*.

3.4.7 Performance in Dynamic Environments

In the next experiment, we measure the performance and adaptivity of *HiPPIS* in dynamic environments, namely sudden changes in the workload. We tailor our query distribution so that a sudden change occurs in the middle of the simulation ($t_c = 31000sec$): From a skewed workload towards $\langle H_{13}, H_{23}, H_{33}\rangle$ we shift to a skewed load towards $\langle H_{11}, H_{21}, H_{31}\rangle$. We show the results for two levels of skew in Figure 3.18.

Our results show that, in all cases, *HiPPIS* quickly increases its precision due to the combination of automatic reindexing and soft-state indices. Floodings increase after t_c, since neither the pivot combination nor the so far created indices can efficiently serve queries with different direction of skew (hence the decline in precision). However, it quickly manages to recover and regain its performance characteristics, as a reindexing is performed and new indices are built. The rate at which these events occur depends on the amount of skew: In the $\theta = 3.0$ case, we show remarkable increase in precision (starting from the plain data insertion at $t = 0sec$), fast recovery after the change in skew and convergence to almost 100% precision. For the less skewed distribution ($\theta = 1.0$), the results record a slight deterioration in the rate of convergence as well as a decline in precision from the change in skew. Once again, we observe that *HiPPIS* performs best in skewed workloads, but its performance in the steady state is invariably high, regardless the workload.

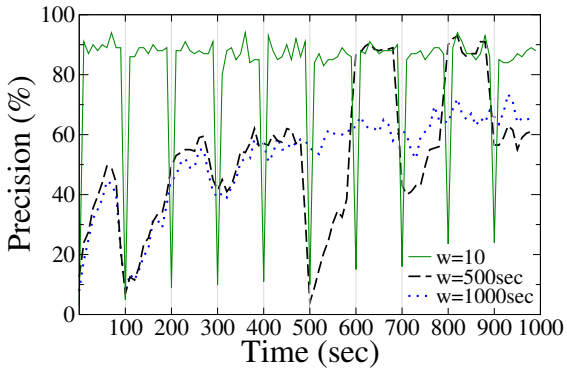


Figure 3.19: *Precision over time when shifts in skew occur every 100 sec*

Table 3.6: *Statistics for various values of* W

W (sec)	#global stats	#reind.	avg.load/node (msg/s)	#msg/query	precision (%)
10	10	10	1061	60	79
500	1	1	645	116	61
1000	0	0	523	133	52

To identify the role of the parameter W in the reindexing decision, we create a workload where the skew changes direction between $\langle H_{13}, H_{23}, H_{33}\rangle$ and $\langle H_{11}, H_{21}, H_{31}\rangle$ every $T_{skew} =$ 100 sec ($\theta = 3.0$).* Figure 3.19 plots the precision of *HiPPIS* over time and Table 3.6 presents various statistics with W ranging from 10 to 1000 sec.

A small W is able to perceive more fine-grained changes in the skew and thus performs frequent reindexings. Indeed, as proven by our experiments, for W sizes smaller than the period of skew change the system performs one reindexing per direction of skew. Moreover, for $W <$ T_{skew}, the smaller its value, the faster *HiPPIS* shifts to the appropriate level combination and the more queries it manages to answer without flooding. On the other hand, as W increases, the changes in skew are less detected and the threshold selection policy is followed: When $W =$ 500 sec a single reindexing occurs, while for $W =$ 1000 sec the system performs no reindexing. As less reindexings occur, the average precision naturally drops with an increase in W. Nevertheless, infrequent reindexings also mean infrequent index invalidations, therefore large W values take better advantage of the soft-state indexing scheme (which would become obvious if the average measurement was over a larger simulation time).

*This is a fairly unrealistic setting for our target application, as we assume that workloads do not change direction of skew this rapidly.

3.4.8 Varying the Number of Dimensions

In this set of simulations we plan to investigate the possible performance variations caused by datasets with variable dimensionality. We assume that each dimension is described by a 3-level hierarchy. By varying the *mul* parameter we try to create equal-size data and query-sets with the same θ value. Figure 3.17 depicts the results for 2 to 8 dimensions for two different values of θ, 1.0 and 3.0.

As the number of dimensions increases linearly, the number of combinations increases exponentially. This radically affects the popular levels' request rates, especially for less skewed workloads, reducing the number of exact match queries for the level combination *HiPPIS* chooses. This becomes obvious when θ increases and the slope becomes more parallel to the dimension axis. *HiPPIS* ranges between 40% and 70% in the low skew case while for bigger skew the precision varies from 80% to 93%. A pure indexing scheme solely relies on the duplicate queries and (to a lesser extent) on the exact match queries of the random pivot level, thus producing poor results.

3.4.9 Updates

In this subsection we focus on the evaluation of the weak consistency update mechanism of *HiPPIS*. Specifically, we run the simulator using the default dataset and two workloads, U_1 and U_2, with skew set to $\theta = 2.0$. U_1 contains exclusively point queries, whereas in U_2, 30% of the workload's queries are aggregate ones. During the simulation, we apply incremental updates at a rate λ_{upd}, that varies from 0.01 to 10 $\frac{updates}{sec}$. The period of the index update procedure is set to 100 sec for all network nodes in all cases. Bearing in mind that the incoming query rate $\lambda_{query} = 10\frac{queries}{sec}$, this translates to queries being posed 1,000 times more frequently than index updates are checked. Incremental updates also occur up to 1,000 times more often than index checking. We measure the percentage of queries whose answers are incomplete, henceforth termed as *inconsistency* and present the results in Table 3.7. Note that a response is considered inconsistent, when at least one record is missing, regardless of the total number of missing records.

Naturally, the faster the update rate, the higher the number of inconsistent answers. However, inconsistency tends to converge as λ_{upd} increases. Even when λ_{upd} is equal to the incoming query rate, meaning that every update is followed by a query, the inconsistency remains in tolerable levels, due to the fact that after the necessary reindexings have occurred, most of the queries are answered directly, without the use of indices. The impact of reindexing is evidently heavier for U_1. Since it does not contain any aggregate queries, the workload is more targeted to the new pivot level values, thus requiring less use of indices. As a result, the inconsistency ratio is noticeably lower than that of U_2.

Table 3.7: *Percentage of inconsistent answers for various λ_u*

λ_{upd}	inconsistency (%)	
(updates/sec)	U_1	U_2
0.01	0.10	1.18
0.1	1.26	5.02
1.0	8.15	18.23
10.0	19.21	20.01

Finally, it is worth noting that when following the weak consistency scheme, the cost of updates is independent of λ_{upd} and equals N messages every period of the index update procedure ($2.56\frac{msg}{sec}$ in this case). On the contrary, a strong consistency scheme would provide 100% accuracy, but require $(\prod_{i=0}^{d} L_i - 1) \cdot \log(N)$ messages per update, resulting in an average rate ranging from 2.16 to 2160 $\frac{msg}{sec}$ for our simulation settings, depending on λ_{upd}. Therefore, for high λ_{upd} the strong consistency scheme should be avoided due to the considerable communication cost it produces.

3.4.10 The Effect of the I_{max} Parameter

The value of the I_{max} parameter is very important as it specifies the maximum number of different non-pivot values that a node can index, and thus defines, as described earlier, the memory requirements of each node. The effect of the I_{max} parameter on the system's precision is examined in this set of experiments, where its value varies from 0 to 3000 for the standard workload, for two levels of skew, $\theta = \{1.0, 3.0\}$, directed towards $\langle H_{13}, H_{23}, H_{33} \rangle$. Results are depicted in Fig. 3.20.

As expected, the system performance improves as I_{max} increases for all workload skews. As the number of indices increases, more queries can be answered using this mechanism. There exists a point I_{thres}, beyond which no significant improvement is observed. The I_{thres} value as well as the documented slope strongly depend on the data and query workloads. For the less skewed workload, the I_{thres} value is larger since more distinct values are requested, thus *HiPPIS* relies more on indices to improve its performance. In the more skewed workloads *HiPPIS* tracks the optimal pivot level sooner and shifts to it, hence less space dedicated to indices is necessary to achieve high performance. Finally, it is worth noticing that the more biased the workload, the lower the performance gains. This is due to the fact that a greater θ value results in more duplicate queries, thus in fewer distinct keys that need to be indexed. The dominant performance mechanism in these cases is the indexing level.

Given this analysis, a value of $I_{max} = 2$k indices is deemed adequate, ensuring that the majority of the created indices will remain in the system. This heavily favors the *N/R* method, since *HiPPIS* discards all indices each time a reindexing occurs. With this value, used in all this

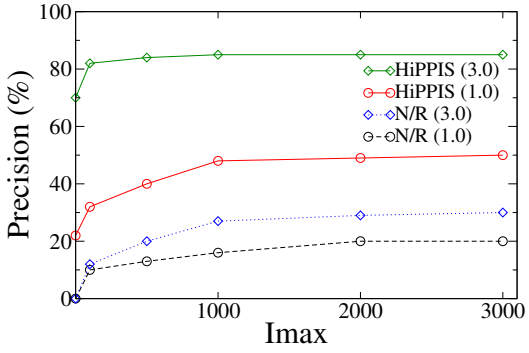


Figure 3.20: *Precision over variable $Imax$ values for both HiPPIS and HiPPIS(N/R)*

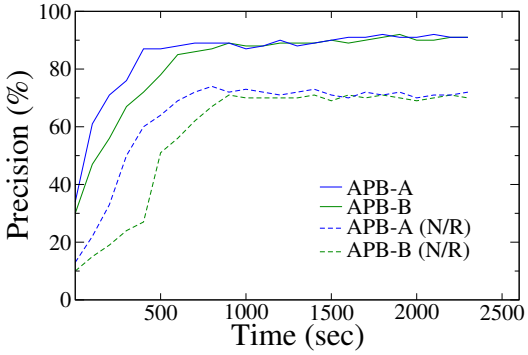


Figure 3.21: *Precision of* HiPPIS *for the APB query workload*

experimental section, each node needs to dedicate at most 100KB of main memory for the soft-state indices.

3.4.11 APB Benchmark Datasets

Finally, we test the performance of *HiPPIS* using some more realistic data and query sets generated by the APB-1 benchmark [apb]. APB-1 creates a database structure with multiple dimensions and generates a set of business operations reflecting basic functionality of OLAP applications. Running the APB-1 data generator with the density parameter set to 0.1 and 1, we produced two 4-dimensional datasets (APB-A and APB-B) with cardinalities 9000, 900, 9 and 24 and two measure attributes. Each dimension comprises of a hierarchy of 7, 4, 2 and 3 levels respectively. APB-A contains 1.2M and APB-B 12M tuples respectively, while the produced workload comprises of 25k queries (queries with $*$ were filtered out from the original query workload) with 1% replication ratio. Results are depicted in Figure 3.21.

We clearly notice that *HiPPIS* exhibits very high performance, reaching over 90% of precision in its steady state after about 4000 queries (400 sec) for APB-A. This experiment shows that for more realistic scenarios, even with more dimensions and levels, *HiPPIS* quickly adapts and serves the vast majority of user requests without flooding. Using plain indices reduces the precision by over 20%, while there is a substantial delay in reaching the steady state (twice as many queries needed).

3.5 Summary

In this chapter we described *HiPPIS*, a distributed system that stores and indexes data organized in hierarchical dimensions for DHT overlays. *HiPPIS*, assuming no prior knowledge of the

workload nor any precomputations, enables online queries on the different dimensions and granularities of the data. Our system dynamically adjusts to the workload by reindexing the stored data according to the incoming queries. With the combination of adaptive indexing and soft-state pointers, *HiPPIS* manages to avoid the network-disastrous flooding in most cases, while enabling both real-time querying and update capabilities on voluminous data. Depending on the needs of the application, *HiPPIS* can also deploy variable consistency update schemes to achieve the desired accuracy in replies without excessive communication overhead.

Our simulations, using a variety of workloads and data distributions, show good performance and bandwidth efficiency. *HiPPIS* is especially effective with skewed workloads, achieving very high precision and fast adaptation to dynamic changes in the direction of skew. Even with few recurring queries, *HiPPIS* manages to answer the majority of queries within $O(\log N)$ steps, by detecting the most popular level combination and shifting to it. Moreover, a significant increase in the number of aggregate queries does not degrade the system's performance, but on the contrary, leads to higher precision. At the same time, the system manages to avoid substantial load imbalance or uneven storage distribution. Finally, adopting a weak consistency update scheme does not significantly degrade the freshness of the responses (less than 20% of the answers are incomplete), even when updates occur as often as queries are posed.

CHAPTER 4

The Brown Dwarf System

In this chapter we present the *Brown Dwarf*, a distributed data analytics system designed to efficiently store, query and update multidimensional data over commodity network nodes, without the use of any proprietary tool. *Brown Dwarf* distributes a centralized indexing structure among peers on-the-fly, reducing cube creation and querying times by enforcing parallelization. Analytical queries as well as updates are naturally performed online through cooperating nodes that form an unstructured P2P overlay. Updates are also performed online, eliminating the usually costly over-night process. Moreover, the system employs an adaptive replication scheme that adjusts to sudden shifts in workload skew as well as network churn by expanding or shrinking the units of the distributed data structure. Our system has been thoroughly evaluated on an actual testbed: It manages to accelerate cube creation up to 5 times and querying up to several tens of times compared to the centralized solution by exploiting the capabilities of the available network nodes working in parallel. It also manages to quickly adapt even after sudden bursts in load and remains unaffected with a considerable fraction of frequent node failures. These advantages are even more apparent for dense and skewed datacubes and workloads.

4.1 Overview

Our goal is to create an on-demand version of a highly efficient data warehousing system, where geographically spanned users, without the use of any proprietary tool, can share and query information. As a motivating scenario, let us consider a business establishment that maintains records

of its operations. These records could well be security, network or system event logs, making the search and analysis of that data an essential part of managing, securing, and auditing how this company's technology infrastructure is used. Instead of creating a centralized data warehouse on-site with a large upfront and maintenance cost, the management chooses to distribute data and computation to possibly multiple location-transparent facilities of commodity nodes and access it more easily and ubiquitously.

To this end, we propose the *Brown Dwarf**, a system that performs online distribution of a centralized warehousing structure (*Dwarf* [SDRK02]) over network hosts in a way that all queries that were originally answered through the centralized structure are now distributed over an unstructured P2P network of commodity nodes.

Dwarf is an approach to compute, index and query large volumes of multidimensional data. While it offers many advantages, like data compression and efficiency in answering aggregate queries, it exhibits certain limitations that prohibit its use as a solution for our motivating problem. Besides the lack of fault-tolerance and decentralization, a *Dwarf* structure may take up orders of magnitude more space than the original tuples [DBS08]. Our *Brown Dwarf* system relaxes these storage requirements and enables the computation of much larger cubes. Moreover, it allows for online updates that can originate from any host that accesses the particular service. Finally, the proposed system can handle significantly larger query rates and actively protect against failing or uncooperative peers, as it offers multiple entry points and adaptive replication of the most loaded parts of the cube.

In summary, the contributions of the proposed system are the following:

- A complete indexing, query processing and update system for data cubes over a distributed environment. The cube is created with just one pass over the data, while updates are processed online. Commodity PCs can participate in this distributed data store, while users need no proprietary tool to access it.

- A robust and efficient adaptive replication scheme, perceptive both to workload skew as well as node churn using only local load measurements and overlay knowledge.

- A thorough validation of the proposed system using an actual deployment. Our findings show that *Brown Dwarf* can be as much as 5 times faster in creating the cube and 60 times faster in querying it compared to the centralized version. Moreover, it offers a fair distribution of the original dwarf and the induced query load at low cost (shared among many nodes); it shows impressively prompt adaptation to query skew and proves resilient to a considerable fraction of node failures even with low replication ratio.

*A *brown dwarf* is an object which has a size between that of a giant planet and that of a small star. It is possible that a non-negligible portion of the mass in the Universe is in the form of brown dwarfs.

Table 4.1: *A sample fact table with three dimensions and one measure of interest*

DIM1	DIM2	DIM3	Measure
S_1	C_2	P_2	\$70
S_1	C_3	P_1	\$40
S_2	C_1	P_1	\$90
S_2	C_1	P_2	\$50

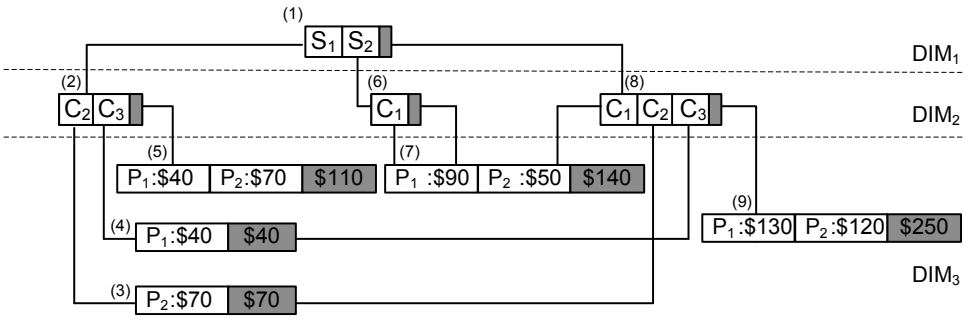


Figure 4.1: *The centralized Dwarf structure for the data of Table 4.1, using the* sum *aggregation function*

4.2 Dwarf and Brown Dwarf

In this section we provide a short description of the original *Dwarf* as well as an overview of *Brown Dwarf*. Presenting this evolution from the centralized to the fully distributed solution, we aim to reveal the reasons that led us to the choice of *Dwarf* as our system's data cube structure and point out the advantages of its distribution.

4.2.1 The Original Dwarf Structure

Dwarf [SDRK02] is a complete architecture for computing, storing, indexing, querying and updating both fully and partially materialized data cubes. *Dwarf*'s main advantage is the fact that it eliminates *both* prefix and suffix redundancies among the dimension values of multiple views.

Prefix redundancy happens when a value of a single or multiple dimensions occurs in multiple group-bys (and possibly many times in each group-by). For example, for the data in Table 4.1, the value S_2 appears in $\langle S_2, C_1, P_1\rangle$, $\langle S_2, C_1, P_2\rangle$ but also in many group-bys (e.g., $\langle S_2, C_1\rangle$, $\langle S_2\rangle$, etc.). On the other hand, suffix redundancy occurs when some group-bys share a common suffix. For example, we can see in Table 4.1 that the $\langle C_1, x\rangle$ group-by has the same value as the $\langle S_2, C_1, x\rangle$ one, with x being any value in the third dimension. Prefix redundancies are met in dense areas of the cube while suffix redundancies are considerable in sparse areas. Both have a significant effect on the size and computation costs over the cube.

The *Dwarf* construction algorithm employs a top-down computation strategy for the data cube, which automatically discovers and eliminates all prefix and suffix redundancies on a given

dataset. This elimination happens prior to the computation of the redundant values. As a result, not only is the size of the cube reduced, but its computation is also accelerated.

To better understand how *Dwarf* indexes the dataset and uses its properties to answer queries, we show in Figure 4.1 the cube created by this algorithm for the fact table of Table 4.1. The structure is divided in as many levels as the number of dimensions. The root node contains all distinct values of the first dimension. Each cell value points to a node in the next level that contains all the distinct values that are associated with its value. Grey cells correspond to *ALL* values of that cell, used for aggregates on each dimension. Any group-by can be realized through traversing the structure and following the query attributes, leading to a *leaf* node with the answer. For example, $\langle S_1, C_3, P_1 \rangle$ will return the \$40 value, while $\langle S_2, ALL, ALL \rangle$ will return the aggregate value \$140 following the nodes (1)→(6)→(7).

4.2.2 The Brown Dwarf Outline

Brown Dwarf (or *BD* for short) is a system that distributes *Dwarf* over a network of interconnected nodes†. The goal is to have the ease of constructing, querying and updating this structure in an online fashion over a network overlay instead of a central location.

The *BD* construction algorithm distributes dwarf nodes to network hosts on-the-fly, as tuples are parsed in a single pass. Pictorially, Figure 4.2 shows that nodes (1) through (9) are selected in this order to store the corresponding dwarf nodes of Figure 4.1. These nodes form an unstructured P_2P overlay, using the indexing induced by the centralized creation algorithm. Queries and updates are then naturally handled using the same path that would be utilized in *Dwarf*, with overlay links now being followed: If the incoming query asks about S_1 it will be forwarded to node (2). From there, depending on the requested group-by (ALL, C_2 or C_3), terminal nodes (3), (4) or (5) can be visited.

Compared to the traditional *Dwarf* (or other centralized indexing methods used in data warehousing), *BD* offers the following advantages:

- The existence of more than one hosting nodes offers the ability to parallelize the cube construction, querying and update process.
- The distribution of the structure enables the computation and storage of much larger cubes.
- *BD* allows for online updates that can originate from any host accessing the update service of the system.

†We will be using the terms *node* and *peer* interchangeably to refer to the computational elements used for cube storage and processing.

- The proposed system can handle significantly larger rate of requests without having to replicate the whole structure, as it offers multiple entry points and adaptive replication of the most loaded parts of the cube.

4.3 The Brown Dwarf System Design

The essence of *BD* is the distribution of the original, centralized structure over the nodes of an unstructured overlay in a way that guarantees equal storage and bandwidth consumption as well as query processing efficiency, even under node churn. While structured P2P overlays provide efficient lookup operations for (key, value) pairs, they do not directly support the storage of more complex data structures. Moreover, they require full control over the induced topology as well as the storage distribution, thus proving unfitting for many realistic scenarios. Contrarily, unstructured overlays (e.g., [gnu03]) offer more loose constraints on topology and data management (nodes are responsible for their own repositories), making them particularly appealing for our application. Furthermore, our indexing mechanism guarantees an $O(1)$ lookup operation (see Section 4.3.2), as opposed to the logarithmic cost of a DHT lookup. Lastly, although DHTs inherently provide replication, they do so in a static way and in a per node basis, whereas the replication scheme we propose needs to be fully adaptive to load, guaranteeing minimum redundancy.

The general approach of *BD* is the following: Each vertex of the dwarf graph (henceforth termed as *dwarf node*) is designated with a unique ID (UID) and assigned to an overlay (or network) node. We assume that each network node n is aware of the existence of a number of other network nodes, which form its *Neighbor Set*, NS_n. Adjacent dwarf nodes are stored in adjacent network nodes in the P2P layer by adding overlay links. Thus, each edge of the centralized structure represents a network link between n and a node in NS_n. Each peer maintains a *hint* table, necessary to guide a query from one network node to another until the answer is reached and a *parent* list, required by our replication process, to avoid inconsistencies.

The *hint* table is of the form (*currAttr, child*), where *currAttr* is the current attribute of the query to be resolved and *child* is the UID of the dwarf node the *currAttr* leads to. If the dwarf node containing *currAttr* constitutes a leaf node, *child* is the aggregate value. The parent list contains the UID of the dwarf node's parent node(s) along with the *currAttr*, whose child led to the specific node. In order to route messages among network nodes, each of the peers maintains a routing table that maps UIDs to *NIDs* (i.e., network IDs, e.g., IP address and port).

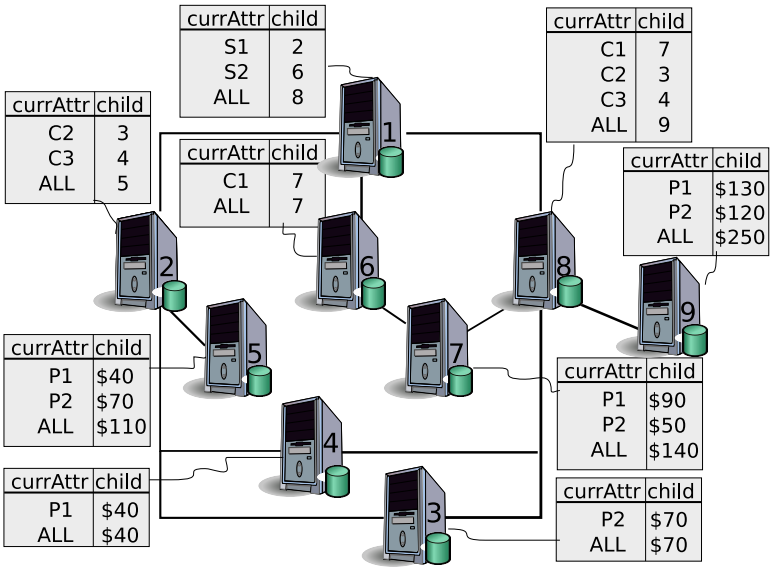


Figure 4.2: *The distribution of the dwarf nodes in the Brown Dwarf of Table 4.1 and their* hint *tables*

4.3.1 Insertion

The creation of the data cube is undertaken by a specific node (*creator*), that has access to the fact table. The creator follows the algorithm of the original dwarf construction, distributing the dwarf nodes on-the-fly during the tuple-by-tuple processing, instead of keeping them in secondary storage. In general, the creation of a cell in the original dwarf corresponds to the insertion of a value under *currAttr* in the hint table. The creation of a dwarf node corresponds to the registration of a value under *child*. Thus, all distinct values of the cells belonging to a dwarf node are eventually registered under *currAttr*. Moreover, the node each *currAttr* points to is kept under the *child* attribute. In the case of a dwarf leaf node, *child* corresponds to the measure or the aggregate value.

Let d be the number of dimensions and $t_1 = \langle a_1, a_2 \ldots a_d \rangle$ be the first tuple of the fact table. Upon processing of t_1, a_1 triggers the creation of the root node, meaning that a network node from the creator's *NS* is allocated (let it be node N_{root}). A new hint table is created and stored in N_{root} under a randomly chosen UID. At this point, only the *currAttr* can be filled in with a_1. Moving to a_2, a new node is allocated from the neighborhood of N_{root} and a new hint table is created following the previous procedure. The UID of the newly allocated node is added to N_{root}'s hint table under a_1. The same procedure is followed by all dimension attributes of t_1 (plus the special *ALL* attribute wherever needed). As tuples of the fact table are being processed one by one, new hint tables are created and existing ones are gradually modified (see Algorithm 4.1).

Note that the proposed insertion mechanism does not entail an a priori creation of the centralized dwarf. Nodes are created and hint tables are filled in gradually, as tuples are processed.

Algorithm 4.1: The BD insertion algorithm

Data: t_{curr}: the tuple to be inserted
t_{pre}: the tuple previously inserted
$N_{creator}$: the node that initiates the insertion
Result: Insertion of dataset

sort the tuples of the fact table;
while *unprocessed tuples exist* **do**
  $t_{cur} \leftarrow$ the next unprocessed tuple of the fact table;
  find at_{last}, the last common attribute of t_{cur} and t_{pre};
  locate the network node N_{last} that hosts the dwarf node that at_{last} leads to;
  add the first uncommon attribute to the hint table of N_{last} under currAttr;
  forall the *remaining attributes of* t_{cur} **do**
    create dwarf node and assign it a UID;
    add UID to the hint table of N_{last};
    randomly pick a node N_{next} from $NS_{N_{last}}$;
    send dwarf node to N_{next};
    $N_{last} \leftarrow N_{next}$;
  end
  beginning bottom up, add the ALL cells and create new dwarf nodes according to the original `SuffixCoalesce` algorithm;
end

The only information the creator needs to hold at each moment is that of d dwarf nodes (the nodes of the path that t_i traverses).

For the first tuple of Table 4.1, the corresponding nodes and cells are created on all levels of the dwarf structure (Figure 4.1). Each of the created nodes (1), (2), (3) are assigned to respective overlay nodes. In the hint table of (1), S_1 is placed under *currAttr* and (2) under *child*. Following the same procedure, the routing table for (2) is filled in with C_2 and (3) and that of (3) with P_2 and \$70 (the measure attribute, since it is a leaf node). Insertion moves on to the next tuple, which shares only prefix S_1 with the previous one. This means that the C_3 value needs to be inserted to the same node as C_2, namely (2), and (4) needs to be allocated. Thus, C_3 must be registered in the node's hint table as a new *currAttr* and (4) as a new child value. Moreover, (3) is now closed, so *ALL* along with the aggregate value \$70 are registered in its hint table. Gradually, all necessary nodes are allocated and their hint tables are filled in with the appropriate routing information (see Figure 4.2).

4.3.2 Query Resolution

Queries are resolved by following their path along the *BD* system attribute by attribute. Each attribute value of the query belongs to a dwarf node which, through its hint table, leads to the network node responsible for the next one.

A node initiating a query $q = \langle q_1, q_2 \dots q_d \rangle$, with q_i being either a value of dimension i or *ALL*, forwards it to N_{root}. There, the hint table is looked up for q_1 under *currAttr*. If it exists, *child* will be the next node the query visits. The above procedure is followed until a measure is reached. Note here that, since adjacent dwarf nodes belong to overlay neighbors, the answer to any point or group-by query is discovered within at most d hops. A DHT-like solution would require an average of $\log N$ steps for each dwarf node discovery (N being the size of the network), producing an average of $d \log N$ overlay hops for a query resolution.

From the above description, it is clear that the system requires an entry point, meaning that query initiators should be aware of N_{root}, where the resolution of any query starts from. This can be achieved through an advertising mechanism, invoked by N_{root} upon allocation. The existence of a single entry-point for *BD*, which constitutes a single point of failure, is tackled by our replication strategy, thoroughly described in the following sections.

Back to our example, let us consider the query $S_1 ALL P_2$. Beginning the search from (1), and consulting the child value corresponding to S_1, we end up at (2). There, since the second dimension value is ALL, the query follows the path indicated by the third entry of the hint table, thus visiting (5). P_2, the third dimension value, narrows the possible options down to the second entry of the hint table, namely \$70.

4.3.3 Incremental Updates

The procedure of incremental updates is similar to the insertion process, only now the longest common prefix between the new tuple and existing ones must be discovered following overlay links. Once the network node that stores the last common attribute is discovered, underlying nodes are recursively updated. This means that nodes are expanded to accommodate new cells for new attribute values and that new dwarf nodes are allocated when necessary. Moreover, the insertion of new tuple to an existing *BD* affects the ALL cells of dwarf nodes associated with the updated nodes.

Assuming $u = \langle u_1, u_2 \dots u_d \rangle$ is the tuple to be added to an existing *BD*, the incremental update procedure starts from the root of the structure following the path designated by u_1, u_2 etc. Once the dwarf node containing the last attribute u_i that is already present is discovered, a new entry for u_{i+1} must be registered to the node where the child of u_i points to. The following attributes $(u_{i+2} \dots u_d)$ will trigger the creation of new dwarf nodes. The special ALL cells are recursively updated for all nodes affected by the change.

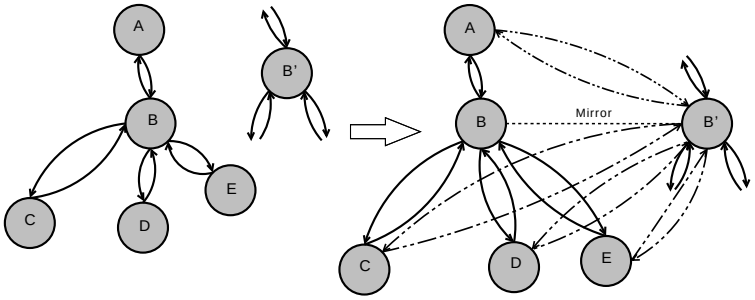


Figure 4.3: *Example of mirroring*

It is obvious that the update procedure is bandwidth-costly, therefore we assume it is invoked when several batches of updates are collected.

4.3.4 Mirroring

In order to ensure availability and eliminate single points of failure, especially in the case of N_{root}, that represents the single entry point for queries in our system, we assume a global replication parameter k. This parameter defines the degree of data redundancy in the system: During the insertion phase, each dwarf node is stored in $k+1$ network nodes instead of just one. Thus, in its initial state, the system hosts $k+1$ instances (*mirrors*) of each dwarf node. The query forwarding algorithm is now amended: A node that receives a query randomly chooses from a list of mirrors the one to forward the query to.

To achieve correct behavior after a mirroring operation, the parent, children and mirror nodes of the original node(s) must be informed of this creation: Parents must know of the new node in order to include it in the forwarding process. The mirror node must be informed of the peers that it precedes according to *BD* as well as its parent(s). Finally, the children must be informed of this new parent. Figure 4.3 describes this process pictorially: Node B chooses to replicate one of its dwarf nodes to B', invoking `mirror` (Algorithm 4.2). B' receives the hint table of B regarding the specific dwarf node and is thus informed of A as well as C, D and E, creating overlay links to them accordingly (parent and children). Node A is informed of the new child, adding B' to its list of children for this dwarf node. Moreover, parent links are also updated (shown in dotted lines). With the exchange of hint tables between B and B', the latter is able to discover all other mirrors and notify them of its existence.

From that point on, the system is responsible for preserving the number of each dwarf nodes' *mirrors* above k. To validate their (un)availability, *mirrors* periodically ping each other. This is also achieved through normal queries, when forwarding to a node that fails to reply. If a network node perceives that the *mirror* of a dwarf node it hosts is unavailable (because of a network failure for example), it initiates the mirroring operation for that dwarf node through the invocation of

Algorithm 4.2: The BD `mirror` algorithm

Data: dn_s: dwarf node to be replicated
Result: Replication of dn_s

N: network node that hosts dn_s;
N sends hint table of dn_s to $N' \in NS_N$;
N' informs parents and mirrors of dn_s;

the `mirror` function: The node chooses another peer in order to replicate the dwarf node to it. The node to receive the replica can be chosen either from the pool of locally known peers or using another independent service. It is important to note here that the whole process does not affect the behavior of the system, as all queries continue to be normally resolved.

4.3.5 Handling Node Failures and Query Skew

A basic requirement for every distributed application is fault-tolerance. Node clusters typically consist of commodity, failure-prone hardware. Especially in the case of data analysis, where complex workloads can take hours to complete, the probability of a failure occurring becomes higher. Apart from node churn, data skew is another factor that stresses the system's ability to operate smoothly, as it can degrade the performance of overloaded nodes, having a disproportionate effect on total query latency. Data replication techniques are commonly utilized in distributed systems in order to remedy these situations. Replicating critical or frequently accessed system resources is a well-known technique utilized in many areas of computer science (distributed systems, databases, file-systems, etc.) in order to achieve reliability, fault-tolerance and increased performance.

In *BD*, we utilize a replication scheme adaptive to skew as well as node churn to address both issues in a unified way, *expanding* popular or unavailable elements of the structure and *shrinking* others that receive few requests. This way, *BD* is able to obtain increased resources to handle spikes in load and release them once the spike has subsided.

4.3.6 Node Churn

When a node wishes to leave the system, it initiates the `graceful-depart` function, given in Algorithm 4.3: For each of the dwarf nodes it stores, the respective parents and children are notified to revise their links. The same is true for the list of mirror nodes. Messages can be grouped per recipient since, in the majority of cases, we anticipate that some nodes will be parents for multiple dwarf nodes or both parents and children.

Given this process, we may now describe how random departures or node failures are handled: A departed node is discovered either through the periodic ping procedure, or through

Algorithm 4.3: The BD `gracefull-depart` algorithm

Data: dn_s: dwarf node to be deleted
Result: Deletion of dn_s

N: network node that initiates deletion;
N informs parents, children and mirrors of dn_s;
N deletes hint table of dn_s;

query routing. In the first case, the mirror node receives no acknowledgement from the departed node within a period of time, considering it unavailable. It then initiates `graceful-depart` with the failed node's NID as a parameter. In the second case, the departed node is, at some point, selected during the routing process. The parent node will be informed of this failure after a timeout occurs. The parent then forwards the current query along with a `graceful-depart` request to another child, mirror of the departed node. Besides processing the query, this mirror initiates `graceful-depart` with the failed node's NID as a parameter. In the end, all of the failed node's parents, children and mirrors will be notified of this event and update their links.

4.3.7 Load-driven Mirroring

In *BD*, network nodes utilize adaptive mirroring according to the load received on a per-dwarf node basis. A network node hosting an overloaded dwarf node can create additional mirrors through the *expansion* process. The node to receive the new mirror can be chosen from the node's *NS* either randomly, or following some more advanced policy, which takes into account parameters like storage, utilization and load. Such a policy could, for instance, dictate the selection of the most underloaded peer, or the peer with the largest amount of free disk space. The newly created mirror will be used by the parent node(s) in order to receive some of the requests. In the opposite case, an underloaded dwarf node can be deleted from the system through the *shrink* process, as long as the total number of its mirrors remains above k. This deletion frees resources that could be allocated for more "popular" parts of the structure.

These procedures require that each peer participating in *BD* monitors the incoming load for each of the dwarf nodes dn_s it hosts. Let $l_s(t)$ be the current load for dn_s. The two procedures can be described as follows:

`Expansion`: As the load increases due to the incoming requests, some dwarf nodes reach their self-imposed limits, which we assume are expressed by the parameter $Limit^s_{exp}$: It represents the maximum number of requests that dwarf node dn_s can process per time unit. When this limit is exceeded, the hosting node invokes `mirror` to replicate it according to its demand and relative to $Limit^s_{exp}$. Specifically, each node dn_s, with $l_s(t) > Limit^s_{exp}$ will be replicated $\lceil l_s(t)/Limit^s_{exp} \rceil$

Algorithm 4.4: The BD adaptive mirroring algorithm

Data: DN_N: the set of dwarf nodes that network node N hosts
k: the replication degree
r_s: the number of mirrors for dn_s

forall the $dn_s \in DN_N$ **do**
  $l_s(t)$ the current load for dn_s;
  if $l_s(t) > Limit^s_{exp} \vee r_s < k$ **then**
    for $i = 1$ *to* $\max(\lceil l_s(t)/Limit^s_{exp} \rceil, k - r_s)$ **do**
      `mirror` (dn_s)
    end
  end
  else if $l_s(t) < Limit^s_{shr} \wedge r_s > k + 1$ **then**
    `gracefull-depart` (dn_s)
  end
end

times. This mechanism allows for adaptive expansion of the network-wide storage according to demand and helps overloaded nodes to offload part of their workload to other server instances.

`Shrink`: Temporal changes in workload may result in the creation of mirror nodes which eventually become underutilized. The system should be able to delete such nodes, provided that their deletion will not result in less than $k + 1$ mirrors. Assuming $Limit^s_{shr}$ is the limit, under which dn_s is considered underloaded and r_s is the number of the mirrors of dn_s that the storing node is aware of, then each dn_s with $l_s(t) < Limit^s_{shr}$ and $r_s > k + 1$ will be deleted, through `graceful-depart`. To ensure that the deletion of dn_s will not cause the overloading of its mirrors, we estimate $Limit^s_{shr}$ using the following rationale: When deleting a replica, we get from r_s to $r_s - 1$ mirrors. Estimating the total load for dn_s to be $r_s l_s(t)$, we require that $l_s(t') \simeq r_s l_s(t)/(r_s - 1) < Limit^s_{exp}$. Thus we choose $Limit^s_{shr} = Limit^s_{exp} \cdot r_s/(r_s + c)$, where c is a positive constant.

In essence, $Limit_{exp}$ and $Limit_{shr}$, relative to the context and values that are assigned to them, implement the application's policy with respect to the quality of service: Indeed, they regulate how reactive (and thus query-efficient) we want the application to be at the cost of more or less storage and data transfers. A more formal description of the adaptive mirroring algorithm can be seen in Algorithm 4.4.

4.4 Optimizations - Discussion

In this section we suggest various optimizations and discuss issues concerning the data consistency in the event of mirroring as well as the viability of such an application in the Cloud.

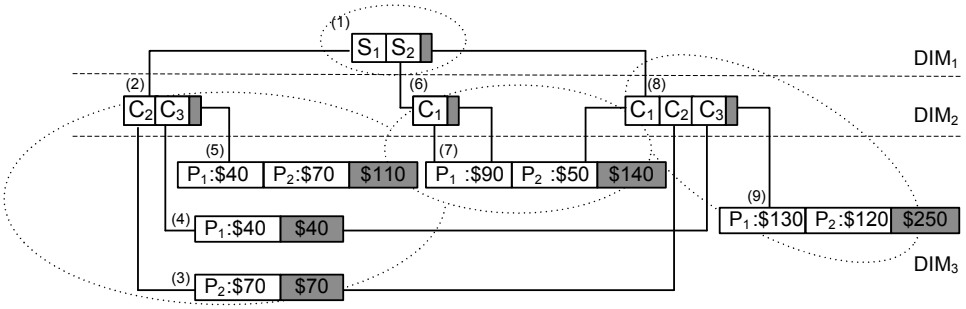


Figure 4.4: *The dwarf structure of Table 4.1, where nodes are grouped with* $h = 2$

4.4.1 Query Performance Optimization

In general, we expect the number of dwarf nodes created to be much larger than the number of participating hosts, with each peer hosting multiple dwarf nodes. This fact, along with the load-driven expansion and shrink of the system may result in individual peers hosting dwarf nodes that are connected in the logical level (a dwarf node and some of its children's mirrors). In this case, a query resolution that follows a random path from a dwarf node to one of its children may result in unnecessary message exchange between peers. Alternatively, upon reception of a query, the peer can choose from the list of children the one that resides in the same node, thus reducing the network messages needed.

4.4.2 Dimension Grouping Optimization

This optimization intends to reduce the communication cost between network nodes during all the operations of *BD*. Instead of storing the data structure at the dwarf node granularity, the system chooses to group related dwarf nodes and store them together as an entity. Due to the lack of a priori knowledge of the graph (since it is constructed and distributed on the fly) and the distributed nature of our system, we employ a simple heuristic: Starting from the leaf nodes of the structure, we create groups that contain sub-graphs of height h, $h < d$. This practically means that a dwarf node dn and all its descendants of depth h reside at the same physical node. Nevertheless, this might not be the case for dwarf nodes along an aggregate path, since an ALL cell might point to an existing dwarf node that belongs to a different group. Figure 4.4 depicts the groups for $h = 2$.

Both data load and update procedures are now affected, since a group, rather than an individual dwarf node, is assigned to a network node. During querying, the number of network hops until the answer is reached is significantly reduced. For point queries, the number of hops is $\lceil \frac{d}{h} \rceil$, while for aggregate queries it may range from $\lceil \frac{d}{h} \rceil$ to d at most. Replication takes great benefit

from this enhancement as well, since the memory needed for statistics as well as the overall replication cost is reduced. However, it must be noted that such grouping may also reduce the effect of decentralization and thus the potential of the system to exploit parallelization (relative to the choice of h and the structure of the dwarf graph). This approach is deemed most beneficial for high-dimensional datasets with the value of h set significantly smaller than that of d.

4.4.3 Consistency Issues

The expand/shrink scheme we described presents a fully adaptive and distributed replication method. It adapts to both the direction and the amount of skew, as it replicates overloaded parts of *BD*. It also adapts to node churn by keeping a minimum replication rate for each dwarf node. This is achieved using only local load measurements and overlay knowledge.

Furthermore, the expand/shrink scheme raises some consistency issues relating to the precision of the information that each node has of the available mirrors: Obviously, an almost concurrent creation of two or more mirror nodes or a series of expansions and shrinks at different parts of the overlay may result in nodes with different and incomplete knowledge of both the number and the identities of available mirrors.

Yet, there exist arguments to manifest that these inconsistencies are temporary: First, the higher the query rate that triggers such occurrences, the larger the probability that newly created replicas or unavailable ones are discovered through the query forwarding process. Second, it is reasonable to assume that, for our target applications, we do not expect a high churn rate from participating peers. We may even assume that some nodes (possibly the initial ones) will be more stable, server-like nodes that rarely disconnect.

The latter also justifies our choice of using the `graceful-depart` that notifies *all* children, parent and mirror nodes (regardless of a graceful or not departure): Assuming that the churn rate is not large enough for this process to become both strenuous and costly, we take steps to avoid the delay in query processing (due to time-outs) rather than minimize inter-node communication.

4.4.4 Cloud Deployment Potentials

In data warehousing, the need to keep large volumes of historical data online and ensure their availability and fast access even under heavy workload dictates a continuous investment in hardware, electrical power and software maintenance. The nature of these applications with the large amounts of data and their subsequent costs as well as the common operations involved (which can be easily parallelized across the data sites) make them particularly well-suited for the Cloud.

We believe that our system is a particularly good candidate for deployment in the Cloud, as it provides several required architectural characteristics, such as:

- **Cost-efficiency**: In the Cloud, data volumes and transfers as well as computational costs should be minimized, as billing is always relative to the resource usage. Especially in the case of data warehousing structures, where additional indices and materialized views swell the size of source data, aggressive data compression results in significant savings. Our system exploits the inherent efficient compression that the original *Dwarf* cube offers and combines it with high-performance query processing and updates. Moreover *BD* is able to obtain increased resources to handle spikes in load and release them once the spike has subsided, adding a feature extremely important in a pay-as-you-go environment such as the Cloud.

- **Elasticity**: The shared-nothing architecture, upon which our system is based, permits the platform to scale out, as the Cloud itself does. Most databases popularly used in BI today have shared-everything or shared-storage architectures, which limit their ability to scale in the Cloud. *BD*, however, allocates computational power and storage adaptively according to demand, enabling an easy integration of additional machines with the existing system on the fly, giving the users the impression of infinite resources at their disposal.

- **Fault tolerance**: Clouds typically consist of commodity, failure-prone hardware. Especially in the case of data analysis, where complex workloads can take hours to complete, the probability of a failure occurring becomes higher. It is thus important to ensure that queries will not have to be reissued every time a node fails. *BD* employs a replication strategy that adapts to node failures, ensuring that each data at any given time exists in at least k physical nodes. The attribute by attribute query resolution makes sure that a message directed to a failed node will be redirected to one of its mirrors, resuming the querying process almost instantly and without having to reissue the whole query batch.

- **Content availability**: Within a Cloud-based analytic database cluster, node failures, node changes, and connection disruptions can occur. Given the vast number of processing elements within a Cloud, these failures can be made transparent to the end user if there exist proper built-in failover capabilities. *BD* replicates data automatically across the nodes in the Cloud and is thus able to continue running in the event of multiple node failures (*k-safety*) and capable of restoring data on recovered nodes automatically.

4.5 Experimental Results

We now present a comprehensive evaluation of *BD*, entirely written in Java, using both an actual and a simulation-based testbed. For our local area experiments, we utilize $N = 16$ commodity nodes (Quad Core @ 2.0 GHz, 4GB RAM) to act as the storage/computation infrastructure.

Table 4.2: *Storage requirements and creation time for* Dwarf *and* Brown Dwarf *data cubes of various dimensionalities and distributions*

d	Fact Tbl (MB)	Uniform size (MB)		Uniform time (sec)		80-20 size (MB)		80-20 time (sec)		Zipf size (MB)		Zipf time (sec)	
		Dwarf	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*
5	0.2	1	1	4	4	1	1	8	7	1	1	3	4
10	0.4	4	5	31	10	4	5	28	14	6	7	54	21
15	0.6	7	9	63	29	10	13	96	43	22	27	226	74
20	0.8	13	17	122	55	18	23	352	82	54	69	543	204
25	1.0	18	23	198	88	29	37	729	196	152	195	1206	535

The centralized approach has also been implemented for direct comparison. To examine the scalability of our application as the infrastructure nodes scale to larger numbers, we simulate *BD* using a network simulator with values of N ranging from 64 up to 1k nodes.

In our experiments, we use both synthetic and real datasets consisting of a fact table representing multidimensional data with numerical facts. The synthetic datasets have been generated by our own and the APB-1 benchmark generator [apb]. Our generator creates the tuples of the fact table to be stored from combinations of the different values of each dimension (*cardinality*) plus a randomly generated numerical fact. By changing the number of distinct values per dimension we generate cubes of different density. Furthermore, we may choose to create tuples that combine dimension values uniformly or with bias (creating 80/20, 90/10 and 99/1 self-similar distributions and Zipfian distributions with $\theta = 0.95$). The aggregate function used in the results is `sum`.

For the application workloads, we include both point and aggregate queries with varying proportions and distributions as well as batch updates. We either query the available dimension values uniformly or with skew, following the Zipfian distribution with various θ values.

4.5.1 Cube Creation

In the first set of experiments, we evaluate the creation of the distributed *BD* structure in terms of construction time, storage and communication overhead. We also prove the fairness of the data distribution over various datasets and local neighborhood information.

Varying the number of dimensions

Assuming no replication (i.e., $k = 0$), we construct *BD* and *Dwarf* cubes with variable number of dimensions d (5 up to 25), with cardinalities equal to 1k values. The datasets consist of 10k tuples, following the uniform, self-similar (80-20) and Zipfian ($\theta = 0.95$) distributions. For the *BD* evaluation, the experiments are conducted in our distributed testbed, consisting of 16 LAN nodes. Storage consumption and insertion times are presented in Table 4.2.

Our system exhibits impressively faster creation compared to the centralized method, due to the fact that *BD* allows for overlapping of the store process (each peer stores its part of the cube independently). The acceleration is more apparent as the number of dimensions and the skew grow, since such datasets result in larger cubes. For instance, *BD* inserts the 25-d skewed cubes up to 3.5 times faster than *Dwarf*. The acceleration factor of course is not directly proportional to the number of participating nodes: The cube calculation remains serial and the network communication introduces latencies. Since the nodes of our testbed are part of a LAN, the network latencies are rather small. In a WAN environment, we would expect results to be somewhat worse. However, we believe that a LAN setting provides a more realistic representation of the network conditions in modern distributed systems (such as the Cloud), where nodes are connected through high speed internet (e.g., in a datacenter facility).

Note that the total cube size is always bigger than the fact table by a factor that increases with dimensionality and skew (152 times for the central and 195 times for *BD* in the worst case). This observation confirms previous findings documenting that *Dwarf* blows up the size of some datasets, especially for sparse cubes [DBS08]. This index growth, which constitutes an intrinsic characteristic of the method, is an extra motivation for the distribution of the *Dwarf* cube.

In addition to that, *BD* induces a small storage overhead. This overhead is mainly attributed to the mapping between the UIDs (set to 4 bytes each in our implementation) that every dwarf node needs to keep in order to be accessible by network peers and dwarf node IDs, as well as the parent list, which is necessary in the `mirror` process. This also explains why the overhead slightly increases with the number of dimensions. Nevertheless, this overhead is shared among the participating nodes. Indicatively, in the case of the 25-d Zipfian dataset, even though the overhead is 43MB, the burden of each of the 16 peers is less than 3MB.Thus, the big advantage of *BD* is the fact that it can store almost N times as much data as *Dwarf* (for $k = 0$), using N computers with capabilities similar to the one used in the central case.

Figure 4.6 plots the distribution of storage and messages among the nodes of the system, where the measured quantities for each node are sorted in ascending order. All nodes host similar quantities of storage space, which is on average equal to $\frac{\text{size(BD)}}{N}$. The cost per fact table tuple insertion is small and increases with d, as more dwarf nodes are created, increasing the number of network nodes to be contacted. This load is also equally distributed among peers. Another observation is that these characteristics are maintained even as the number of dimensions increases: The structure may increase exponentially in the number of components, yet *BD* performs well in distributing it over the network.

To further examine both storage and load distribution among network nodes, we employ the simulation environment with $N = 128$ nodes and distribute over them datasets of 100k tuples, with 10% density and d ranging from 2 to 8 (due to memory constraints). Figure 4.5 plots the storage consumption and the message distribution. Results show that there are very few nodes

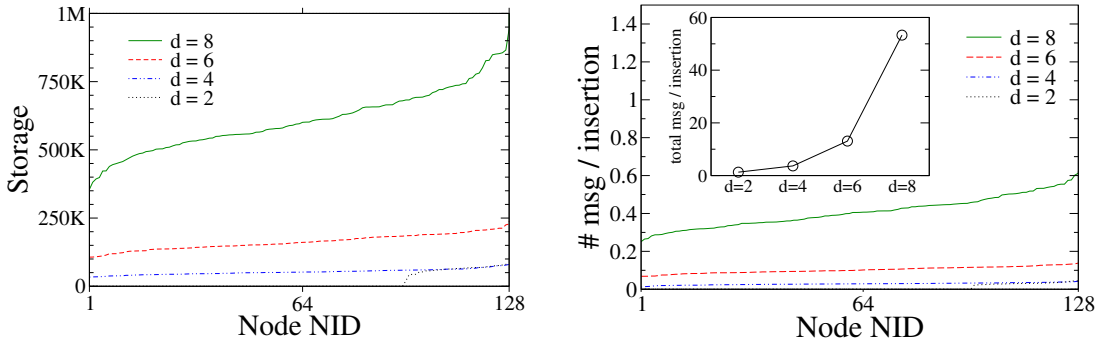


Figure 4.5: *Storage and message distribution over the simulated nodes for various datasets*

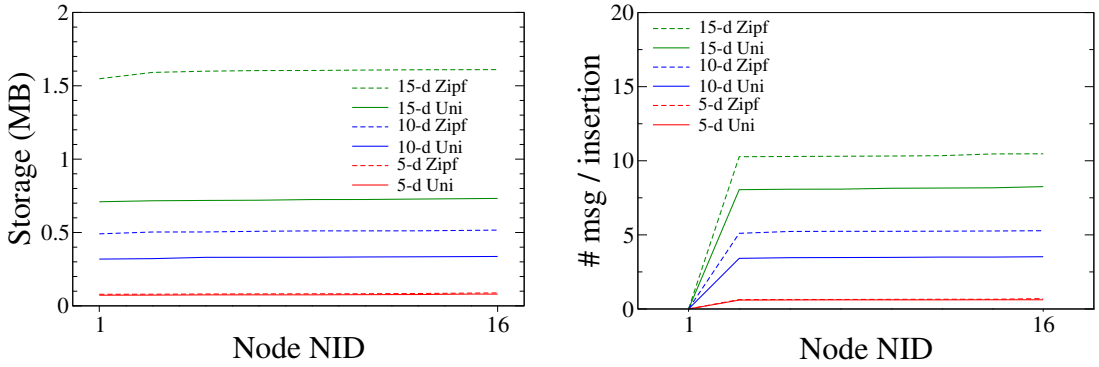


Figure 4.6: *Storage and message distribution over the network nodes for various datasets*

that both receive many requests and store significantly more data than the majority, unlike the case of 8 peers that we examined above. This is due to the combination of comparatively small datasets and the limited number of node acquaintances (NS=16) compared to the total network size (every node knows about 12% of the whole network, whereas in the previous experiment, this percentage was 100%). As our simulations proved, increasing the size of NS favors data balancing, as a larger choice of nodes enables a more even distribution of storage and load among the participants.

Table 4.3 presents the cube insertion times for the uniform datasets when the replication parameter k ranges from 0 to 4. Although the cube is inserted $k + 1$ times and the storage consumption as well as the communication cost sustain a k-fold increase, the increase in the total insertion time is not proportional to k. Once again, parallel disk I/O operations alleviate the impact of the linear increase in the size of the data to be stored, resulting in an average 2–2.5 factor increase in insertion times in the worst case (comparing $k = 0$ with $k = 4$).

Varying the number of participating infrastructure nodes

To examine the gains in insertion time caused by the parallelization of the storage process and to

Table 4.3: *Creation time (in sec) for* Brown Dwarf *data cubes of various dimensionalities varying the k parameter*

d \ k	0	1	2	4
5	4	5	6	8
10	10	13	19	23
15	29	36	49	76
20	55	75	104	152
25	88	110	150	220

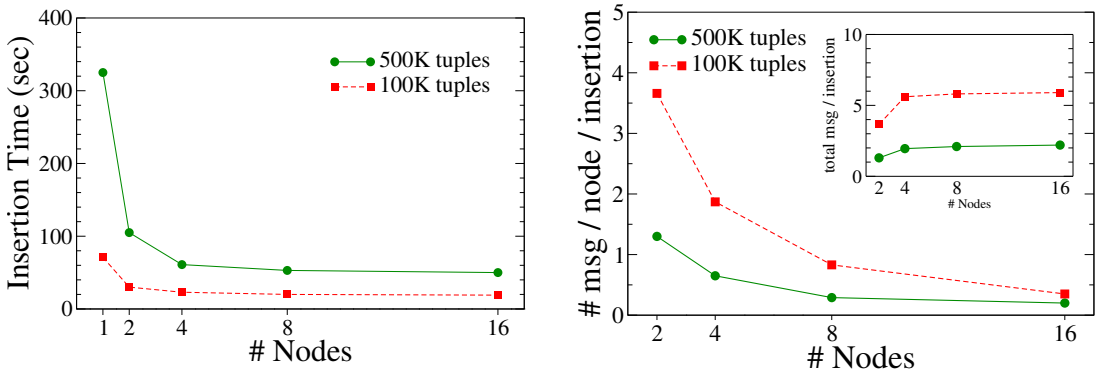


Figure 4.7: *Time and incoming messages per network node for various network and data cube sizes*

analyze the evolution of the communication cost we vary the number of participant commodity nodes. As input data, we use two 5-d, uniformly distributed cubes, consisting of 100k and 500k tuples respectively. Cardinalities for all dimensions are set to 100.

The graphs in Figure 4.7 reveal on one hand that the increase in the number of participating nodes enhances the system performance. However, the speedup is not linear and there is a point beyond which no dramatic improvement is demonstrated. This is due to the fact that the serial nature of the cube creation algorithm poses a limit in the parallelization of the storage process itself, since dwarf nodes can not be stored faster than their calculation. Moreover, as the number of nodes increases, so does the cost of the node orchestration, which in turn hinders the acceleration. On the other hand, a larger number of nodes induces bigger total communication costs, as shown in the included graph. The total number of messages per insertion shows similar behavior to that of the insertion time: The communication cost is not severely affected after a certain point. It is also worth noticing that, even though the communication cost increases, the overhead per network node decreases as messages are scattered among more peers.

Varying the number of tuples

We now examine how the *BD* cube behaves when scaling the number of tuples in the fact table. Keeping the number of dimensions constant ($d = 5$) and all cardinalities equal to 100, we create

Table 4.4: *Effect of various dataset sizes on 5-d cubes*

# Tuples	Fact Tbl size	size (MB)		time (sec)	
	(MB)	*Dwarf*	*BD*	*Dwarf*	*BD*
10k	0.1	1.2	1.5	4	4
100k	1.7	4.8	5.9	71	20
500k	7.0	24.7	30.4	325	53

Table 4.5: *Effect of various densities on 5-d cubes*

Density	Fact Tbl size	size (MB)		time (sec)	
(%)	(MB)	*Dwarf*	*BD*	*Dwarf*	*BD*
0.01	1.4	5.4	6.6	54	13
0.1	1.4	4.6	5.5	39	9
1.0	1.4	3.0	3.5	19	5
10.0	1.4	1.4	1.6	5	3

datasets of 10k, 100k and 500k tuples. Table 4.4 summarizes the results. As the number of tuples increases, this leads to higher acceleration factors for our insertion method: While the insertion of the 10k dataset lasts the same in both systems, the 500k dataset is inserted almost 6 times faster. We observe that our system maintains roughly the same storage overhead for all dataset sizes, since the increase in the number of tuples results in an increase in storage for both *Dwarf* and *BD* cubes.

Varying the density of the cube

In this experiment we examine how the density of the data cube affects its insertion in the *BD* system. We create 5-*d* datasets of 100k tuples following a uniform distribution, with variable densities ranging from 0.01% to 10%. We document the amount of storage allotted by each network node for *Dwarf* and *BD* and present it in Table 4.5. Our system remains faster in creating and distributing the structure by a constant factor of about four. The next observation is that the lower the density, the larger the total amount of storage for the *BD*, since sparse cubes leave little room for redundancies, thus resulting in larger dwarf structures. Therefore the denser the cubes, the less noticeable the difference in storage between *Dwarf* and *BD*.

4.5.2 Updates

In this section, we observe the behavior of *BD* when update batches are inserted to the distributed structure using our real testbed of 16 nodes. Utilizing the same 10k-tuple datasets of varying dimensions (5–25) described before, we present measurements for two different settings. In the first setting, we apply 1% incremental updates which follow the uniform and the self-similar (80-20) distribution. In the second setting, we apply increments of sizes from 0.1% up to 10% to the

Table 4.6: *Effect of 1% increments over various dimensions*

d	Uniform time (sec)		Uniform msg/upd	80-20 time (sec)		80-20 msg/upd
	Dwarf	*BD*	*BD*	*Dwarf*	*BD*	*BD*
5	7.1	7.2	14.6	7.5	6.4	13.7
10	17.7	14.3	50.8	21.3	14.4	49.8
15	30.8	21.8	111.0	43.4	31.2	120.4
20	48.6	27.9	193.3	104.1	65.8	200.2
25	89.1	39.1	300.7	172.1	103.6	305.7

Table 4.7: *Effect of various update types and sizes on the 10-d dataset*

update size %	update time (sec)		msg/update	
	old	*fresh*	*old*	*fresh*
0.1	1.9	1.8	49.5	40.2
1.0	14.3	11.6	50.8	41.5
10.0	127.2	78.0	61.2	56.6

10-d cube. These increments are of two types: Consisting of tuples generated either by combining existing dimension values (*old*), or by adding new values to the dimensions' domains (*fresh*). For both types, we record the total update time as well as the number of messages required per individual update. We present the results for both settings in Tables 4.6 and 4.7 respectively.

Taking advantage of the inherent parallelization that updates (similar to insertions) exhibit, *BD* is up to 2.3 times faster for the high-dimensional sets. Dimensionality plays a dicisive role in both the time and the cost of updates. This observation is clearly documented: The more the dimensions, the larger the *BD* cube created, thus the more dwarf nodes and cells are affected (see Table 4.6). As observed in the case of cube creation, skewed datasets take longer to update, due to the fact that updates in a dense part of the cube affect more dwarf nodes and cells, thus slowing down the process and creating larger network traffic.

Table 4.7 shows that the size of the update batch over the original cube has negligible effect on the communication cost per update. However, it is interesting to note that while *fresh* updates over the original cube create more nodes and cells in the structure, yet the cost is inversely proportional: The number of update messages per update are almost 20% less compared to *old* updates. This is due to the fact that the *fresh* batch contains new attribute values and fails to find redundancies with the originally inserted *BD* structure. As a result, more new nodes are created, yet less recursive updates of the affected *ALL* values are performed, hence less messages are transmitted.

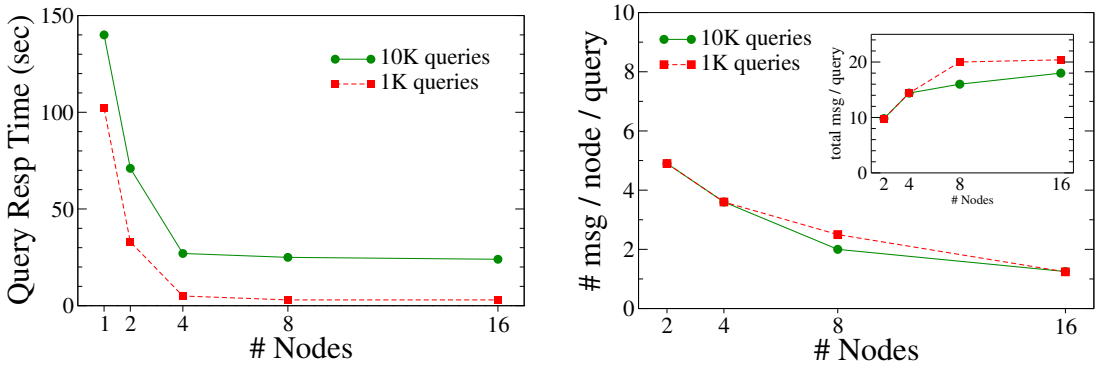


Figure 4.8: *Resolution time and messages per network node for a 20-d dataset over various network sizes and query workloads*

4.5.3 Query Processing

In this section we investigate the query performance of *BD* compared to that of *Dwarf* and examine the load distribution among peers with and without adaptive mirroring.

Varying the number of dimensions

Using the same datasets as in the insertion and update experiments, we pose two 1k querysets that follow the uniform and Zipfian ($\theta = 0.95$) distributions respectively, with the ratio of point queries set to 0.5. Moreover, P_d, which we define as the probability of a dimension not participating in a query, is set to 0.3. Table 4.8 summarizes the results. It should be noted that the dwarf index does not remain in memory for either method, thus I/O is performed for every query.

First, we observe that in all cases *BD* resolves the workload noticeably faster than the centralized version. While the query response times rise with the dimensionality for *Dwarf*, *BD* times remain almost constant and only the 25-d workloads cause a slight slowdown. The resolution of each dimension of the query is an atomic operation that may be performed by separate peers. Thus, having 16 nodes perform I/O operations in parallel instead of just one significantly boosts performance. Especially in the case of biased and high dimensional workloads, where there is more room for parallelization, *BD* exhibits impressive acceleration factors, performing up to 60 times faster than *Dwarf*. It is therefore apparent, that *BD* is able to handle a significantly larger (by orders of magnitude) request rate than its centralized version.

Moreover, the number of messages per query is in all cases bound by $d+1$: The system needs d messages to forward the query to the dwarf nodes along the path towards the answer and one to send the response back to the initiator.

Varying the number of nodes

For the 20-d dataset of the previous experiment, we plot the response times and the per query communication when scaling the number of network nodes from 1 (centralized case) to 16. Apart

Table 4.8: *Query resolution times and communication cost over various 1k querysets*

d	Uniform time (sec)		Uniform msg/query	Zipf time (sec)		Zipf msg/query
	Dwarf	*BD*	*BD*	*Dwarf*	*BD*	*BD*
5	5.2	4.0	5.8	1.9	1.7	5.5
10	30.1	2.6	10.9	29	1.2	10.6
15	65.2	2.9	15.6	55.4	1.2	15.5
20	102.1	3.0	20.8	88.3	1.5	20.3
25	182.5	13.2	25.9	172.1	9.2	25.6

Table 4.9: *Measurements for various APB datasets*

Density	#Tuples	Fact Tbl (MB)	size (MB) *Dwarf*	size (MB) *BD*	insertion time (sec) *Dwarf*	insertion time (sec) *BD*	querying time (sec) *Dwarf*	querying time (sec) *BD*
0.1	1.2M	24	17	20	42	16	40	12
0.2	2.5M	49	35	41	82	32	55	12
0.3	3.7M	73	51	60	126	53	80	12
0.5	6.2M	131	74	98	314	93	93	13

from the uniform workload of 1k queries that was used in the previous experiment, we also pose a workload of 10k queries with the same characteristics, to further stress the system. Figure 4.8 pictorially presents the results.

The first graph plots the total response times for the query batches. As observed, for a small network size, the increase in nodes dramatically accelerates responses: Expanding the network from 2 to 4 peers results in 4 times faster response times. The performance gain though becomes smaller, as the size grows: The difference is almost negligible when going from 8 to 16 nodes. This is due to the fact that, depending on the dataset and its dimensions, the parallelization ability of *BD* is saturated after a certain number of nodes.

The second graph presents the average number of messages needed to answer a query for both workloads. We notice that as the number of peers increases, the communication cost increases too, but not uncontrollably, since it converges to $d+1$ messages per query (inner graph). Furthermore, the cost is scattered among peers, resulting in less load per network node (outer graph).

Benchmarks and real data sets

Next, we examine the behavior of *BD* with more realistic input sets. We utilize 6 different datasets. Using the APB-1 benchmark [apb], which simulates a realistic OLAP business situation and is used in a plethora of papers to evaluate data warehousing solutions. With its generator we produce four 4-d datasets with densities varying from 0.1 to 0.4. The dimension cardinalities are 24, 9000, 900 and 9, while there also exists one measure attribute. The other 2 datasets

Table 4.10: *Measurements for the real datasets*

	size (MB)		insertion time (sec)		A (sec)		B (sec)		C (sec)	
Dataset	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*	*Dwarf*	*BD*
weather	9.3	11.4	120	23	234	11	165	12	114	12
Forest	8.0	9.8	66	20	144	11	111	11	70	12

are subsets of the *weather* [wea] and *forest* [Bla] data sets (10k tuples each). The *forest* dataset has 10 dimensions with cardinalities as reported in [DBS08], while the *weather* dataset has 9 dimensions, corresponding to truncated ocean weather measurements for September 1984.

As far as the querysets are concerned, we use the APB query generator to produce 1k query workloads, both point and aggregate ones. For the real datasets, in order to produce our workloads, we first order the tuples and then use the Zipfian distribution to select those that will form 10k query sets. We vary θ from 0 (uniform distribution) to 2, producing three workloads (denoted as A, B and C respectively). The ratio of point queries is set to 0.5, while for the aggregate ones we set $P_d = 0.3$.

The results, presented in Tables 4.9 and 4.10, are in line with the findings of the previous experiments. First, we notice that the dwarf algorithm can, depending on the input data, perform efficient compression of the cube. The storage overhead is at most 14MB (for the APB dataset of 6.2M tuples), a steady 17% increase compared to the centralized case. Nevertheless, the cube is now shared among each of the 16 peers participating in the system. For the construction times, *BD* is obviously faster than *Dwarf*. Our results show that the distributed version is over 5 times faster compared to the centralized run (for the *weather* dataset), giving impressive cube creation times (about 1.5 minutes for 6.2M tuples). Query response times are up to 20 times faster for *BD*, which is able to handle almost 1k queries per second.

4.5.4 Mirroring

This set of experiments aims to evaluate the mirroring process, both static and adaptive, in terms of storage overhead, load balancing and scalability.

Static Mirroring

To examine the load distribution among the nodes of *BD* when varying the k parameter, we pose different querysets in a testbed of 16 network nodes. Using a 10-d cube with 10k tuples (uniformly distributed) we pose 5k querysets, following the uniform and the Zipfian (with various θ values) distributions respectively, with the ratio of point queries set to 0.5 and $P_d = 0.3$. Queries arrive at an average rate of $\lambda = 100\frac{queries}{sec}$. For this set of experiments, only static mirroring is allowed. For the replication factor we chose the values $k = 0$, meaning no replication

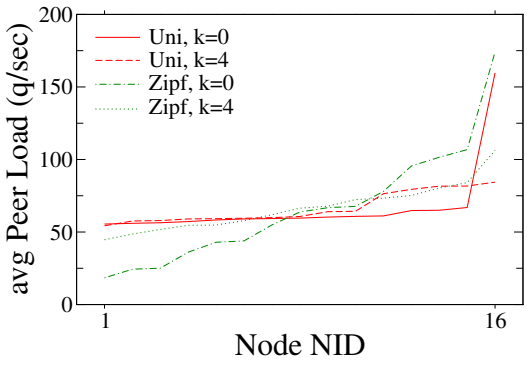


Figure 4.9: *Load distribution among peers for various replication factor values (static replication)*

and $k = 4$, meaning that each dwarf node is stored in one third of the network nodes. From the biased workloads we present results for $\theta = 1.5$, as results with different θ values demonstrate little qualitative difference.

Figure 4.9 shows the average load per network node throughout the experiment. As we would expect, the more skewed the workload, the more uneven the load distribution gets. Skewed workloads may produce fewer loaded *BD* components, yet their load is larger compared to random querysets. When increasing the k parameter from 0 to 4, we observe that the load distribution becomes fairer, especially in the uniform case, where the initial load of the most loaded dwarf nodes gets allotted over their mirrors. Nevertheless, for more skewed workloads there still remain a few nodes with noticeably heavier load due to the fact some nodes have such a high load in the first place, that it still remains high after the replication. It should also be noted that when increasing k, there is a trade-off between load distribution on one hand and creation time and storage overhead on the other. Choosing $k = 4$ means that the cube is inserted 5 times, increasing the cube insertion time by a factor of 3.9.

From these experiments we deduce that a static replication strategy leads to an uneven load distribution with a small part of the system being significantly loaded regardless of the incoming requests.

Adaptive mirroring

We now enable the adaptive mirroring mode of *BD*, testing it with the same setup as the static case. Figure 4.10 displays the number of the created replicas in total as well as the induced system-wide communication overhead over time for different queryset distributions and for $Limit^{s}_{exp} = 10\frac{queries}{sec}$.

In all cases, our scheme increases the number of replicas at overloaded parts of the structure through its *expansion* mechanism in order to bring the system to a balance and eliminate the instances of overloaded dwarf nodes. The more skewed the queryset, the more replicas the *BD* system produces. This is natural since, as shown earlier in the experimental section, overloaded

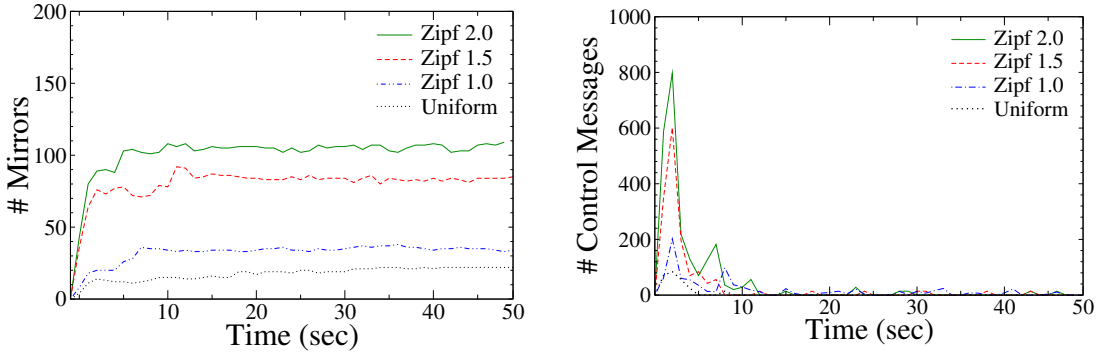


Figure 4.10: *Number of replicas over time and control message overhead for different query distributions (adaptive mirroring, with $Limit^{s}_{exp} = 10$)*

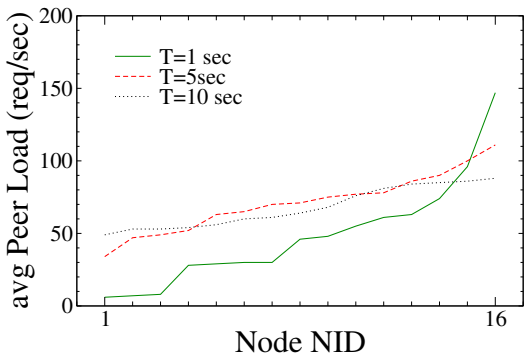


Figure 4.11: *Load distribution before, during and after mirroring ($\theta = 1.5$, $Limit^{s}_{exp} = 10$)*

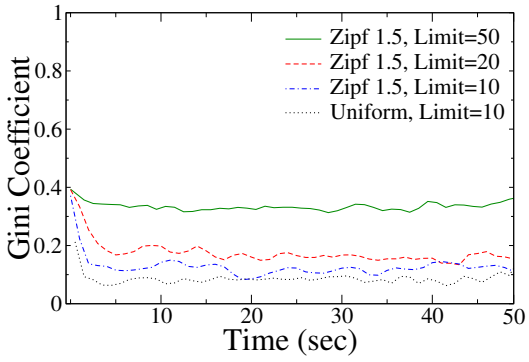


Figure 4.12: *Gini coefficient over time for various workloads and $Limit^{s}_{exp}$ values*

dwarf nodes for skewed distributions have substantially higher load. The rate at which replicas are created decreases with time and reaches a steady state, where the number of mirrors remains almost constant. It is worth noticing that *BD* reaches the steady state fairly quickly (within a few seconds – less than 10 in our experiments), due to the ability of the *expansion* mechanism to create multiple mirrors according to the amount of overload.

However, until steady state is reached, we observe a short period of fluctuation in the number of replicas, which is more apparent for workloads of high skew. The simultaneous initiation of the mirror process leads to temporary inconsistencies with regard to which mirror each node is aware of. In this case, it takes some time until all mirrors discover each other, creating an uneven load distribution among them. It is also very important to stress that the mirroring process diminishes substantial load inequalities with minimal storage overhead: About 100 replicas are created at most in an initial *BD* structure of 130k dwarf nodes, which translates to less than 0.1% of extra storage consumption.

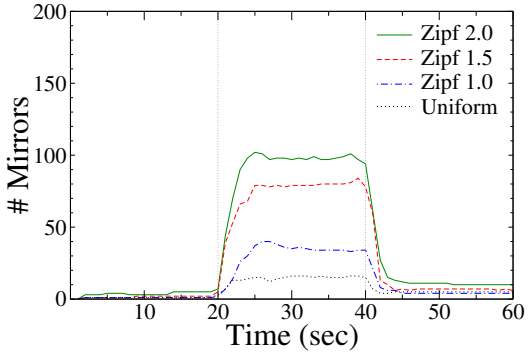


Figure 4.13: *Number of replicas over time for a pulse-like query rate with* $Limit^s_{exp} = 10$

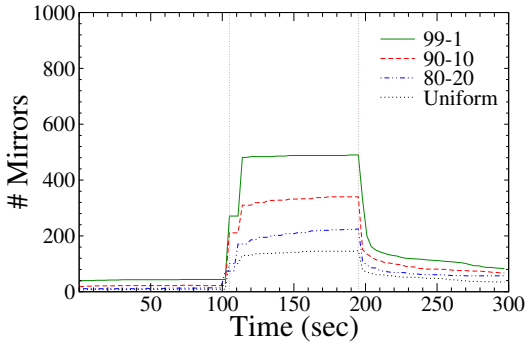


Figure 4.14: *Replicas over time for a pulse-like query rate in our simulation environment*

The second graph of Figure 4.10 depicts the number of control messages in the course of time. These are the messages required to inform parent, children and mirror nodes of the creation of a new replica or the deletion of an existing one, as well as the insertion message itself in case an *expansion* occurs. As expected, *expansion* and *shrink* comes with a certain communication cost: The more the replicas created or deleted, the more the control messages required. However, this message burst lasts only for a short period of time, until the system reaches a steady state. Moreover, the communication cost is shared among the participating resources.

Figure 4.11 also shows that *BD* moves towards a more balanced load distribution with each step. Load snapshots of a skewed workload ($\theta = 1.5$) at the beginning, the middle and a random point in the steady state show that our method manages to decrease disparity between node loads. That is its main advantage compared to static mirroring: The system gradually moves towards states where more server instances are involved in query processing and less overloaded dwarf nodes exist. Without adaptive mirroring, the system exhibits overloaded peers with much higher loads, while the large majority of the rest receive very few requests. Thus, the degraded performance of overloaded nodes degrades the whole system's performance.

Experiments with different values of the $Limit^s_{exp}$ parameter show that the higher its value, the smaller the total number of mirrors created and the smoother the transition to the steady state. However, higher values of $Limit^s_{exp}$ fail to guarantee fairness of load distribution. This becomes clear in Figure 4.12, which plots the value of the *Gini coefficient* G over time for various $Limit^s_{exp}$ values. G is a summary statistic that serves as a measure of inequality in a population. It is calculated as the sum of the differences between every possible pair of individuals, divided by the mean size. Its value ranges between 0 and 1, where 0 corresponds to perfect equality. Assuming our population comprises of the number of received requests by each mirror, 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. In all cases, its value stabilizes extremely quickly, a proof that *BD* replica-sets are pretty stable over time. The runs with the low threshold offer better load balancing (with G dropping from 0.4 to 0.1 for the skewed workload) as more mirrors are created, giving the system a chance to balance the inequalities.

To further examine the behavior of our system under stress conditions and sudden changes in load, we conduct another series of experiments. Using the same workloads and an initial query rate of 10 $\frac{queries}{sec}$, we suddenly increase the query rate by a factor of ten (λ reaches 100 $\frac{queries}{sec}$) after 20 sec of querying time. After another 20 sec, the rate decreases again to its initial value. We evaluate the efficiency of the *expansion* as well as the *shrink* mechanisms to perceive the change and adapt the number of replicas accordingly, in order to perform as required with minimum storage consumption.

Figure 4.13 presents the number of existing replicas over time throughout the experiment for $Limit^s_{exp} = 10$. Almost immediately after the increase in λ, the number of replicas increases rapidly, almost 10 times as much. After the end of the pulse, the *shrink* mechanism erases underloaded dwarf nodes, freeing up disk space, making it (possibly) available for the creation of other mirrors. Again, we observe how quickly *BD* manages to detect the change in load. Within a few seconds the mirrors decrease dramatically and keep decreasing gradually, tending to reach the state that existed before the pulse was applied. However, the more biased the workload, the more the steady state before and after the pulse differs.

To affirm our findings, we repeat the experiment in our simulation testbed of 128 nodes. This time the query rate changes from 100 to 1000 and back to 100 $\frac{queries}{sec}$, since the network size is an order of magnitude larger than before. The results, depicted in Figure 4.14 are qualitatively similar to those of the real deployment, proving the ability of *BD* to fully exploit the elasticity that the distributed environment offers.

4.5.5 Effect of Dimension Grouping

Here we evaluate the impact of dimension grouping on performance as well as fairness in both storage and load. Using the 20-d dataset and varying the h parameter from 1 to 10, we apply a batch of 20k updates in an existing *BD* structure of 100k tuples. Afterwards, we pose 1k uniformly distributed queries. We measure the time and communication cost of the operations, as well as the value of *G* for both the storage consumption and the produced load (Table 4.11). Assuming our population comprises of the size of the stored data and the number of received requests by each node respectively, we calculate *G* as an index of storage and load distribution among servers.

As h increases the communication cost of all operations noticeably decreases, since paths of h nodes reside in the same host, consequently reducing the operation times. Naturally, as the grouping becomes more coarse grained it causes imbalance in both storage and load. Although

Table 4.11: *Measurements for various values of h using the 20-d dataset*

	Storage	Update			Querying		
h	G	time/update (ms)	msg/update	G	time/query (ms)	msg/query	G
1	0.01	153	35.6	0.05	109	17.2	0.05
2	0.20	148	30.3	0.13	38	8.6	0.08
5	0.39	102	23.4	0.29	15	3.9	0.18
10	0.61	91	20.8	0.30	13	2.3	0.32

Table 4.12: *Implications on data and query processing for increasing number of failures and different T_{fail} values*

$\|N_{fail}\|$	$T_{fail}(sec)$	query loss (%)	total redirections	msg/query	time per query (ms)
0	-	0	0	9.8	57
1	90	0	11	9.8	79
2	90	0	204	10.4	257
4	90	2.9	841	11.1	734
1	60	0	21	9.9	107
2	60	0	258	10.5	304
4	60	4.3	894	11.2	812

the effect on storage is much more severe, the load remains more balanced among servers, even in the exaggerated case of $h = 10$. This experiments proves the trade-off between performance gain and balance, suggesting a choice of h much smaller than d.

4.5.6 Node Failures

Our system relies on the cooperation of commodity nodes forming an unstructured P2P overlay. In such an environment it is likely that failures will occur throughout the execution of a workload. Indeed, Google reports an average of 1.2 failures per analysis job in their MapReduce infrastructure [DG08]. It is thus important to examine the impact that failing nodes have on our system.

Using the 10-d dataset (with $k = 3$) and a uniformly distributed query set of 5k queries that arrive at a rate of 10 $\frac{queries}{sec}$, we enforce node failures as follows: every T_{fail} sec, a subset N_{fail} of the online peers fails in a circular way, while previously offline nodes are reinserted to the network. Note that, by failing we mean that nodes depart ungracefully, without informing any other peer. Starting from $|N_{fail}| = 1$ we gradually increase it up to the value of 4 (since each dwarf node exists in $k + 1 = 4$ different network nodes), aiming to test *BD*'s fault tolerance and examine its performance under volatile conditions. The adaptive mirroring mode is turned off, in order to better interpret the results.

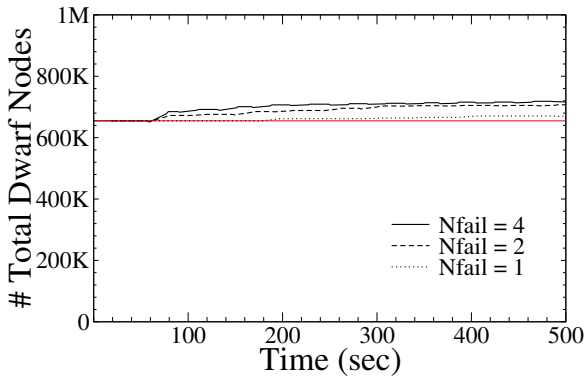


Figure 4.15: *Number of replicas over time with various numbers of failing nodes*

It is worth noting that the parameters used in these experiments are far more pessimistic than the reported ones. According to [DG08], Google experiences failures in 0.76% of the average number of machines allocated for an analysis job, with T_{fail} being almost 500 sec on average. Table 4.12 summarizes our findings. Note that the query time column represents the absolute time for a single query to be resolved (not the average completion time of many queries run in batch mode).

We observe in Table 4.12 that the system maintains the theoretical guarantee that for any failure level below the replication no data or query loss will occur (see lines with $|N_{fail}| < 4$). Even when 25% of the nodes fail, a very small portion of the queries has to be restarted (less than 5% in the worst case). This happens because, for a query to fail, *all* the replicas of at least one the respective dwarf nodes that reside on the answering path must be offline. Because of the automatic replenishment of the replica-set whenever a dwarf node falls below $k+1$ copies, query loss is very small. The number of messages needed per query now tops d due to the redirections needed for some requests. Redirections and mostly the induced timeouts increase the average response time by a factor of roughly 13 compared to the no-failures run. Still, this number does not incorporate the query resolution speed-up that *BD* exhibits when many queries are sent in batches.

In Figure 4.15, we plot the total number of replicas in the system over time, for the various $|N_{fail}|$ values. We observe that the number of mirrors remains stable, despite of the random node departures and very close to the initial value for $k = 3$ (represented by the horizontal red line). In fact, the number of replicas is somewhat larger than the theoretical value. This happens because more than one peers might initiate a mirroring process at the same time, thus producing more replicas. This becomes more obvious as the ratio of failing nodes increases. The small fluctuations are due to the deletion of mirrors as nodes fail.

4.6 Summary

In this chapter we presented *Brown Dwarf*, a system that distributes a data cube across peers in an unstructured P2P overlay. To our knowledge, this is a unique approach that enables users to pose group-by queries and update multidimensional bulk datasets online, without the use of any proprietary tool. *BD* creates a distributed *Dwarf* with a single pass over the the input data, allotting dwarf nodes across peers and interconnecting them.

Our system employs many plausible features required by an application and its respective hardware: It is scalable, as it can use an unbounded number of cooperating nodes, distributing computation and storage; it provides data availability through its adaptive replication scheme according to both workload and node failures; it efficiently answers all point and aggregate queries in a bounded number of steps; finally, it is cost-effective, as it uses only commodity hardware and with its expand/shrink scheme each dataset takes up only the necessary amount of storage.

Our evaluation shows that the data cube is evenly distributed across a number of cooperating peers. Both creation and querying times are significantly reduced (often by an order of magnitude) due to the parallel paths taken in the overlay. Also, *BD* expands popular parts of the structure using local only load measurements, while constantly monitoring the whole set to remain above an acceptable replication threshold. Finally, it minimizes node overloads and query processing times, even in very demanding and dynamic workload/churn conditions.

CHAPTER 5

HORAE: An Analytics Platform for Temporal Data

In this chapter we deal with the special case of *time series* data, meaning data characterized by a temporal aspect. Taking into account the properties of time series data as well as the requirements of its analysis, we propose *HORAE*, a shared-nothing analytics platform that stores, queries and updates time series data in a fully distributed manner.

HORAE employs a hybrid solution for data storage and processing: High-rate updates and queries targeting the most recent items are handled by a *HiPPIS*-like, DHT-based component, that enables fast insertion times and multidimensional indexing. The large bulk of the data is handled through an enhanced *Brown Dwarf* datacube structure, that adaptively materializes and replicates according to demand. The two components seamlessly integrate to offer the advantages of powerful aggregate data processing along with scalability and elasticity of commodity resources.

Our prototype implementation over an actual testbed proves that HORAE is able to efficiently handle large rates of both updates and queries, tolerate high failure ratios and expand or contract its resources according to demand. A direct comparison with a state-of-the-art warehousing solution demonstrates HORAE's advantages in both performance and elasticity under variable workloads: HORAE accelerates query resolution by orders of magnitude, manages to quickly adapt even after sudden bursts in load and remains unaffected with a considerable fraction of frequent node failures.

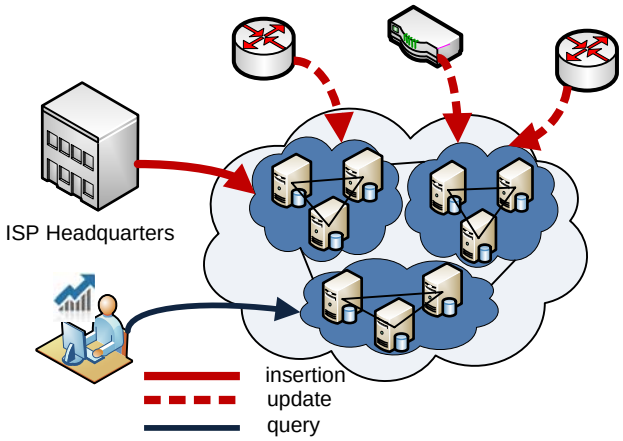


Figure 5.1: *Scenario of a data warehouse-like system for managing temporal data of an ISP*

5.1 Overview

Most of the data used in data warehousing comes in the form of security, network and system event logs. For example, network usage data collected from logs across network and security devices of a communications service provider can be correlated in real time to identify security threats as they happen, catch fraudulent activities, make better use of resources and improve the quality of service.

In such applications, data are usually determined by a temporal aspect: *Time series data* are characterized by a time attribute (e.g., time-stamps of router data or dates of purchases) presented at different levels of granularity through the use of *concept hierarchies* (e.g., `Day`<`Month`<`Quarter`<`Year`). Thousands or millions of such records are produced per second and modern systems are expected to be able to both incorporate and process them.

As a motivating scenario, let us consider a network service provider that maintains records of its router operations. Instead of creating a centralized data warehouse on-site with a large upfront and maintenance cost, the management chooses to transfer data and computation off-site to a location-transparent facility and access it more easily and ubiquitously. In this manner, the establishment significantly lowers maintenance and hardware costs while enjoying a scalable, real-time decision support system that is viable even under heavy update load (Figure 5.1).

Existing systems inadvertently fail in one of the two basic requirements, powerful data processing and high-rate updates. Even the new class of Cloud-based analytics engines that have been proposed for large-scale data management, although they offer scalability, robustness and availability at low cost, they pose some limitations. Since they are based on the MapReduce programming model, they mostly target analytics jobs submitted in batches rather than real-time and interactive "per-tuple" processing. This is exactly what we pursue.

In previous chapters we presented two systems that emerged by our efforts to provide an always-on, real-time data access system for concurrent update and query processing with fast response times. Both works, *HiPPIS* and *Brown Dwarf*, aim to satisfy the same general need: The creation of a data-warehouse-like platform, employed on commodity machines, which will be able to provide an always-on, real-time data access and support for online processing. Techniques from the field of P2P computing have been exploited in order to ensure scalability, fault tolerance and fairness in resource utilization. However, each work approaches the same issue from a different perspective, setting different priorities.

HiPPIS focuses on the management of hierarchical data, allowing queries of various granularities through roll-up and drill-down operations. This fact makes *HiPPIS* well suited for scenarios where a more detailed representation of the data is needed. The simplicity of the *HiPPIS* data structure allows for fast insertion of the initial fact table, without any preprocessing. However, since no a-priori materialization of the cube is performed, group-by queries require further processing after the collection of all tuples that correspond to them. Updates are as fast as insertions, incurring a small overhead which depends on the level of consistency needed by each application. Therefore, in situations where data are constantly updated at a high rate, *HiPPIS* can cope in a cost-efficient way.

Brown Dwarf manages to distribute a well known and established data structure, which materializes a data cube, achieving, in some cases, significant compression rates. At the cost of preprocessing, which is paid only once though, aggregate queries can be answered as easy and naturally as point ones. However, updates in *Brown Dwarf* are quite costly, since a single new tuple insertion triggers multiple changes in aggregate values across the structure. Therefore, *Brown Dwarf* is more efficient in environments where the update rate is not very high, compared to the query rate, or where updates can be applied in batches.

In this chapter we design a complete system that combines the two approaches in order to benefit from the strengths of both of them. To that end we present the *HORAE*[*] platform, where techniques from both *HiPPIS* and *Brown Dwarf* are applied in order to disseminate, query and update high volumes of multidimensional time series data: Recent data, which are bound to be updated rapidly and queried in finer granularity will be stored in a *HiPPIS*-like system, whereas historical data will be stored in less detail in *Brown Dwarf* cubes.

Our prototype implementation tries to maintain the best of both worlds: A powerful indexing/analytics engine for immense volumes of data both over historical and real-time incoming updates and a shared-nothing architecture that ensures scalability and availability at low cost. Geographically spanned users, without the use of any proprietary tool, can share information

[*]Horae were the goddesses of seasons and the natural portions of time in Greek mythology.

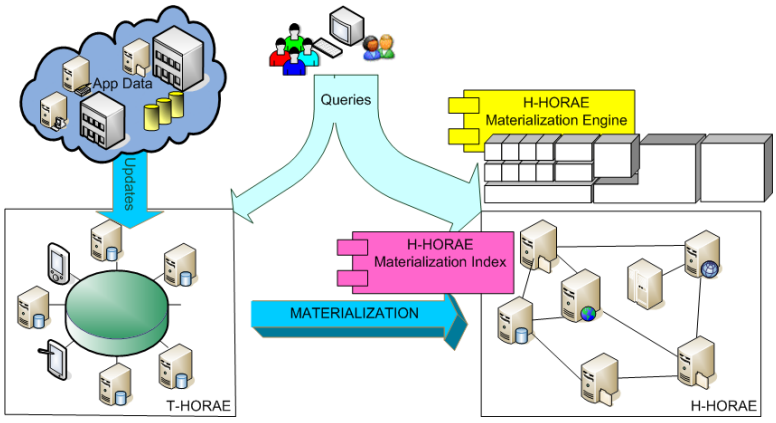


Figure 5.2: *An overview of the HORAE system for managing the temporal data of an enterprise*

that arrives from distributed locations at a high rate in the form of time series and query it in different levels of granularity. An overview of the HORAE system is depicted in Figure 5.2, where a use case scenario of an enterprise managing its temporal data is presented.

HORAE utilizes a mixture of two mechanisms: First, it employs a *HiPPIS*-like, DHT-based storage and indexing substrate in order to buffer incoming updates at high rate and answer queries relative to the most recent data. This "transactional" part, called *T-HORAE*, maintains the required scalability and distribution constraints while, due to its simple insertion mechanism, it efficiently handles frequent data insertions. Its adaptive reindexing mechanism ensures that queries of any granularity are still effectively answered.

H-HORAE stores, replicates and maintains the large amount of data as they are gradually transfered from T-HORAE by distributing a highly efficient cube indexing structure over an unstructured P2P overlay, similarly to the *Brown Dwarf*. Its advantages are many-fold: It provides an adaptive materialization scheme that summarizes data according to the level of temporal detail they are requested, in order to minimize storage consumption and maximize query throughput. It organizes the cube indexing so that updating costs and times are minimized. Finally, it employs both static and adaptive replication over commodity hardware to ensure availability and elastic behavior in a transparent manner.

The contributions of HORAE are tha folowing:

- A complete indexing, query processing and update system for multidimensional time series data over a distributed environment, where frequent updates are performed. This distributed data store comprises of commodity PCs, while users need no proprietary tool to access it.

- Advanced features that allow our system to adapt its behavior according to some strategy, by conforming either to the incoming workload or to the common observation that queries

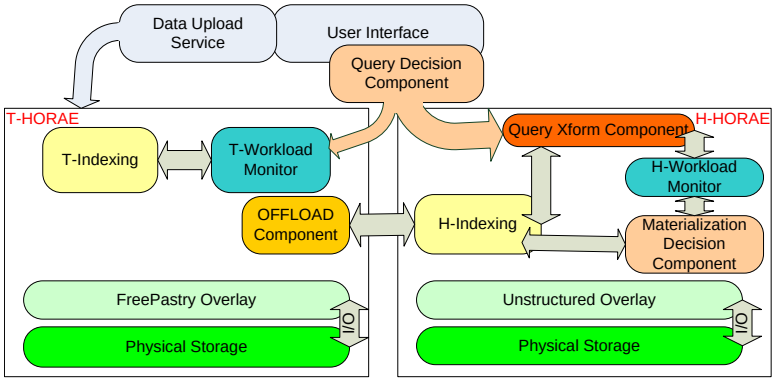


Figure 5.3: *The architecture of the* HORAE *system*

over time tend to be less detailed as time progresses. Both the granularity of materialization as well as the amount of dedicated resources are adjusted for best system performance and optimal storage utilization. Thus, query resolution is accelerated and storage consumption is minimized according to demand.

- A thorough validation of the proposed system using an actual deployment on both synthetic and real datasets. We measure our system's performance at all stages of its operation and compare it against Hive [TSJ+09], a state-of-the-art distributed warehousing initiative based on Hadoop [had]. Our findings show that our scheme, while comparable in the update procedure, greatly reduces query times and required storage, while providing advanced elasticity and availability operations according to the application or workload demands.

5.2 System Design

HORAE is an integrated approach that employs data warehousing and distributed systems techniques in order to organize and analyze multidimensional data in a scalable and efficient manner. It operates on time series data, produced at high rates and arriving in a (roughly) time-ordered manner† as they are created. As such, we define the system's goals to be both efficient and online update and querying capabilities, scalability, fault tolerance and ease of deployment.

The high-level architectural components of HORAE are shown in Figure 5.3. Our design comprises of two complementary subsystems, *T-HORAE*, which constitutes the "transactional" part, and *H-HORAE*, which processes the "historical" application data. The rationale for this distinction derives from our requirement for both efficient analytics and online processing of

†This is not a strict requirement for our system. We deal with out-of-order updates, assuming that the maximum amount of observed *lag* is bounded.

updates. Moreover, since we target time series data, we anticipate (although not require) that demand detail is fine-grained for more recent entries and gets more coarse for historical data. Thus, we provide a system design with a transactional logic (i.e., online, fast) that asynchronously materializes its contents inside the warehouse-logic component. The latter is designed so that any group-by query over big multidimensional datasets is efficiently handled.

T-HORAE is in charge of storing and indexing the incoming updates from multiple enterprise sources. Its function is to provide a "buffering" between high-rate updates and the materialized bulk of historical data kept in H-HORAE. Built on top of a DHT overlay, it offers a no-precomputation data insertion, enabling the inclusion of data as fresh as possible to query responses. Utilizing an indexing mechanism adaptive to workload skew, it also ensures high query throughput with small communication and computation overhead.

H-HORAE is the subsystem that stores the large bulk of data as they are timely transferred from T-HORAE. In order to allow for efficient analytical querying, H-HORAE materializes incoming tuples using a highly efficient cube index and distributes them across an unstructured overlay of nodes. An adaptive materialization engine ensures that the granularity of the created sub-cubes matches the workload patterns, while a load-based replication module provides availability and elastic system behavior.

In the following we analyze the components and operations of the two subsystems as well as the way they seamlessly cooperate for storage, indexing and analysis of time series data.

5.2.1 Data and Query Model

Our data spawns the d-dimensional space, with `Time` being the primary dimension. For simplicity reasons, we only consider hierarchies for the `Time` dimension in our analysis. However, our system can be generalized to support hierarchies in any dimension. `Time` is organized along L hierarchy levels ℓ_i, $0 \leq i \leq L-1$ with ℓ_0 corresponding to the most detailed level and ℓ_L being the special *ALL* ($*$) value. We define that ℓ_k lies *higher* (*lower*) than ℓ_l and denote it as $\ell_k > \ell_l$ ($\ell_k < \ell_l$) *iff* $k > l$ ($k < l$), i.e., if ℓ_k corresponds to a less (more) detailed level than ℓ_l (e.g., *Hour*>*Second*). We assume that our database comprises of fact table tuples of the form: $\langle \text{tID}, T_{\ell_{L-1}}, \ldots, T_{\ell_0}, D_1, \ldots, D_{d-1}, \text{fact}_1, \ldots, \text{fact}_s \rangle$, where T_{ℓ_i}, $0 \leq i \leq L-1$ is the value of the ℓ_i^{th} level of `Time` and D_j, $1 \leq j \leq d-1$ is the value of the j^{th} dimension of this tuple and fact_m, $1 \leq m \leq s$ are the numerical facts (we assume they correspond to the most detailed level of the cube).

Our goal is to index large and constantly incoming volumes of these tuples so that we can answer queries of the form: $q = \langle q_t, q_1, q_2, \ldots, q_{d-1} \rangle$, where each query element q_t can be a value from a valid hierarchy level of the `Time` dimension ($q_t = T_{\ell_i}$, $0 \leq i \leq L-1$) and each q_j can be any value of the j^{th} dimension, including the special $*$ value ($q_j = \{D_j, *\}$, $1 \leq j \leq d-1$).

Table 5.1: *Data and Metadata sample for our use case*

Time Hierarchy	tupleID		Time		Customer	Product	Sales
			Fact Table				
Hour	1	H_1	M_1	S_1	C_1	P_1	\$10
↑	2	H_1	M_1	S_1	C_1	P_2	\$20
Minute	3	H_1	M_2	S_2	C_2	P_2	\$30
↑	4	H_1	M_2	S_3	C_2	P_2	\$40
Second	5	H_2	M_3	S_4	C_1	P_1	\$50

Table 5.1 contains a sample fact table of 3 dimensions (`Time`, `Customer`, `Product`) and a measure of interest (`Sales`). It also includes the declaration of the hierarchy imposed on the `Time` dimension of the sample dataset.

5.2.2 T-HORAE Subsystem

The T-HORAE subsystem is based on *HiPPIS*. As described in Chapter 3, the main idea of *HiPPIS* is that peers initially index tuples using a default level combination (*pivot*). Inserted tuples are internally stored in a hierarchy-preserving manner. Query misses are followed by soft-state pointer creations so that future queries can be served without reflooding the network. Peers maintain local statistics which are used in order to decide if a reindexing to a different combination of hierarchy levels is necessary, according to the current query trend. *HiPPIS* is thus able to process queries of variable granularity.

Yet, the maintenance of soft-state indices can be very costly in the case of frequent updates: All affected indices must be checked after a tuple insertion among a large number of existing ones. Moreover, the reindexing cost grows proportionally to the size of the data, making changes in workload skew harder to handle. These limitations conflict with our requirements of a system targeted to time series data. We provide multiple algorithmic/design changes in T-HORAE to support high-rate updates and efficiently adapt to the incoming load: The indexing scheme gives more consideration to the `Time` dimension to better support time series data, the soft state indices are eliminated to facilitate fast updates and the internal data storage is structured in a way that offers more independence for more cost-efficient adaption to the various workloads. Below we describe the T-HORAE operations in detail.

Data Insertion/Update

Upon initial data loading or upon arrival of incoming update batches the *Data Upload* service is called. Indexing is performed by the *T-Indexing* module: The ID of each tuple to be originally inserted is the hash of the T_{ℓ_0} value, contrarily to *HiPPIS* where all dimensions, not only `Time`, contribute to the tuple ID. This choice consciously reflects the assumption that `Time` is the most important dimension, which will be included in the majority of the queries and that queries

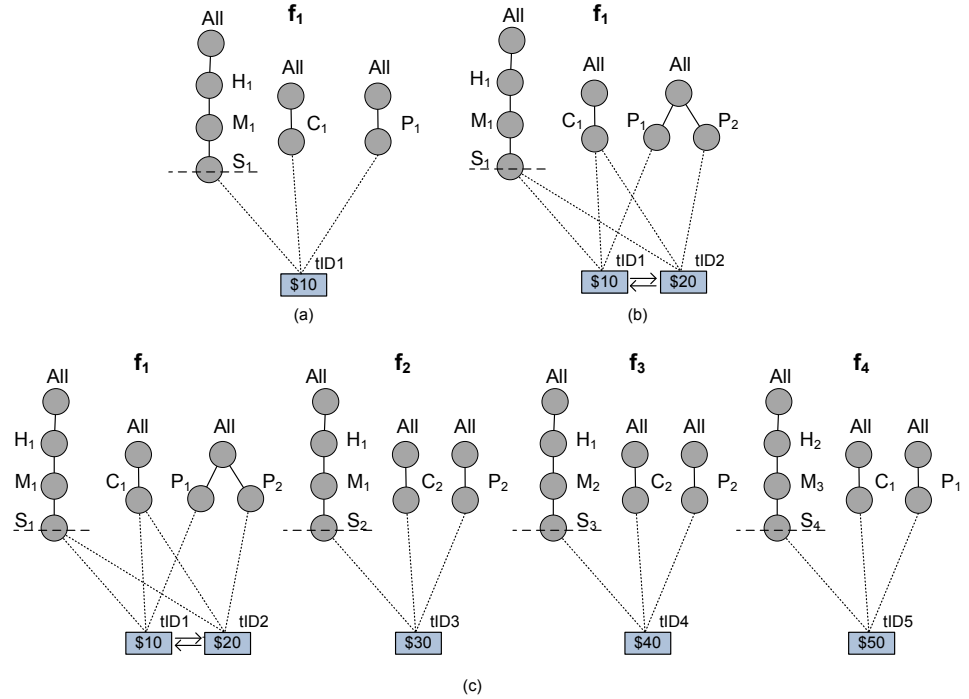


Figure 5.4: *The T-HORAE forests after the insertion of (a) the first, (b) the second and (c) all tuples of Table 5.1*

concerning recent events (i.e., those mostly kept at T-HORAE) will require the most detail. The DHT assigns each tuple to the corresponding node with ID numerically closest to this value.

As in *HiPPIS*, tuples are internally stored in forest-like structures, each of them consisting of d rooted trees, one for each dimension. The tree corresponding to `Time` has a height of L ($L-1$ levels plus the special ALL value $*$), while the rest $d-1$ trees have a height of 2. Each forest f is characterized by its *pivot* P_f, which defines the level of indexing and can be any of the ℓ_i, $0 \leq i \leq L-1$ values, excluding $*$. Indeed, if $P_f = *$, then all data would be stored in one node, defeating the purpose of our distributed overlay. The value corresponding to P_f is called *pivot value* T_f. In T-HORAE, each forest f can *independently* reindex its data to a different level, creating new forests with pivots different than P_f according to query trends. This mechanism is explained in detail later in the section.

For tuple updates, we must discover which forest to append each new tuple to, so that it is included in future queries. This translates to finding the forest with common path starting from the root of the `Time` tree and moving towards the leaves. This is achieved by consequent lookups starting from T_{ℓ_0} and moving towards $T_{\ell_{L-1}}$ until the first match is found. If no match is found, a new forest must be created with ℓ_0 as pivot. The process results in a cost of $O(L \log n)$ messages. Considering that the number of levels for *Time* is usually limited and that queries for recent

events tend to be more detailed (thus recently created forests mostly remain indexed according to ℓ_0), it is safe to assume that this cost is close to a simple DHT insertion.

Figure 5.4 shows an example of data insertion using the tuples of Table 5.1. When items with the same ID arrive at a host (Figures 5.4 a and 5.4 b), different values at levels lower in the hierarchy than the pivot create branches, while items with different IDs are indexed separately (Figure 5.4 c).

Data Lookup

Queries with $q_t = T_{\ell_i}$ that concern the P_f of a forest f are *exact match* queries and can be answered in logarithmic number of steps. More formally, assuming the query q with $q_t = T_{\ell_i}$ as defined before, the query is an *exact match* if $\exists f : \ell_i = P_f \wedge q_t = T_f$. Queries on `Time` values that are not indexed cannot be answered unless circulated around the overlay. This way, all nodes that contain (parts of) the answer are discovered and the corresponding answers are returned to the query initiator to perform local aggregations.

Assuming the initial state of Figure 5.4 c, the query $\langle S_1, C_1, P_2 \rangle$ is forwarded to the node responsible for the hash of S_1. This is an exact match query and the value \$20 is returned. When querying for $\langle H_1, *, P_2 \rangle$, we discover that the key that derives from hashing H_1 does not exist. Flooding is performed and the nodes responsible for f_1, f_2 and f_3 answer with the initiator (assuming the aggregation function is `sum`) performing the addition and returning \$90.

Data Reindexing

While the initial P_f setting in T-HORAE gives preference to queries over a specific level, it is possible that some workloads will defy this choice over time. The *T-Workload Monitor* module of T-HORAE nodes actively monitors the current query trend by maintaining local statistics about the *popularity* of each level of `Time` per forest. As *popularity* of ℓ_i of a specific forest f we define to be the number of queries it has received regarding ℓ_i within the most recent time-frame W. This time-frame should be properly selected to perceive variations of query distributions and, at the same time, stay immune to instant surges in load. If the most popular level of f, ℓ_{max}, exceeds P_f in *popularity* by some *threshold*‡, the node considers the possibility of a new partitioning according to ℓ_{max}.

If $\ell_{\text{max}} < P_f$, then the node hosting f can autonomously make a decision about reindexing, since it holds all the data and corresponding statistics of all subtrees of f in `Time`. Indeed, any forest stores all values of the levels below the pivot and therefore has a global view of the queries regarding ℓ_{max}. *T-Indexing* reinserts all tuples of f according to the new pivot ℓ_{max}. This process splits the initial forest, creating as many new forests as the number of distinct `Time` values of ℓ_{max} belonging to f and scatters them across the overlay. Each of the new forests has ℓ_{max} as pivot.

‡The *threshold* value identifies the dispersion of the different popularity values. As such, we choose it to be proportional to the Mean Difference Δ of these, like in the *HiPPIS* case (see section 3.3.5).

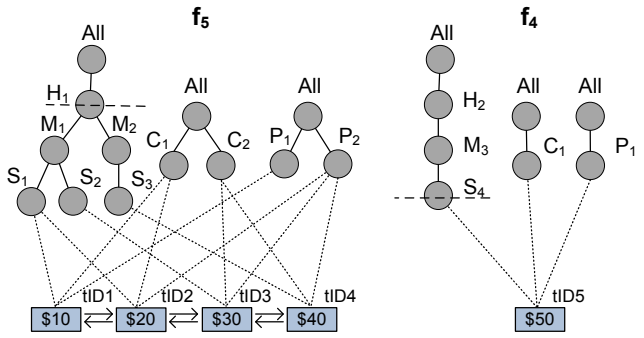


Figure 5.5: *Data distribution after a shift towards the value H_1*

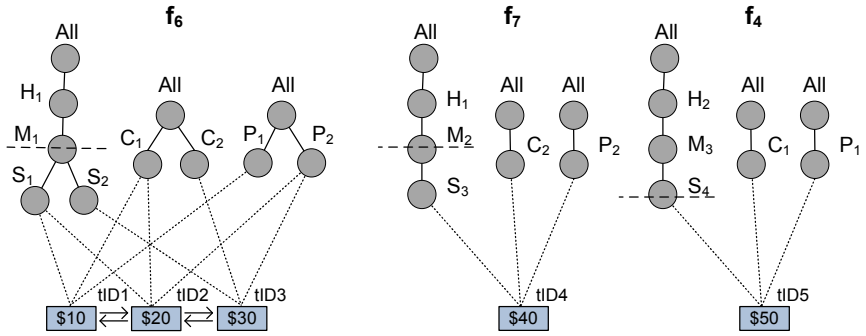


Figure 5.6: *Data distribution after a shift towards the level* `Minute` *of the forest f_5*

In the opposite case ($\ell_{\max} > P_f$), the node is not capable of making an individual decision, since the value of $T_{\ell_{\max}}$ is located in other forests as well. A *SendStats* global message signals the transmission of statistics for all forests (if any) containing $T_{\ell_{\max}}$. The initiator, after collecting them, checks if the condition that its *popularity* exceeds that of P_f by *threshold* holds. If this is the case, the initiator sends a *Reindex* message to all nodes that replied with statistics, resulting in the merging of the involved forests to one, with $\ell_{\max}$ as pivot.

In our example, assume that the node storing f_1 observes that `Hour` is more popular than `Second`, i.e., H_1 gets more queries than S_1. Statistics from nodes holding f_2 and f_3 have to be checked before making a final decision. If these statistics confirm the local findings, then reindexing is performed by all involved nodes (Figure 5.5). If, at some later point, the statistics of f_5 suggest `Minute` as the most popular level, the hosting node can individually decide to shift to that level and reindex its data (Figure 5.6). To ensure the correctness of operations during reindexing and to avoid simultaneous reindexings, a *Lock* message is flooded to signal the prevention of other reindexings while one is in progress.

Data Offload

T-HORAE is intended to store recent transactions due to its simple and efficient insertion/update mechanism. As time progresses, part of the T-HORAE data is moved to H-HORAE using

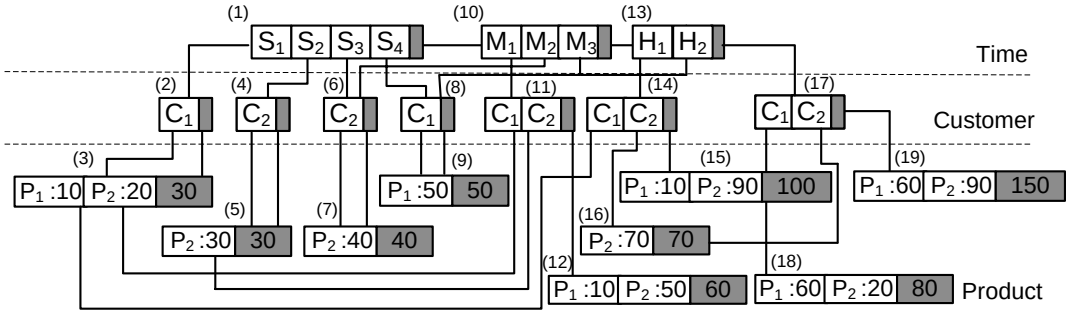


Figure 5.7: *Dwarf cube for the data of Table 5.1*

the *Offload* module to facilitate more powerful analysis. Our system dynamically regulates the frequency and size of data that the module operates on.

To understand the necessity of the module, let us refer to the initial requirements: New events to be stored arrive asynchronously from possibly multiple sources. T-HORAE handles these updates in batch mode (e.g., on a per-minute basis), since they must inevitably converge through some common channel (e.g., a web application). We define *lag* as the maximum delay between the most recent and oldest time-stamped data items inside a batch to be inserted. This is an important quantity, as it closely relates to the amount of buffering necessary by T-HORAE so that transferred data will be correctly processed by the more strict H-HORAE indexer. Larger *lag* values force T-HORAE to wait more before offloading (to avoid out-of-order updates), while smaller values could trigger a more "aggressive" strategy. Finally, to keep a roughly stable size of data in T-HORAE, tuples must leave the subsystem at a rate proportional to the rate they enter it. Thus, another parameter that influences the offload process is the update rate λ_{upd}.

Our choice is to invoke the offload procedure periodically, every T_{off}, while the exact division of data between the two components is governed by the W_{rem} parameter: A value of $W_{\text{rem}} = 1$ hour means that T-HORAE stores transactions that occurred within the last hour. Obviously, $W_{\text{rem}} > T_{\text{off}}$, as we wish to take into consideration the time necessary to index the offloaded data, while answering queries. Thus, our system dynamically adjusts these two parameters according to the following formulas: $T_{\text{off}} = c_{\text{off}} \cdot \max\{\frac{1}{\lambda_{up}}, lag\}$ and $W_{\text{rem}} = c_{\text{rem}} \cdot \max\{\frac{1}{\lambda_{\text{upd}}}, lag\}$, where $c_{\text{rem}} > c_{\text{off}} > 1$.

5.2.3 H-HORAE Subsystem

The H-HORAE subsystem serves as an archival storage of fully or partially summarized historical data. Its design modifies the *BD* structure in a way that favors frequent updates over temporally ordered information, produced in a distributed manner at a high rate and adjusts the granularity of the materialization according to a predifined strategy: Either the data's recency or the incoming query workload.

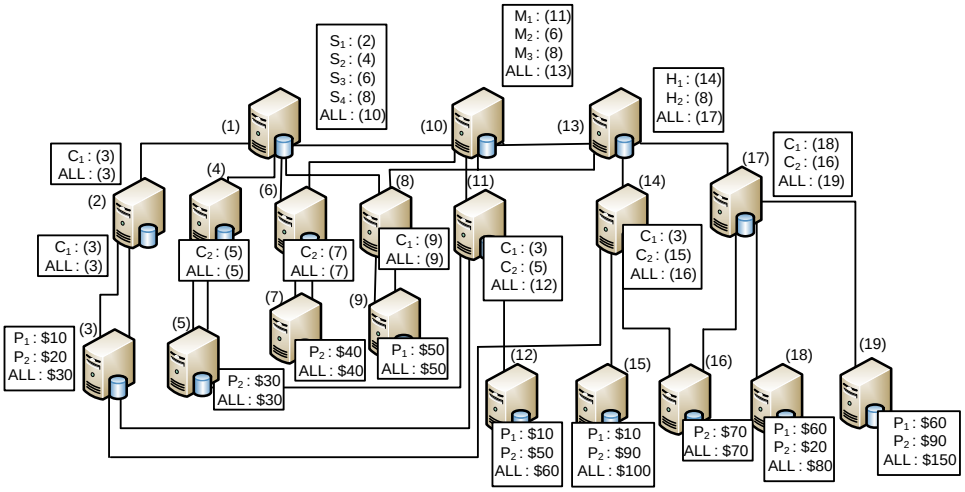


Figure 5.8: *The distribution of dwarf nodes in the* Brown Dwarf *structure of Table 5.1 and their hint tables*

BD has proved that it greatly increases scalability and performance (through parallelization) while enabling the computation of much larger cubes (see Chapter 4). However, the update procedure is costly in terms of time as well as bandwidth due to the a-priori materialization. Therefore updates are usually performed in batches. Even so, sometimes it is more efficient to reconstruct the whole dwarf structure from scratch. Moreover, it precludes the processing of multiple-granularity queries on the time hierarchy. H-HORAE modifies *BD* to accommodate time series data: By re-organizing the indexing structure according to `Time`, it allows for a design that favors frequent updates and adaptation to workload trends. The latter is a completely new feature that enables H-HORAE to vary its operational gains between storage and precision in a totally customizable way.

Concisely, the top dimension of the structure is that of `Time` and tuples are inserted according to the most detailed level of it. No *ALL* cell is calculated for `Time`; instead, a daemon process periodically constructs roll-up aggregates for values that lie higher in the hierarchy, optionally erasing the lower level ones for cube size reduction.

Apart from the benefits that the distribution of the *Dwarf* offers, such as the acceleration of the cube construction, the ability to store much larger cubes and the dramatic reduction in query response times, H-HORAE allows for online, cost-efficient updates that can originate from any host accessing the update service of the system even at very high rates. Moreover, it makes better use of the available storage and bandwidth, as the granularity of aggregation can be adjusted, according to the application needs.

Insertion

The insertion operation refers to the initial cube creation using historical data of the past and is undertaken by the *H-Indexing* component. `Time` is chosen first in the dimension ordering, while

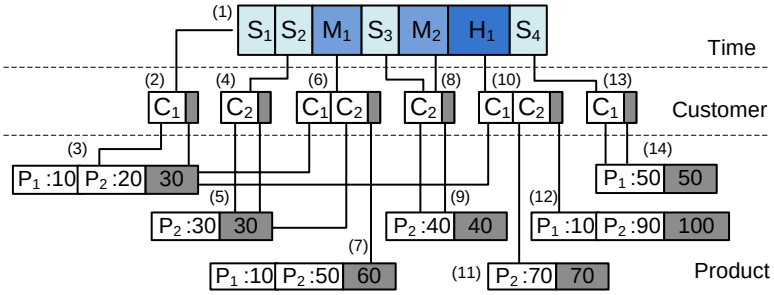


Figure 5.9: *The full H-HORAE cube for the fact table of Table 5.1*

the rest of the dimensions are placed in a descending cardinality order. As proven in the original paper, dimensions with higher cardinalities result in a smaller *Dwarf* cube when placed on the higher levels of the Dwarf cube. The construction is preceded by a sort on the fact table, as in the original *Dwarf*. Figure 5.7 depicts the *Dwarf* structure according to the original algorithms for the data of Table 5.1, while the corresponding *BD* distribution is presented in Figure 5.8 for comparison reasons.

An important difference of our system's cube construction is the lack of the *ALL* cell in the `Time` dimension. Tuples are being processed one by one, first according to ℓ_0 (`Second` in our example). As soon as all the values of ℓ_0 that are mapped to the same value of ℓ_1 have been processed, our algorithm creates an *aggregate* cell for that specific value of ℓ_1. This procedure is followed for all L levels of the hierarchy. In general, the aggregate of a value of ℓ_i is created by calling the `SuffixCoalesce` routine of *Dwarf* (see [SDRK02]), providing as input the set of dwarf sub-cubes of the ℓ_{i-1} that correspond to the specific value of ℓ_i. For the sample data of Table 5.1, after having created the nodes and cells for the first two tuples according to the original cube construction algorithm, when proceeding to the third tuple the system realizes that all tuples belonging to M_1 have been processed, thus it creates the aggregate cell and the corresponding subdwarf for M_1. Similarly, upon reception of the last tuple, the system constructs the subdwarf for Y_1. The final graph can be seen in Figure 5.9, while its distribution over the nodes of an unstructured overlay in Figure 5.10. Note that the highest level of aggregation is defined by the highest level of the hierarchy and no *ALL* cell exists for the `Time` dimension.

Querying

Queries are resolved by following their path along the system attribute by attribute. A node initiating a query $q = \langle q_t, q_1 \dots q_{d-1} \rangle$ forwards it to the root dwarf node of the distributed structure (N_{root}). There, the hint table is looked up for q_t under *currAttr*. If q_t is of ℓ_0 and exists, *child* is the next node the query visits. The above procedure is followed until a measure is reached. Since adjacent dwarf nodes belong to overlay neighbors, the answer to any point or group-by query is discovered within at most d hops. The same procedure is followed if q_t is of a level ℓ_i, $i \neq 0$ and

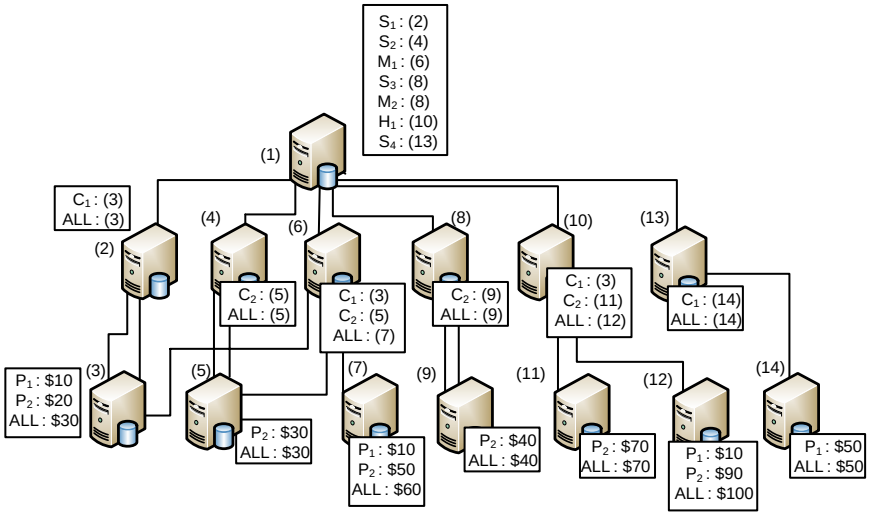


Figure 5.10: *The distribution of the H-HORAE cube of Figure 5.9*

this aggregate exists, with the answer being reached within d hops. If, however, the aggregate cell has not yet been created, the initial roll-up query must be substituted by multiple queries of ℓ_{i-1}. Let $V_{\ell_{i-1} \to q_t}$ be the set of values of ℓ_{i-1} that are descendants of the queried value of ℓ_i, $i \neq 0$. Then the requester must issue $|V_{\ell_{i-1} \to q_t}|$ queries for each value belonging to $V_{\ell_{i-1} \to q_t}$, keeping q_j, $1 \leq j \leq d-1$ the same as in the original query, gather the results and spend some post-processing to calculate their aggregate. If the aggregate for some value of ℓ_{i-1} does not exist, the query is further analyzed, until ℓ_0 is reached. In the worst case, the answer is at most $d \cdot |V_{\ell_0 \to q_t}|$ hops and an aggregation away. The querying process is orchestrated by the *Query Xform* module, through which pass all queries concerning H-HORAE.

In our sample case, a query for $\langle M_1, ALL, P_1 \rangle$ follows the path (1)→(6)→(7) and returns \$10, while $\langle M_3, C_1, P_1 \rangle$ is translated to $\langle S_4, C_1, P_1 \rangle$ through the metadata information and returns \$50 after visiting (1), (13) and (14).

Updating

This is an important operation, as, by nature, time series data undergo very frequent updates. Assuming that already inserted tuples are read-only and can neither be changed nor deleted, as the common practice in data warehouses dictates, the update procedure translates to the insertion of new tuples in the existent cube. In H-HORAE, we require that no updates for past events are posed to the system. T-HORAE's offload component makes sure that this requirement is met by offering batches within which all arriving updates are sorted before their integration with the existing materialized H-HORAE structure.

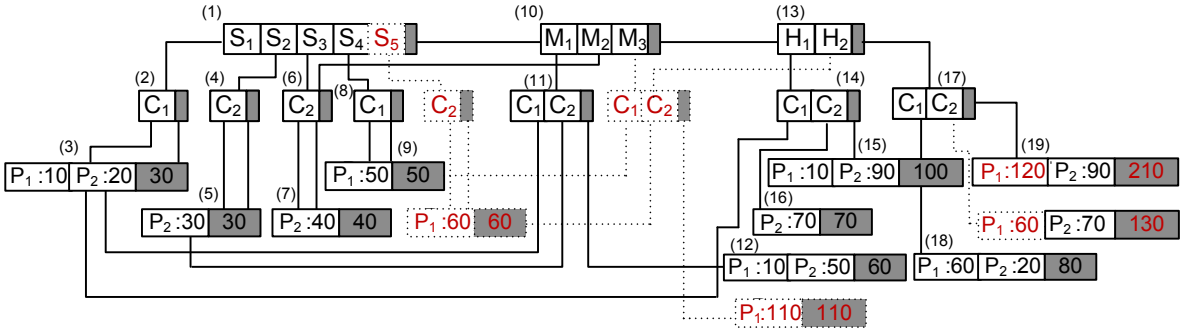


Figure 5.11: *The effect of the insertion of a new update tuple* ($\langle H_2, M_3, S_5, C_2, P_1, \$60\rangle$) *on the original* Dwarf *cube*

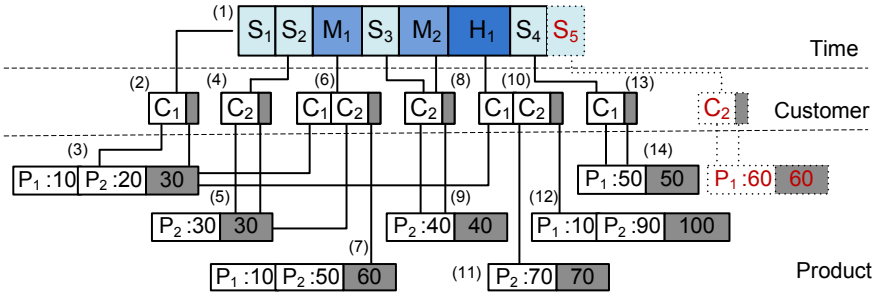


Figure 5.12: *The effect of the insertion of a new update tuple* ($\langle H_2, M_3, S_5, C_2, P_1, \$60\rangle$) *on the H-HORAE structure*

The difference is that now the longest common prefix between the new tuple and existing ones must be discovered following overlay links. Once the network node that stores the last common attribute is discovered, underlying nodes are recursively updated. This means that nodes are expanded to accommodate new cells for new attribute values and that new dwarf nodes are allocated when necessary. As in the insertion case, tuples are initially inserted in ℓ_0 and as soon as a value of ℓ_i, $i \neq 0$ is complete, the specific aggregate is constructed.

The important benefit of H-HORAE compared to *BD* is that it significantly reduces the update cost, due to the arrival of tuples in temporal order, combined with the lack of an *ALL* cell in the first dimension. The temporal order guarantees that the first attribute of the new tuple will either create a new cell in the first dimension, or coincide with the last cell of N_{root}. Therefore, no aggregate cell of `Time` will be affected. This is not the case for the rest of the dimensions though, where all the affected aggregates are recalculated.

In our example, if $\langle H_2, M_3, S_5, C_2, P_1, \$60\rangle$ arrives, the system will create a new cell in (1) for S_5 and two new dwarf nodes for the rest of the attribute values, while all other nodes remain unaffected. Contrarily, in the original *Dwarf* structure of Figure 5.7, 12 dwarf nodes would be accessed. The effect of the update procedure on both the original *Dwarf* and the H-HORAE distributed cube is illustrated in Figures 5.11 and 5.12 respectively. The created cells and nodes

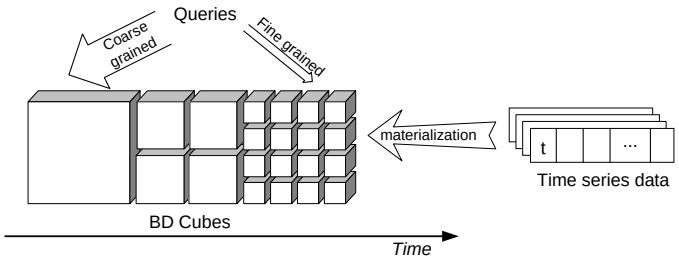


Figure 5.13: *The time-driven materialization mechanism of H-HORAE*

along with the newly added links are depicted with dashed lines, while both the inserted and the updated attributes and fact values are depicted in red. It is thus apparent that updates in our system are less costly, as they affect fewer dwarf nodes and cells compared to the original *Dwarf*.

Adaptive Materialization

The proposed system, as described above, follows a static strategy for materialization: A roll-up view in the `Time` hierarchy is created as soon as all required data are available, without destroying the drill-down views, which remain in the system. Instead, more dynamic approaches can be adopted.

Time-driven materialization In data warehousing applications involving temporal data it is often the case that recent data are demanded in a fine-grained manner, whereas queries for historical data usually concern aggregated periods of time. A daemon process periodically and asynchronously creates the roll-up views and erases the corresponding drill-down ones. The period of this process is chosen taking into account the characteristics of the application. Therefore, the materialization process gradually follows the roll-up path and eliminates the more detailed views as time passes, as depicted in Figure 5.13.

The time thresholds beyond which a roll-up view is created and the corresponding finer grained ones are erased are denoted as $(T_{\ell_1}, T_{\ell_2} \ldots T_{\ell_{N-1}})$. This practically means that for records stored more than T_{ℓ_i} the system constructs and maintains only the aggregate views that belong to ℓ_i, erasing all views from ℓ_0 to ℓ_{i-1}. The values of the thresholds can either be set by the administrator a priori, taking into account the needs of the specific application or dynamically adjusted according to the monitored query trend.

Load-driven materialization Depending on the application, not all data are queried upon in the same level of granularity. N_{root} maintains statistics for the query load of each granularity level through the *H-Workload Monitor* and the *Materialization Decision* component asynchronously creates aggregate views for popular values and erases the detailed ones if they are infrequently requested. The query load thresholds above which a roll-up view is created and below which the finer grained ones are erased are denoted as $(TMat_{\ell_1}$,

$TMat_{\ell_2}, \ldots, TMat_{\ell_{L-1}}$) and ($TDel_{\ell_0}$, $TDel_{\ell_2}, \ldots, TDel_{\ell_{L-2}}$) respectively and can be set according to the needs of the application.

In the case of adaptive materialization, there exists a trade-off between the size and the complexity of the cube, which consequently affects update and query response times as well as the accuracy of the responses. Creating and erasing the aggregate views periodically spares significant amount of storage space. However, the main advantage of the method is the acceleration of the updates and the increase in query throughput, due to the smaller overall size of the distributed cube: Keeping dwarf cubes small helps the system navigate more quickly through them. On the other hand, queries that follow an opposite trend than the one expected either are more costly, or cannot be answered accurately. More precisely, coarse grained queries concerning very recent events that only exist in the finest granularity translate to multiple fine grained ones. The system, apart from the bandwidth cost, has to pay a post-processing cost as well, calculating the aggregate of the gathered responses.

Moreover, the erasure of fine grained aggregate views leads to irreversible information loss. Therefore fine grained queries concerning them are only answered in approximation, for instance with the ratio of the aggregate fact to the number of distinct values of the queried level that belong to the data aggregation level. Orthogonal approximation methods (e.g., [DGR07, Gar06, GKMS03] etc.) can be applied to mitigate this fact.

5.2.4 Replication

Replication is an important mechanism to achieve scalable performance, especially under heavy loads, as well as fault tolerance. HORAE provides replication in both its modes. In T-HORAE, replication and load balancing are handled by the underlying DHT, while H-HORAE adopts the *BD* replication scheme, which is adaptive to both node churn and data skew. Nodes periodically ping each other and replace replicas hosted by failed peers, preserving the degree of data redundancy above k. Moreover, monitoring its load on a per dwarf node basis, H-HORAE creates additional mirrors of overloaded nodes and deletes underloaded ones (with load above a $Limit^s_{exp}$ and below a $Limit^s_{shr}$ threshold respectively) through the *expansion* and *shrink* process.

5.3 Experimental Evaluation

In this section we provide a thorough evaluation of HORAE, testing its behavior both per component and as a whole. T-HORAE is based on a heavily modified version of FreePastry [fre], although any DHT implementation could be used as a substrate. H-HORAE is written in Java, using the socket API for inter-node communication. HORAE is deployed on an actual test-bed of $n = 16$ commodity nodes of our lab infrastructure (Quad Core @ 2.0 GHz, 4GB RAM).

Hive [TSJ+09] (version 0.5.0) has been deployed in the same testbed for direct comparison: 15 worker nodes (spawning 2 Mappers and 2 Reducers running concurrently, given 512 MB RAM each) and a single machine in the role of HDFS, MapReduce and HBase master. For a fair comparison, replication in HDFS was turned off.

In our experiments, we use both synthetic and real datasets consisting of a fact table representing multidimensional time series data with numerical facts. The synthetic datasets have been generated with our own and the APB-1 benchmark generator [apb]. Our generator creates the tuples of the fact table to be stored from combinations of the different values of each dimension, set to 1000 by default, plus a randomly generated numerical fact. For `Time`, which consists of 3 hierarchy levels, `Second`<`Minute`<`Hour`, the value for each tuple is chosen using a Poisson distribution of $\lambda = 5\frac{\text{tuples}}{\text{sec}}$. Furthermore, we may choose to create tuples that combine dimension values uniformly or with bias (creating zipfian distributions with $\theta = 1$).

Using the APB-1 benchmark generator [apb] we produce three 4-d datasets (A, B and C with densities 0.1, 0.2 and 0.3 respectively) with dimension cardinalities 24, 9000, 900 and 9 and one measure attribute. The `Time` dimension consists of 3 hierarchy levels, `Month`<`Quarter`<`Year` and covers the period from January 1995 to June 1996. The real dataset contains publicly available network audit data for the 1998 DARPA Intrusion Detection Evaluation Program [dar98]. It includes 1.1 million records, collected over a period of 6 weeks and organized in 7 dimensions. The `Time` hierarchy is organized along `Second`<`Minute`<`Hour`<`Day`.

The aggregate function used in the results is `sum`. For the application workloads, we include both point and aggregate queries with varying granularities and distributions, as well as continuous updates.

Experiments are conducted using both the static and the adaptive mode of *H-HORAE*. In the former case, denoted as H_f (the full version of H-HORAE), materialization is performed for all hierarchy levels in a synchronous way. In the latter case, H_{ad}, we have implemented both the proposed strategies, namely the time-driven and the load-driven. In the time-driven case, denoted as H_{ad_t}, materialization is performed through a daemon process, with thresholds statically set according to the dataset, whereas in the load-driven case, denoted as H_{ad_l} aggregates are created and erased according to the query workload. For direct comparison we have also conducted experiments with the centralized *Dwarf*, as well as its distributed approach, *BD*.

5.3.1 Data Load

The first set of experiments evaluates the initial data load in each of the system's components in terms of completion time as well as cost in both storage and communication overhead.

In Table 5.2, we directly compare all versions of *H-HORAE* with the original *Dwarf* and its distribution, *BD*, when initially inserting the APB and DARPA datasets. H_f synchronously

Table 5.2: *Measurements for various dataset insertions*

		size (MB)					time (sec)					load (msg/insertion)			
dataset	#Tup.	*Dwarf*	*BD*	H_f	H_{ad_t}	H_{ad_l}	*Dwarf*	*BD*	H_f	H_{ad_t}	H_{ad_l}	*BD*	H_f	H_{ad_t}	H_{ad_l}
APB-A	1.2M	56	59	53	9	10	485	101	100	57	31	2.3	1.5	0.3	1.2
APB-B	2.5M	102	115	93	24	25	957	220	198	123	71	2.4	1.7	0.4	1.4
APB-C	3.7M	163	182	146	32	34	1530	321	289	167	104	2.4	1.6	0.4	1.5
DARPA	1.1M	178	191	156	127	143	614	222	208	189	103	5.9	5.2	1.2	4.9

creates rollup aggregates upon completion of a hierarchy level value. In H_{ad_t} materialization is performed through a daemon process, with thresholds statically set according to the dataset: For the APB datasets, (T_{ℓ_1}, T_{ℓ_2}) is set to (1 quarter, 1 year) and for the DARPA dataset (2 hours, 2 days). In this case we assume that only the aggregates that conform to the T_{ℓ_i} thresholds are being created in the first place. For instance, for the DARPA dataset this means that records of the last 2 hours are stored in the granularity of minutes, of the last 2 days in the granularity of hours and the rest as days. H_{ad_l}, initially stores data only in the most detailed level. The creation of roll-up views (and the deletion of drill-down ones) are not considered part of the loading procedure of H_{ad_l}, since it takes place periodically and independently of the insertion, based on the incoming query load.

We observe that H_f noticeably reduces the size of the created cube compared to *BD*, providing cubes smaller even than the ones generated by the centralized *Dwarf*, due to the lack of the *ALL* cell in the first dimension. The reduction is even more impressive with H_{ad_t}, reaching 82% for the APB and 34% for the DARPA datasets. In this case, the choice of the T_{ℓ_i} thresholds plays a decisive role in the cube size reduction: The smaller the T_{ℓ_i}, the more coarse grained the views of the stored data, thus the smaller the cube size. In the worst case, the size of a H_{ad_t} cube is equal to that of H_f. The size of the H_{ad_l} cube is comparable to that of H_{ad_t}.

Analogous to the cube size reduction is the acceleration of creation time. First, we confirm that the distribution of the cube leads to a better exploitation of existing resources, enforces parallelization and thus reduces cube construction times. H_f and H_{ad_t} exhibit a further reduction (reaching 10% and 45% respectively) compared to *BD*. H_{ad_t} is 89% and 70% faster in storing the cube compared to the centralized system using the APB and DARPA datasets respectively. The absence of the *ALL* cell in `Time` in both cases and the selective materialization of H_{ad_t} result in smaller cubes with less dwarf nodes, thus requiring less communication and I/O cost. This is in line with the results concerning the messages per tuple insertion. H_{ad_l} exhibits even more impressive acceleration, reaching 93% compared to *Dwarf* for APB-C. Thus, H_{ad_l} proves faster than H_{ad_t}, although more costly in messages. This is due to the fact that no aggregation cell is calculated for the `Time` dimension, dramatically reducing the time of the data load operation.

Table 5.3: *Initial load of 100K tuples of various dimensionalities in the T-HORAE and H-HORAE subsystems*

		Fact Tbl	size (MB)			time (sec)				load (msg/insertion)		
	d	(MB)	T	H_f	H_{ad_l}	T	H_f	H_{ad_l}	Hive	T	H_f	H_{ad_l}
uniform	5	380	355	240	230	31	40	31	8	3	9	7
	10	810	725	635	565	38	132	115	9	3	15	12
	15	1320	1135	1125	1020	45	297	257	9	3	21	18
	20	1815	1645	1635	1475	52	492	420	9	3	29	25
zipf	5	380	345	225	217	35	42	35	9	3	11	7
	10	815	718	595	545	38	187	146	9	3	16	13
	15	1325	1080	990	956	52	520	378	9	3	22	19
	20	1810	1520	1465	1385	65	950	812	9	3	31	27

Table 5.3 presents results for the initial insertion of 100K tuples of various dimensionalities and distributions in T-HORAE, H-HORAE and Hive. Considering that H_{ad_l} represents a more adaptive and automatic strategy for materialization, we only test the H_{ad_l} mode.

For T-HORAE, the resulting structure is roughly the same size as the fact table for low-dimensionality datasets. As dimensions and skew grow we observe an increasing compression, due to the elimination of prefix redundancies in each dimension. As far as H-HORAE is concerned, it noticeably reduces the size of the created cube compared to the original fact table, due to both Dwarf's redundancy elimination and the lack of the *ALL* cell in the first dimension. The storage gain is more evident for low dimensional datasets (about 40% in the 5-d case) and decreases as dimensionality rises. H-HORAE proves less storage consuming than T-HORAE as well, even though it stores the materialized cube. The compression is even more apparent with the adaptive version (an additional 10% compared to the H_f cube for the majority of cases).

As far as the insertion time and load are concerned, T-HORAE achieves high efficiency and low bandwidth cost thanks to the simplicity of its insertion mechanism. As the number of dimensions increases, there is a slight increase in the total insertion time as well, due to the longer processing time per tuple needed to create the internal forest structure. However, time is not directly proportional to the dimensionality due to the parallelization, since multiple nodes process disjoint parts of the dataset. The average number of messages per insertion remains stable for all datasets, translating to a steady load per node. We also observe an increase (less than 20%) in insertion time for the more biased dataset, as dense areas create larger forests which take longer to process.

H-HORAE is naturally slower, since aggregations are calculated on the fly. Even in the H_{ad_l} case, although no materialization in `Time` is performed, aggregations in all other dimensions still take place. Hive outperforms both our methods in insertion, however its demonstrated times

Table 5.4: *Measurements for 10k updates over various datasets*

	time(sec)					load (msg/update)			
dataset	*Dwarf*	*BD*	H_f	H_{ad_t}	H_{ad_l}	*BD*	H_f	H_{ad_t}	H_{ad_l}
APB-A	1123	603	404	315	316	21.5	9.1	8.2	8.2
APB-B	1158	611	418	323	318	23.1	10.3	8.8	8.8
APB-C	1203	624	424	328	321	25.2	10.9	9.1	9.2
DARPA	1535	649	458	380	375	28.6	13.6	9.3	9.2

are comparable to those of T-HORAE. This shortcoming is amortized by the remarkable gain in querying time, shown later in the experimental section.

5.3.2 Incremental Updates

In this set of experiments, we test the behavior of HORAE under continuous updates. First, to compare the various versions of H-HORAE we apply 10k tuple insertions over the APB and DARPA datasets and measure the total time needed to process the update batch as well as the communication cost of the procedure. Table 5.4 summarizes the results. It is worth noting that, in the case of H_f, the update process includes the creation of roll-up aggregate views. This is not the case for H_{ad_t} nor H_{ad_l} though, where the creation of roll-up views and the deletion of drill-down ones take place asynchronously.

All three modes of *H-HORAE* drastically improve the update performance, accelerating the process compared to the centralized *Dwarf* and *BD*. H_f is about 3 times faster than the central algorithm and over 30% more efficient than *BD*. Apart from the parallelization of the process, which is enabled through the distribution of the cube structure, the updates come in order, guaranteeing that no update affects an already created roll-up view. Furthermore, recursive updates of affected *ALL* cells take place only in the dimensions other than the first, reducing the communication cost. Indeed, the required messages per update drop almost 3 times compared to *BD*. Both H_{ad_t} and H_{ad_l} prove even faster (about 20%) and more cost-efficient than H_f, due to the fact that materialization in the various hierarchy levels is performed asynchronously. Lastly, from the APB datasets we conclude that the larger the cube, the more the nodes and cells affected by updates, thus the more costly the process in terms of time and messages.

Next, we pose a batch of 20K update tuples of various dimensionalities as well as distributions into each of the T-HORAE, H-HORAE and Hive subsystems, which already contain a cube structure of 100K tuples. Results are presented in Table 5.5. Again, we chose to present measurements only for the load-driven strategy of H-HORAE, since, as proven by Table 5.4, its performance is almost identical to that of the time-driven version.

At this point we note that for T-HORAE the update operation is the same as the insertion, therefore results are similar to the ones presented in the previous set of experiments. As for

Table 5.5: *Insertion of 20K update tuples of various dimensionalities in the T-HORAE and H-HORAE subsystems*

		Time (sec)				load (msg/update)	
	d	T	H_f	H_{ad_l}	Hive	T	H_f	H_{ad_l}
uniform	5	9	789	662	6	3	12	10
	10	11	803	690	6	3	23	19
	15	13	822	718	6	3	35	31
	20	18	847	743	7	3	42	35
zipf	5	10	792	667	6	3	12	10
	10	11	809	698	6	3	24	19
	15	15	831	727	6	3	37	32
	20	19	859	755	7	3	44	36

Table 5.6: *Cost of materialization from seconds to minute and from minutes to hour granularities*

d	materialization	time(sec)	$size_{bef}$	$size_{after}$	Δsize
10	sec$\rightarrow$ min	0.3	35K	42K	7K
20	sec$\rightarrow$ min	0.5	75K	119K	44K
10	min$\rightarrow$ hour	4.1	438K	1.7M	1.3M
20	min$\rightarrow$ hour	9.2	2.6M	8.1M	5.5M

H-HORAE, both its modes perform updates significantly slower than T-HORAE, because the process is much more complex, requiring recursive updating of all aggregated measures involved. Contrarily, updating T-HORAE is as easy as hashing a tuple and inserting it using the underlying DHT. The adaptive mode of H-HORAE proves faster (about 15%) and more cost-efficient than the full one, due to the asynchronous materialization in the various hierarchy levels. However, updating remains by orders of magnitude more costly than in T-HORAE, which constitutes the main argument for the use of T-HORAE as a reception component whose buffering helps in the amortization of this cost. In all cases, the simple nature (single table) of the dataset enables Hive to perform updates in an extremely efficient manner regardless of the number of dimensions or level of skew.

Table 5.6 aims to show the overhead of the adaptive materialization of H-HORAE. More specifically, it reports on the time as well as the increase in storage when materializing to `minute` granularity from the 60 corresponding seconds and to `hour` from the 60 corresponding minutes for the 10-d and 20-d cases. The reported numbers are indicative of the cost of the materialization that is periodically executed and show that the increase in time and size closely relates to the degree of materialization (the higher the aggregation step the bigger the cost) as well as the input cube.

It is lastly worth noting that the updates come in order, guaranteeing that no update will affect an already created roll-up view. This motivates once again the existence of T-HORAE as a buffer before H-HORAE.

5.3.3 Querying

We now investigate the query performance of HORAE in the static as well as the adaptive modes compared to that of *BD*, using exclusively the DARPA dataset (since results for the APB datasets are qualitatively similar). Furthermore, we examine the overhead introduced by the lack of the *ALL* cell in the `Time` dimension and the information loss caused by erasing fine-grained views in the case of H_{ad_t} and H_{ad_l}. H_{ad_l} creates and erases aggregates according to demand. We assume the materialization threshold for `minute` is 5 $\frac{\text{queries}}{\text{sec}}$ and for `hour` 10 $\frac{\text{queries}}{\text{sec}}$, while the deletion thresholds for both `second` and `minute` is 0. Using the proper terminology, ($TMat_m$, $TMat_h$) = (5, 10) and ($TDel_s$, $TDel_m$) = (0, 0). For H_{ad_t}, as before, (T_{ℓ_1}, T_{ℓ_2}) is set to (2 hours, 2 days).

For this purpose, we pose 4 different querysets (DARPA$_1$, DARPA$_2$, DARPA$_3$ and DARPA$_4$) consisting of 1k queries each. DARPA$_1$ is a workload that ideally conforms to the set of T_{ℓ_i} thresholds of H_{ad_t}. DARPA$_2$ and DARPA$_3$ are created by following a two-step approach. First, a tuple of the dataset is selected using a zipfian distribution of $\theta = 1$, favoring the most and least recent records respectively. Then, the level of `Time` to be used in the query is selected according to the same biased distribution. This way, recent records are queried upon in more detail than older ones for DARPA$_2$, while DARPA$_3$ contains more fine-grained queries for past events. DARPA$_4$ follows the uniform distribution. For all querysets, we set $P_d = 0.3$, which we define as the probability of a dimension not participating in a query.

As seen in Table 5.7, DARPA$_1$ does not affect the query throughput nor the per query communication cost of H_{ad_t}, since all queried levels exist. However, as the queryset approximates the inverse distribution than the one assumed by our thresholds (DARPA$_3$), the messages needed to answer a query increase, naturally affecting the response times. This is attributed to the fact that queries concerning aggregates of recent data cannot be answered directly, since these aggregates have not been constructed yet, but need to be translated to multiple queries of a lower hierarchy level, thus issuing more messages. H_{ad_l} is more costly for the first two workloads, since aggregations do not exist a priori, but are constructed on demand. However, for DARPA$_3$, H_{ad_l} proves more efficient: Once the aggregations are calculated, queries concerning them are answered directly.

For the adaptive versions of H-HORAE, it is possible that a query concerns a level of `Time` that is no longer available. In such an event, only an approximation of the answer will be returned. H$_{ad_t}$ accurately answers all queries of DARPA$_1$, since none of them concern an erased

Table 5.7: *Measurements for various workloads over the DARPA dataset*

	time (sec)				load (msg/query)				%Inaccuracy	
queryset	*BD*	H_f	H_{ad_t}	H_{ad_l}	*BD*	H_f	H_{ad_t}	H_{ad_l}	H_{ad_t}	H_{ad_l}
$DARPA_1$	5	6	6	10	7	7	7	54	0%	0%
$DARPA_2$	5	9	8	164	7	9	12	57	19%	2%
$DARPA_3$	5	12	248	172	7	13	387	62	45%	2%
$DARPA_4$	5	24	21	201	7	16	32	354	32%	24%

Table 5.8: *Querying times and cost for of 1K querysets of various distributions in the two subsystems*

		time (s)				load (msg/q)			#reind	%Precicion	%Inaccuracy
Q	d	T	H_f	H_{ad_l}	Hive	T	H_f	H_{ad_l}	T	T	H_{ad_l}
	5	124	6	6	22738	3	5	7	0	98	0
Q_1	10	127	7	7	22371	3	9	9	0	98	0
	15	128	7	8	21847	3	14	14	0	98	0
	20	133	10	10	21529	3	18	19	0	98	0
	5	127	17	254	22737	5	5	158	5	91	3
Q_2	10	128	18	246	21374	4	10	343	5	92	4
	15	130	19	272	22333	5	16	424	5	91	3
	20	131	24	298	21736	5	21	693	5	91	3
	5	189	10	195	21293	114	6	170	2	34	28
Q_3	10	192	10	199	22382	115	12	358	1	33	27
	15	193	12	201	21746	115	18	450	1	33	27
	20	208	15	205	22067	114	24	704	1	33	28

level. However, for querysets that do not conform to the assumption that the temporal granularity of the posed queries is relative to the time the query is being posed, the percentage of queries answered in approximation rises significantly. Even for the uniform query distribution, it reaches 32%. It becomes apparent that in order to achieve the lowest possible inaccuracy, the T_{ℓ_i} thresholds need to fit the expected workloads. Therefore, the dynamic threshold selection according to the monitored query trend is chosen as a more suitable solution in cases where there is no intuition about the incoming workload. Indeed, in all cases the inaccuracy rate remains in lower levels when adopting the load-driven policy.

Testing the query performance of both subsystems for various dimensionalities and distributions, for all uniform datasets we create 3 different querysets (Q_1 through Q_3). Q_1 and Q_2 are created similarly to $DARPA_2$ and $DARPA_3$. Q_3 is a uniformly distributed workload: The level of `Time` as well as its value (as of any other dimension) are chosen randomly. For all querysets, we set $P_d = 0.3$. Apart from the total resolution time for the 1K query batch and the communication cost, in the T-HORAE case we measure the number of reindexings that occurred as well as the

percentage of queries answered directly, i.e., without flooding (*precision*). The resolution times for Hive are added for direct comparison.

Observing the querying times as well as the communication cost per query of T-HORAE for all workloads in Table 5.8, we come to the conclusion that biased queryloads, regardless the direction of skew, are answered more efficiently. This is due to the reindexing mechanism of T-HORAE, which reorganizes the forests by shifting to the pivot levels that are more beneficial to the system. Indeed, the more biased the workload, the more quickly the system adopts the proper pivot levels, being thus able to answer the majority of the queries (more than 90%) without flooding in almost 2/3 the time compared to the uniform distribution. The number of reindexings as well as the corresponding load remains low due to the efficiency of the reindexing algorithm, which monitors forests independently and only rehashes parts of the data where the popularity definitively indicates a new pivot. Lastly, the dimensionality of the dataset does not affect resolution times, which remain invariably low.

The full version of the H-HORAE is able to answer all queries without any information loss. All point and aggregate queries are answered within d hops, except for queries concerning aggregates of very recent data that have not been created yet as well as queries containing $*$ in `Time`. The latter are substituted by multiple queries that concern finer grained levels, whose aggregates exist. Therefore, Q_1 is the queryset that is resolved in the fastest and more cost efficient way, since its distribution follows the materialization flow. As the queryset approximates the uniform distribution, the messages needed to answer a query increase, naturally affecting the response times. This is attributed to the fact that queries concerning aggregates of recent data cannot be answered directly, since these aggregates have not been constructed yet, but need to be translated to multiple queries of a lower hierarchy level, thus issuing more messages.

The adaptive H-HORAE creates and erases aggregates according to demand. Compared to the full mode, it demonstrates similar performance for Q_1, that queries recent events in finer granularity. We believe that this is the case for the majority of querysets concerning temporal data. However, for the rest of the querysets, resolution takes up an order of magnitude more time and messages. The slowdown is especially apparent in the case of the uniform distribution (Q_3), since it takes longer for the thresholds $TMat_h$ to be reached, thus aggregate subdwarfs for coarse grained levels of `Time` are not created as fast as in Q_2. Moreover, Q_2 also activates the erasure of fine grained views faster, resulting in more limited cube sizes, which allow queries to navigate faster through them. The percentage of the queries that are answered in approximation reaches 28% in the worse case (uniform distribution).

In all cases, both HORAE components clearly outperform the cloud-based Hive solution as far as resolution time is concerned. Although Hive parallelizes the procedure by issuing multiple mappers per node, it resolves the given queryset two orders of magnitude slower than T-HORAE and H-HORAE, needing more than 20 sec to answer a single query. This is due to the fact that

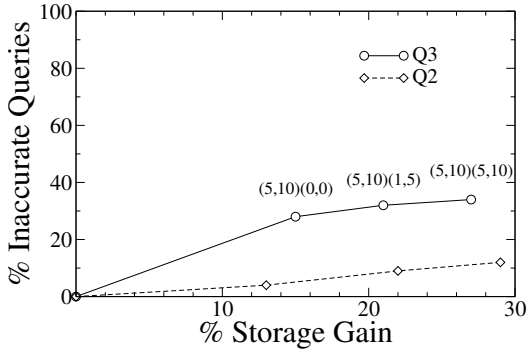


Figure 5.14: *Storage gain vs.% inaccurate queries for various $TDel$ values in H-HORAE*

Table 5.9: *Average Deletion time per T_{off} epoch in T-HORAE*

d	#deletions	time (sec)
5	18685	1.7
10	18906	2.0
15	18755	6.5
20	19044	8.7

Hive tends to have high latency and incur substantial overheads in job submission and scheduling.

There clearly exists a trade-off between the size of the created cube and the accuracy of the query responses. Figure 5.14 depicts the effect of different ($TDel_s$, $TDel_m$) values on these measures for the 10-d dataset under the Q_2 and Q_3 workloads, while ($TMat_m$, $TMat_h$) remain equal to (5, 10). The higher the threshold values, the faster the coarse grained aggregates replace the finer grained ones. This leads to a gain in storage, yet a loss of detailed information, thus an increase in inaccurate answers. However, after a certain combination of the deletion thresholds (which depends on the posed load), the portion of inaccurate answers tends to converge to a steady value. Therefore, to maximize the storage gains at a small cost, one would need to combine workload with application-specific knowledge in order to choose suitable materialization parameters.

5.3.4 Data Offload

We now examine the adaptivity of our offload component under varying update rates and lag values. We set the value of c_{off} so that, assuming a lag of 60 sec, $T_{\text{off}} = 1$ h. We pose continuous updates to T-HORAE, with d ranging from 5 to 20 and $\lambda_{up} = 5\frac{\text{upd}}{\text{sec}}$. In Table 5.9 we observe that the average number of deleted tuples per T_{off} epoch is 20K, regardless of the dimensionality (as it only depends on the lag and update rate). The respective time increases as dimensions grow,

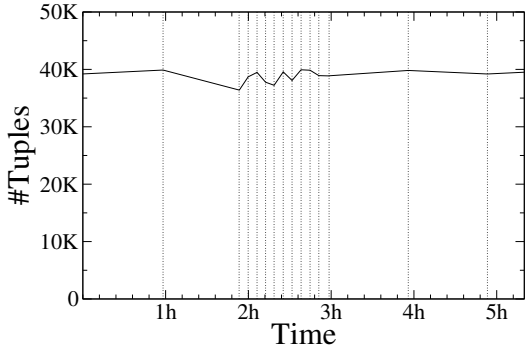


Figure 5.15: *Number of the tuples stored in T-HORAE over time during a pulse-like update rate*

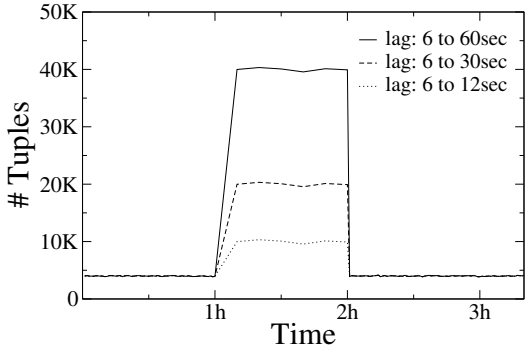


Figure 5.16: *Number of the tuples stored in T-HORAE over time during a pulse-like lag*

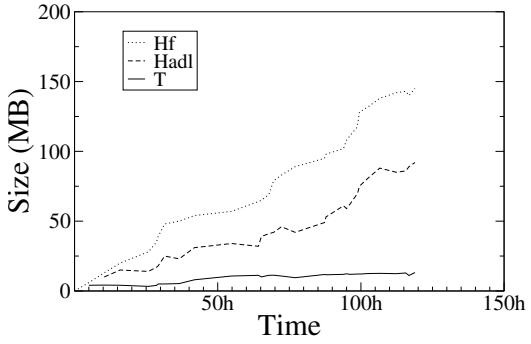


Figure 5.17: *Size distribution over time in the HORAE system for the DARPA dataset*

since they incur larger processing to locate the various tuples hashed at possibly different levels. Yet, the reported times are less than 10 sec, i.e., less than 1% of the estimated T_{off}.

Figure 5.15a plots the number of tuples remaining in T-HORAE in time when a sudden increase in the incoming λ_{up} from about 5 to $50\frac{\text{upd}}{\text{sec}}$ occurs (when $t = 2h$). The load reverts to its initial value when $t = 3h$. As we notice, our system promptly comes up with an accurate estimate for the incoming rate, adapting the rate at which data is offloaded respectively and managing to keep the size of T-HORAE's data fairly stable. On the other hand, when lag varies greatly, T-HORAE delays the collection of tuples before offloading them (to minimize the chance of out-of-order items). This is depicted in Fig. 5.15b, where the lag changes from 6 to 12, 30 and 60 sec per processed batch. Indicatively, T-HORAE chooses to store almost 8 times as many tuples to cope with the 10-fold increase in lag.

5.3.5 Integrated System

In this section, we evaluate the system as a whole, under the APB (A, B and C) and the DARPA datasets. The querysets concerning the APB datasets are produced by the generator and we have

Table 5.10: *Measurements for real and benchmark datasets*

		query distribution(%)		time per query (ms)			avg.time per query (ms)	
dataset	#tuples	T	H	T	H_f	H_{ad_l}	H_f	H_{ad_l}
APB-A	1.2M	10	90	502	14	17	56	56
APB-B	2.5M	9	91	510	17	18	59	60
APB-C	3.7M	9	91	514	20	24	62	64
DARPA	3.0M	62	38	607	52	87	357	374

no control over them. The queryset for the DARPA dataset is created in a way similar to Q_1 (recent events are queried in more detail).

After the insertion of the first 100K tuples in T-HORAE, we start to pose queries with $\lambda_q = 100\frac{\text{queries}}{\text{sec}}$. At the same time, updates keep arriving at the system according to the tuple timestamps. This rate is not constant, but includes bursts in time. Figure 5.17 plots the storage size over time in both T-HORAE and H-HORAE subsystems for the DARPA dataset. Despite the fact that the λ_{up} varies over time, T-HORAE manages to maintain an almost steady size, thanks to the adaptation of T_{off} and W_{rem} to that measure. As for the H-HORAE, the size increases over time as new tuples are periodically transferred from T-HORAE, with the increase being smoother for H_{ad_l}, since aggregates are calculated on demand and unpopular views are erased.

Table 5.10 contains measurements concerning the querying process. First, we note the distribution of queries between the two components. For DARPA, more than 60% of the queries are directed to T-HORAE, since the created queryset was intended to favor recent records. For the APB cases, where we have no control over the generated queryset, the vast majority of the queries are answered by H-HORAE. However, it was observed that almost 99% of the queries were targeted to the most detailed level, therefore resolution times are considerably smaller than those of DARPA. While T-HORAE is in general slower than H-HORAE, this difference is amortized in the integrated system. Moreover, since the two subsystems work simultaneously, parallelization is achieved, further "hiding" the T-HORAE slowdown. Hive remains by orders of magnitude slower, with a per query resolution time around 20 sec.

5.4 Summary

In this chapter we described HORAE, a data-warehouse-like system deployed on a shared-nothing architecture especially designed to handle time series data produced at a high rate. HORAE combines the speed and robustness of a DHT-based layer for efficient update processing with the power of handling aggregate queries of a distributed data cube structure over an unstructured overlay. Its advantages include high-throughput and online updating and querying, elastic

provisioning of commodity resources according to demand and significant gains in storage and update times by on-demand aggregation.

Results from our prototype implementation show that, while HORAE inserts and updates data slower (although at a comparable scale) than Hive, it greatly outperforms it during querying on any kind of aggregate/point queryset combination. Moreover, it allows for highly adaptive resource allocation according to application or workload demands, managing to quickly adapt even after sudden bursts in load; load-driven materialization, varying its gains between storage and precision; availability, remaining unaffected with a considerable fraction of frequent node failures. Altogether, it proves a practical solution that is easy to install and requires no proprietary tools.

CHAPTER 6

Related Work

This thesis proposes three systems for addressing the analytics of modern, network-centric enterprises. The first one focuses on the efficient storage and retrieval of multidimensional, hierarchical information in DHTs, while the second one extends a traditional, well-established data warehousing approach and combines it with P2P techniques in order to create a distributed warehouse-like system. The third system integrates the two former techniques to create an always-on, real-time access and support system for time-series data arriving at large rates. Therefore, the research presented it this thesis spans multiple diverse fields of related work. In the following, mechanisms relevant to those exploited in the dissertation as well as alternative approaches are presented and compared to the proposed ones.

6.1 Sharing of Structured Data in P2P

The sharing of relational data using both structured and unstructured P2P overlays is addressed in a number of papers. PIER [HHL+03] proposes a distributed architecture for relational databases supporting operators such as join and aggregation of stored tuples. A DHT-based overlay is used for query routing. On top of the DHT overlay, a Prefix Hash Tree (PHT) is built for secondary indexing. The PIER platform is also used along with a Gnutella overlay in [LHH+04] for common file-sharing. The unstructured overlay is used for locating popular items while the PIER search engine favors the publishing and discovery of rare items.

The Chatty Web [ACMH03] considers P2P systems that share (semi)structured information but deals with the degradation, in terms of syntax and semantics, of a query propagated along a network path. In [TH04], the authors propose optimization techniques for query reformulation in P2P database systems.

In GrouPeer [KTSR09], SPJ queries are sent over an unstructured overlay in order to discover peers with similar schemas. Peers are gradually clustered according to their schema similarity. PeerDB [NOTZ03] also features relational data sharing without schema knowledge. Query matching and rewriting are based on keywords provided by the users. GridVine [ACMHVP04] and pSearch [TXD03] are based on structured P2P overlays. GridVine hashes and indexes RDF data and schemas, and pSearch represents documents as well as queries as semantic vectors. A work by Vaisman et al. [VEP09] stressing the need for P2P OLAP mainly focuses on answering OLAP queries over a network of data warehouses that do not share the same schema.

An interesting method for representing hierarchical data is presented in the work of Koloniari and Pitoura [KP04]. The method is applied on unstructured networks containing XML documents in order to favor the routing of path queries. Each XML document is represented by an unordered label tree and bloom filters are used to summarize it.

All these approaches offer significant and efficient solutions to the problem of sharing structured and heterogeneous data over P2P networks. Nevertheless, they do not deal with the special case of hierarchies over multidimensional datasets nor with temporal data that arrive at high rates. Moreover, they do not support aggregate queries over voluminous datasets, unlike the systems proposed in this thesis.

6.2 Data Warehousing and Traditional Structures

A *data warehouse* is a central repository that hosts immense volumes of historical data from multiple sources and provides tools for their aggregation and management at different levels of granularity. The basic abstraction in data warehousing is the *data cube*, a multidimensional array in the form of which data are usually organized and viewed. Data cubes are characterized by their *dimensions*, which represent the notions that are important to an organization for managing its data (e.g., time, location, product, customer, etc) and the *facts*, which are the numerical quantities to be analyzed (e.g., sales, profit, etc). Their candidate workloads usually consist of read-only queries interleaved with batch updates. They allow for efficient summarization of data by reducing the dimensions and producing aggregate views of the data. However, data can be presented in an even more fine-grained manner through the use of *concept hierarchies*.

Gray *et al.* introduced the data cube operator in 1997 [GCB$^+$97]. The data cube generalizes many useful operators, namely aggregation, group by, roll-ups, drill-downs, histograms and cross-tabs. A basic problem related to the data cube is the complexity of its computation and the

corresponding storage requirements, since the number of possible views increases exponentially to the number of dimensions. Materialization is commonly used in order to speed-up query processing. This approach fails in a fully dynamic environment where the queries are not known in advance or when the number of possible queries becomes very large. On the contrary, the systems proposed in this dissertation rely on adaptive mechanisms to tackle the index growth while preserving low response times. *HiPPIS* uses a no-precomputation scheme that adjusts the level of data indexing to the incoming load, while *Brown Dwarf* distributes a compressed data cube structure to multiple commodity PCs to accommodate larger cubes. *HORAE* adopts an on-demand materialization mechanism that conforms to the monitored query trend.

Several indexing schemes have been presented for storing data cubes [LPZ03,WLFY02,SDRK02]. However, only few support both aggregate queries and hierarchies. In the work of Sismanis *et al.* [SDKR03], hierarchies are exploited to enable faster computation of the possible views and a more compact representation of the data cube. The Hierarchical Dwarf contains views of the data cube corresponding to a combination of the hierarchy levels.

Another approach is the DC-Tree [EKK00], a fully dynamic index structure for data warehouses modeled as data cubes. It exploits concept hierarchies across the dimensions of a data cube. In this work, the attributes of a dimension are partially ordered with respect to the valid hierarchy schema for each dimension. The DC-tree stores one concept hierarchy per dimension and assigns an ID to every attribute value of a data record that is inserted. CURE [MI06] presents a novel lattice traversal scheme, in order to construct complete data cubes with arbitrary hierarchies. The extension of this work by the same authors [MI10] introduces query-processing and incremental maintenance algorithms for CURE cubes.

These approaches are very efficient in answering both point and aggregate queries over various data granularities, but do so in a strictly centralized and controlled environment. Moreover, they present an off-line approach in terms of data location and processing: Views are usually calculated on a daily or weekly basis after the operational data have been transferred from various locations. In contrast, this thesis presents data-warehouse-like systems that allow for online, near real-time data processing making use of a shared-nothing architecture that ensures scalability and fault tolerance.

6.3 Distributed Data Warehousing Techniques

The notion of a distributed data warehouse has been used in the past, although a more accurate characterization for the proposed systems would be 'cooperative', rather than 'distributed'. In [KNO+02], the authors consider a number of data warehouses and peers forming an unstructured P2P overlay for caching OLAP views. Views are divided in chunks and peers retrieve

cached chunks from the network and the data warehouse if needed. In [ABJ+03], the authors define the distributed data warehouse as a structure that consists of multiple local data warehouses adjacent to data collection points and a coordinator site, responsible for interacting with each of the local sites and for correlating and aggregating query results. A similar approach is described in [CDH99], where a two-layer architecture consisting of multiple local data warehouses and a global one is proposed. All these approaches adopt some hybrid query processing model by allowing requests to route to different cites. WebContent [AAC+08] describes a P2P platform for managing distributed repositories of XML and semantic Web data, where various data processing building blocks are integrated as Web services. Yet, none of the above works distributes the warehouse structure itself, keeping the processing sites and their main functionality centralized.

Recently, effort has been made to exploit parallel processing techniques for data analysis by integrating query constructs from the database community into MapReduce-like software. This new class of analytics engines leverage the recent innovation in the industry around large-scale data management. Deployed on shared-nothing, commodity hardware architectures, they cover the newly added requirement for scalability, robustness and availability at low cost.

The Pig project at Yahoo [ORS+08], the SCOPE project at Microsoft [CJL+08] and the open-source Hive project [TSJ+09] mainly focus on language issues, addressing the creation of SQL interfaces on top of Hadoop [had]. HadoopDB [ABPA+09] proposes a system-level hybrid approach, where MapReduce and parallel DBMSs are combined. SQL queries are translated with the use of Hive into MapReduce jobs, which are eventually executed on a single node, running a DBMS.

Yet, even the new platforms pose some limitations, targeting mostly batch-mode analytics jobs, as they can provide large amount of processing power, rather than real-time, "per-tuple" processing. Our systems maintain high efficiency in both batch and single operations, offering a near real-time analytics platform. Especially *HORAE* proves a solution suitable for applications handling and analyzing time-series data, where updates are produced continuously and responses should be as fresh as possible. Compared to a state-of-the-art warehousing solution such as Hive, HORAE accelerates query resolution by orders of magnitude, manages to quickly adapt even after sudden bursts in load and remains unaffected with a considerable fraction of frequent node failures.

To conclude with relative solutions, parallel database solutions (e.g., [ora, ter]) offer great efficiency at the cost of elasticity and robustness in failures [PPR+09]. Indeed, resources cannot be automatically allocated (nor released) according to demand and the addition of new machines to the system requires significant effort as well as downtime. Lastly, parallel databases do not operate on heterogeneous environments. Contrarily, all systems proposed in this dissertation, based on a shared nothing architecture, guarantee resilience as well as scalability on top of the advantages that a distributed storage offers, without compromising query and update efficiency.

Powerful replication mechanisms (inherent in the *HiPPIS* case and the load-driven one in the *BD* case) ensure data availability despite node failures. Moreover, all operations are designed to redirect messages in case of failed nodes, avoiding the reissuing of queries every time a node fails. Physical or virtual resources can join and leave the system on the fly in a transparent manner.

CHAPTER 7

Conclusions and Future Directions

In the era of data explosion, where almost every transaction is logged, the need to sort through enormous amounts of data, extract useful information and exploit it to detect interesting trends, understand phenomena and behaviors, predict future events and finally make decisions based on solid facts is more than ever compelling. The requirements of today imply the need for an always on, real-time data access and support system for concurrent processing of large query rates without deterioration in response times.

The widely used centralized analytics tools, while highly efficient on complex queries upon large volumes of historical data, fail to meet the constantly increasing needs for storage and computation. Distributed systems and techniques have emerged to fill this gap. Cloud Computing is the latest trend in distributed computing that has drawn the attention of scientists and business experts around the world as a platform that offers seemingly infinite resources on demand. However, the new class of analytics engines, albeit the scalability, robustness and availability it offers, fails to provide real-time, "per-tuple" processing, remaining mainly batch-oriented.

My research focuses on the distribution and manipulation of large volumes of multidimensional data that can be used by analytical processing applications. The thesis initially proposes two systems, *HiPPIS* and *Brown Dwarf*, which both aim to satisfy the same general need: The creation of a data warehouse, deployed on commodity machines, which will be able to provide an always-on, real-time data access and support for online processing. Techniques from the field of P2P computing have been exploited in order to ensure scalability, fault tolerance and fairness in resource utilization.

Both *HiPPIS* and *Brown Dwarf* rely on a shared-nothing architecture of commodity nodes, investing on scalability and availability at low cost. Physical or virtual resources can join the system easily and transparently to relieve stress and assist in coping with increasing demand on storage and/or computing power.

Fault tolerance is another requirement both systems satisfy. Analytics jobs are particularly sensitive to node failures, due to their long completion time. Reissuing the whole query batch in the event of a failed node is not a viable solution, especially when decisions must be made quickly. Fault tolerance and the closely related availability issues are tackled through the inherent DHT replication mechanism in *HiPPIS* and through the adaptive replication scheme of *Brown Dwarf*, perceptive to both tthe incoming load and the node churn using only local load measurements and overlay knowledge.

However, each system deals with the same issue from a different perspective, setting different priorities:

HiPPIS focuses on the management of hierarchical data, allowing queries of various granularities through roll-up and drill-down operations. This fact makes *HiPPIS* suitable for scenarios where a more detailed representation of the data is needed. The simplicity of the *HiPPIS* data structure allows for fast insertion of the initial fact table, without any pre-processing. However, since no a-priori materialization of the cube is performed, group-by queries require further processing after the collection of all tuples that correspond to them. Updates are as fast as insertions, incurring an overhead which depends on the level of consistency needed by each application. Therefore, in situations where data are constantly updated at a high rate, *HiPPIS* can cope in a cost-efficient way.

Brown Dwarf manages to distribute a well-known data structure which materializes a data cube, achieving, in some cases, significant compression rates. At the cost of pre-processing, which is paid only once though, aggregate queries can be answered as easily and naturally as point ones. However, the aggregate functions must be determined beforehand. Updates in *Brown Dwarf* are quite costly, since a single new tuple insertion triggers multiple changes in aggregate values across the structure. Therefore, *Brown Dwarf* is more efficient in environments where the update rate is not very high, compared to the query rate, or where updates can be applied in batches.

There clearly exists a trade-off: Unprocessed (non-materialized) datasets occupy less space and offer easier update functionality at the expense of increased client processing. On one hand, *HiPPIS* offers fast insertion and update of data represented in a fine grained manner through the use of concept hierarchies, but exhibits slower processing of queries. On the other hand, *Brown Dwarf* efficiently answers all point and aggregate queries in a bounded number of steps, but faces more costly updates due to the materialization of the cube.

Having determined the cases where each of the proposed systems is better suited, the *HORAE* system has naturally emerged to bridge the gap and offer an integrated solution that combines the best of the two worlds for the handling of time series data: A powerful indexing/analytics engine for immense volumes of data both over historical and real-time incoming updates with a shared-nothing architecture that ensures scalability and availability at low cost. High-rate updates and queries targeting the most recent items are handled by a DHT-based, HiPPIS-like component that enables fast insertion times and multidimensional indexing. The large bulk of the data is handled through an enhanced *Brown Dwarf* structure, that adaptively materializes and replicates according to demand. The two components seamlessly integrate to offer the advantages of powerful aggregate data processing along with scalability and elasticity of commodity resources.

Ongoing work includes the use of the *HiPPIS* system for the preservation of the anonymity of horizontally partitioned data (see Appendix A). Privacy preservation of distributed data is of great importance, since their analysis along with other related data, often produced by different vendors, may reveal personal details and sensitive information about individuals. Domain generalization is used to remedy such situations: Mapping attribute values to values that belong to a more general domain by climbing up hierarchy levels can help render individuals indistinguishable. *HiPPIS*, which inherently handles hierarchical, distributed data, is enhanced in its indexing mechanism in a way that preserves data anonymity under continuous updates. Further exploring domains and applications where the proposed systems can be successfully used is what I currently pursue.

My work so far has addressed the manipulation of structured data. Relaxing the constraints of the proposed systems on a globally defined schema is part of my future work. Efficiently handling semi-structured data (e.g., XML documents) and supporting dynamic changes in schemata of structured data entail many research challenges.

Moreover, it would be interesting to investigate how MapReduce technologies and the near real-time systems presented in this thesis can complement each other in large scale data analysis. Indeed, as the experimental evaluation of the implemented systems indicates, MapReduce-based analytical engines prove extremely efficient in ETL tasks, but fail both in incremental processing and interactive response times, which often make the qualitative difference in tasks like monitoring, online customer support, debugging etc. On the other hand, all systems proposed in this dissertation offer per-tuple processing, additionally providing operator-level restart in case of individual node failures (unlike the batch-oriented MapReduce model). This observation motivates the need for integration of the two categories, that will allow each system to do what it is best at.

Publications

Journals

- Katerina Doka, Dimitrios Tsoumakos, and Nectarios Koziris. *Brown Dwarf: A Fully-Distributed, Fault-Tolerant Data Warehousing System.* Journal of Parallel and Distributed Computing, under minor revision.

- Katerina Doka, Dimitrios Tsoumakos, and Nectarios Koziris. *Online Querying of d- Dimensional Hierarchies.* Journal of Parallel and Distributed Computing, 71 (2011) pp. 424-437, doi:10.1016/j.jpdc.2010.10.005.

- A. Asiki, K. Doka, I. Konstantinou, A. Zissimos, D. Tsoumakos, N. Koziris, and P. Tsanakas. *A Grid Middleware for Data Management Exploiting Peer-to-Peer Techniques.* Future Generation Computer Systems, 25(4):426–435, 2009.

Conferences

- Katerina Doka, Dimitrios Tsoumakos, and Nectarios Koziris. *Brown Dwarf: a P2P Data-Warehousing System.* In Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM'10), pages 1945–1946, 2010.

- K. Doka, D. Tsoumakos, and N. Koziris. *Distributing the Power of OLAP.* In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10), pages 324–327. ACM, 2010.

- K. Doka, D. Tsoumakos, and N. Koziris. *Efficient Updates for a Shared Nothing Analytics Platform.* In Proceedings of the International Workshop on Massive Data Analytics over the Cloud (MDAC'10), pages 1–6. ACM, 2010.
- K. Doka, D. Tsoumakos, and N. Koziris. *A Fully-Distributed, Fault-Tolerant Data Warehousing System* 9th Hellenic Data Management Symposium (HDMS'10), 2010 (informal proceedings).
- A. Zissimos, K. Doka, A. Chazapis, D. Tsoumakos, and N. Koziris. *Optimizing Data Management in Grid Environments.* In Proceedings of the 11th International Symposium on Distributed Objects, Middleware, and Applications (DOA'09), pages 497–512. Springer, 2009.
- A. Asiki, D. Tsoumakos, A. Doka, and N. Koziris. *Support for Concept Hierarchies in DHTs.* In Proceedings of 8th International Conference on Peer-to-Peer Computing (P2P'08), 2008.
- K. Doka, D. Tsoumakos, and N. Koziris. *HiPPIS: An Online P2P System for Efficient Lookups on d–Dimensional Hierarchies.* In Proceedings of the 10th International Workshop on Web Information and Data Management (WIDM'08). ACM New York, NY, USA, 2008.
- K. Doka, A. Asiki, D. Tsoumakos, and N. Koziris. *Online Querying of Concept Hierarchies in P2P Systems.* In Proceedings of the 16th International Conference on Cooperative Information Systems (CooPIS'08), 2008.
- A. Asiki, K. Doka, I. Konstantinou, A. Zissimos, and N. Koziris. *A Distributed Architecture for Multi-Dimensional Indexing and Data Retrieval in Grid Environments.* In Proceedings of the Cracow Grid Workshop (CGW'07), 2007.
- I. Konstantinou, K. Doka, A. Asiki, A. Zissimos, and N. Koziris. *Gredia Middleware Architecture.* In Proceedings of the Cracow Grid Workshop (CGW'07), 2007.
- A. Zissimos, K. Doka, A. Chazapis, and N. Koziris. *GridTorrent: Optimizing Data Transfers in the Grid with Collaborative Sharing.* In Proceedings of the 11th Panhellenic Conference on Informatics (PCI'07), 2007.

Bibliography

[AAC+08] S. Abiteboul, T. Allard, P. Chatalic, G. Gardarin, A. Ghitescu, F. Goasdoué, I. Manolescu, B. Nguyen, M. Ouazara, A. Somani, N. Travers, G. Vasile, and S. Zoupanos. WebContent: Efficient P2P Warehousing of Web Data. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*, volume 1, pages 1428–1431. VLDB Endowment, 2008.

[ABJ+03] M. Akinde, M. Böhlen, T. Johnson, L. Lakshmanan, and D. Srivastava. Efficient OLAP Query Processing in Distributed Data Warehouses. *Information Systems*, 28(1–2):111–135, 2003.

[ABPA+09] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*, pages 1084–1095. VLDB Endowment, 2009.

[ACMD+03] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a Self-Organizing Structured P2P System. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 29–33, 2003.

[ACMH03] K. Aberer, P. Cudré-Mauroux, and M. Hauswirth. The Chatty Web: Emergent Semantics through Gossiping. In *Proceedings of the 12th International Conference on World Wide Web (WWW'03)*, pages 197–206. ACM, 2003.

[ACMHVP04] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, and T. Van Pelt. Gridvine: Building Internet-Scale Semantic Overlay Networks. *Lecture Notes in Computer Science*, pages 107–121, 2004.

[AFG+09] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS–2009–28, EECS Department, University of California, Berkeley, February 2009.

[ali] The ALICE Project, A Large Ion Collider Experiment. `http://aliceinfo.cern.ch`.

[AMS99] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[apb] *OLAP Council APB-1 OLAP Benchmark*.

[atl] The ATLAS Project. `http://atlas.ch/`.

[BKK+03] H. Balakrishnan, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2):43–48, 2003.

[Bla] J. A. Blackard. The Forest CoverType dataset. `ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype`.

[BvET08] J. Barr, T. von Eicken, and E. Troan. Application Architecture for Cloud Computing. White Paper, 2008.

[BYJK+02] Z. Bar-Yossef, TS Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting Distinct Elements in a Data Stream. *Lecture Notes in Computer Science*, 2483:1–10, 2002.

[cas] The Apache Cassandra Project. `http://cassandra.apache.org/`.

[CCMR06] P. Coles, T. Cox, C. Mackey, and S. Richardson. The Toxic Terabyte: How Data-Dumping Threatens Business Efficiency. *IBM Global Technology Services*, 2006.

[CCR04] M. Castro, M. Costa, and A. Rowstron. Peer-to-Peer Overlays: Structured, Unstructured, or Both? Technical Report MSR-TR-2004-73, Microsoft Research, 2004.

[CDH99] Q. Chen, U. Dayal, and M. Hsu. A Distributed OLAP Infrastructure for e-Commerce. In *Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems*, pages 209–220. IEEE Computer Society, 1999.

[CDK05] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Longman, 2005.

[cha] Chartbeat. `http://www.chartbeat.com/`.

[CJL+08] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*, volume 1, pages 1265–1276. VLDB Endowment, 2008.

[CKR+07] M. Cha, H. Kwak, P. Rodriguez, Y.Y. Ahn, and S. 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*, page 14. ACM, 2007.

[cli] Clicky. `http://www.getclicky.com/`.

[Coh03] B. Cohen. Incentives Build Robustness in BitTorrent. In *Proceedings of the 2003 Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66. Springer, 2001.

[dar98] MIT Lincoln Laboratory, DARPA Intrusion Detection Evaluation. `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/index.html`, 1998.

[dat10] Technology: The Data Deluge. The Economist, `http://www.economist.com/node/15579717`, 2010.

[DATK08] K. Doka, A. Asiki, D. Tsoumakos, and N. Koziris. Online Querying of Concept Hierarchies in P2P Systems. In *Proceedings of the 16th International Conference on Cooperative Information Systems (CooPIS'08)*, 2008.

[DBS08] J. Dittrich, L. Blunschi, and M.A.V. Salles. Dwarfs in the Rearview Mirror: How Big are they Really? In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*, volume 1, pages 1586–1597. VLDB Endowment, 2008.

[DG08] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107, 2008.

[DGR07] A. Deligiannakis, M. Garofalakis, and N. Roussopoulos. Extended Wavelets for Multiple Measures. *ACM Transanctions on Database Systems*, 32(2), 2007.

[dig] Digg. `http://digg.com/`.

[DP10] F. Dabek D. Peng. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, 2010.

[DS10] N.A. Diakopoulos and D.A. Shamma. Characterizing Debate Performance via Aggregated Twitter Sentiment. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems (CHI'10)*, pages 1195–1198. ACM, 2010.

[DTK08] K. Doka, D. Tsoumakos, and N. Koziris. HiPPIS: An Online P2P System for Efficient Lookups on d–Dimensional Hierarchies. In *Proceedings of the 10th International Workshop on Web Information and Data Management (WIDM'08)*. ACM New York, NY, USA, 2008.

[DTK10a] K. Doka, D. Tsoumakos, and N. Koziris. Distributing the Power of OLAP. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*, pages 324–327. ACM, 2010.

[DTK10b] K. Doka, D. Tsoumakos, and N. Koziris. Efficient Updates for a Shared Nothing Analytics Platform. In *Proceedings of the International Workshop on Massive Data Analytics over the Cloud (MDAC '10)*, pages 1–6. ACM, 2010.

[DTK10c] Katerina Doka, Dimitrios Tsoumakos, and Nectarios Koziris. Brown Dwarf: a P2P Data-Warehousing System. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM'10)*, pages 1945–1946, 2010.

[DTK11] K. Doka, D. Tsoumakos, and N. Koziris. Online Querying of d-Dimensional Hierarchies. *Journal of Parallel and Distributed Computing*, 71(3):424 – 437, 2011.

[ec2] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[Eco10] The Economist. Tanks in the Cloud. `http://www.economist.com/node/17797794`, 2010.

[EKK00] M. Ester, J. Kohlhammer, and H.P. Kriegel. The DC-Tree: A Fully Dynamic Index Structure for Data Warehouses. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'00)*, pages 379–388. IEEE Computer Society Press, 2000.

[emu] eMule. `http://www.emule.com/`.

[fac] Facebook. `http://www.facebook.com/`.

[fac11] Facebook Statistics, Stats & Facts For 2011 . `http://www.digitalbuzzblog.com/facebook-statistics-stats-facts-2011/`, 2011.

[FKNT03] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. The Physiology of the Grid. *Grid Computing: Making the Global Infrastructure a Reality*, pages 217–250, 2003.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200, 2001.

[fre] The FreePastry Project. `http://freepastry.rice.edu/FreePastry`.

[gap] Google AppEngine. `http://code.google.com/appengine/`.

[Gar06] M. Garofalakis. Wavelet-Based Approximation Techniques in Database Systems. *IEEE Signal Processing Magazine*, 23:54–58, 2006.

[GCB+97] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[GKMS03] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. One-Pass Wavelet Decompositions of Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15:541–554, 2003.

[gnu03] Gnutella File-sharing and Distribution Network. `http://rfc-gnutella.sourceforge.net/`, 2003.

[gooa] GoodData. `http://www.gooddata.com/`.

[goob] Google Analytics. `http://www.google.com/analytics/`.

[Gro10] Aberdeen Group. Data Management for BI: Strategies for Leveraging the Complexity and Growth of Business Data, 2010.

[GSBK04] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive Replication in Peer-to-Peer Systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, volume 24, pages 360–371, 2004.

[had] Hadoop Web Page. `http://hadoop.apache.org/core/`.

[hba] The Apache HBase Project. `http://hbase.apache.org/`.

[HHL+03] R. Huebsch, J.M. Hellerstein, N. Lanham, B.T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of the 29th International conference on Very Large Data Bases (VLDB'03)*, page 332. VLDB Endowment, 2003.

[HK06] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[IDC10] IDC. The Digital Universe. `http://www.emc.com/collateral/demos/microsites/idc-digital-universe/iview.htm`, 2010.

[JC06] W. Jiang and C. Clifton. A Secure Distributed Framework for Achieving k-Anonymity. *The VLDB Journal*, 15(4):316–333, 2006.

[JMW03] S. Jain, R. Mahajan, and D. Wetherall. A study of the Performance Potential of DHT-based Overlays. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, page 11. USENIX Association, 2003.

[joo] Joost P2PTV. `http://www.joost.com/`.

[kaz] Kazaa. `http://www.kazaa.com/`.

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. Oceanstore: An Architecture for Global-Scale Persistent Storage. *ACM SIGARCH Computer Architecture News*, 28(5):190–201, 2000.

[KK07] D. Kato and T. Kamiya. Evaluating DHT Implementations in Complex Environments by Network Emulator. In *Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS'07)*, 2007.

[KNO+02] P. Kalnis, W.S. Ng, B.C. Ooi, D. Papadias, and K.L. Tan. An Adaptive Peer-to-Peer Network for Distributed Caching of OLAP Results. In *Proceedings of the 2002*

ACM SIGMOD International Conference on Management of Data (SIGMOD'02, pages 25–36. ACM New York, NY, USA, 2002.

[Kno09] Eric Knorr. Dealing with the Data Explosion. Infoworld, `http://www.infoworld.com/d/storage/dealing-data-explosion-690`, 2009.

[KP04] G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT'04)*, pages 525–526. Springer, 2004.

[KTSR09] V. Kantere, D. Tsoumakos, T. Sellis, and N. Roussopoulos. GrouPeer: Dynamic Clustering of P2P Databases. *Information Systems*, 34(1):62–86, 2009.

[LCC+02] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of the 16th International Conference on Supercomputing (SC'02)*, pages 84–95. ACM, 2002.

[LCP+04] E.K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Survey and Tutorial*, 2004.

[LDR05] K. LeFevre, D.J. DeWitt, and R. Ramakrishnan. Incognito: Efficient Full-Domain k-Anonymity. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*, page 60. ACM, 2005.

[LDR06] K. LeFevre, DJ DeWitt, and R. Ramakrishnan. Mondrian Multidimensional k-Anonymity. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 25–25. IEEE Computer Society Press, 2006.

[lhc] The Large Hadron Collider. `http://lhc.web.cern.ch/lhc/`.

[LHH+04] B.T. Loo, J.M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P File-Sharing with an Internet-Scale Query Processor. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, page 443. VLDB Endowment, 2004.

[lig] The LIGO Project, Laser Interferometer Gravitational Wave Observatory. `http://www.ligo.caltech.edu/`.

[lin] LinkedIn. `http://www.linkedin.com/`.

[LPZ03] L.V.S. Lakshmanan, J. Pei, and Y. Zhao. QC-Trees: an Efficient Summary Structure for Semantic OLAP. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, page 75. ACM, 2003.

[LSM+05] J. Li, J. Stribling, R. Morris, M.F. Kaashoek, and T.M. Gil. A Performance vs. Cost Framework for Evaluating DHT Design Tradeoffs Under Churn. In *Proceedings of the 24th IEEE Conference on Computer Communications (INFOCOM'05)*, volume 1, pages 225–236. IEEE, 2005.

[LWFP06] J. Li, R. Wong, A. Fu, and J. Pei. Achieving k-Anonymity by Clustering in Attribute Hierarchical Structures. In *Proceedings of the 8th International Conference on Data Warehousing and Knowledge Discovery (DaWaK'06)*, pages 405–416. Springer, 2006.

[LWFP08] J. Li, R.C.W. Wong, A.W.C. Fu, and J. Pei. Anonymization by Local Recoding in Data with Attribute Hierarchical Taxonomies. *IEEE Transactions on Knowledge and Data Engineering*, pages 1181–1194, 2008.

[MGL+10] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB'10)*, volume 3. VLDB Endowment, 2010.

[MI06] K. Morfonios and Y. Ioannidis. CURE for Cubes: Cubing Using a ROLAP Engine. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06)*, page 390. VLDB Endowment, 2006.

[MI10] K. Morfonios and Y. Ioannidis. Revisiting the Cube Lifecycle in the Presence of Hierarchies. *The VLDB Journal*, 19:257–282, 2010. 10.1007/s00778-009-0160-3.

[mic] Microsoft Azure. `http://www.microsoft.com/Cloud`.

[MM02] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, pages 53–65, 2002.

[MM10] E. Mustafaraj and P. Metaxas. From Obscurity to Prominence in Minutes: Political Speech and Real-Time Search. In *Proceedings of the International Web Science Conference (WebSci '10)*. ACM, 2010.

[Mon] C. Monash. The 1-petabyte Barrier is Crumbling. `http://www.networkworld.com/community/node/31439`.

[NOTZ03] W. S. Ng, B. C. Ooi, K. L. Tan, and A. Zhou. Peerdb: A P2P-Based System for Distributed Data Sharing. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, volume 1063, pages 17–00. IEEE Computer Society Press, 2003.

[ora] Oracle Exadata. `http://www.oracle.com/us/products/database/database-machine/index.html`.

[ORS+08] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. 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. ACM, 2008.

[PHAML98] M.D. Powell, S.H. Houston, L.R. Amat, and N. Morisseau-Leroy. The HRD Real-Time Hurricane Wind Analysis System. *Journal of Wind Engineering and Industrial Aerodynamics*, 77:53–64, 1998.

[PNT06] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, Load Balancing and Efficient Range Query Processing in DHTs. *Lecture Notes in Computer Science*, 3896:131, 2006.

[PPR+09] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. 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)*. ACM, 2009.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware'01)*, pages 329–350. Springer, 2001.

[red] reddit: The Voice of the Internet – News Before it Happens. `http://www.reddit.com`.

[RFI02] M. Ripeanu, I. Foster, and A. 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.

[san] Sandvine Inc., The Impact of File Sharing on Service Provider Networks. An Industry White Paper.

[SDKR03] Y. Sismanis, A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical Dwarfs for the Rollup Cube. In *Proceedings of the 6th ACM International Workshop on Data Warehousing and OLAP (DOLAP'03)*, pages 17–24. ACM, 2003.

[SDRK02] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the Petacube. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 464–475. ACM, 2002.

[Sie08] L. Siegele. Let it Rise. `http://www.economist.com/node/12411882?story_id=12411882`, 2008.

[SJ06] A. Singhal and S. Jajodia. Data Warehousing and Data Mining Techniques for Intrusion Detection Systems. *Distributed and Parallel Databases*, 20:149–166, 2006.

[sky] Skype. `http://www.skype.com/`.

[SMK+01] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM International Conference on Data Communication (SIGCOMM'01)*, pages 149–160, 2001.

[SR06] D. Stutzbach and R. Rejaie. Improving Lookup Performance over a Widely-Deployed DHT. In *Proceedings of the 25th IEEE Conference on Computer Communications (INFOCOM'06)*, volume 6, 2006.

[Swe02] L. Sweeney. k-anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty Fuzziness and Knowledge Based Systems*, 10(5):557–570, 2002.

[ter] Teradata. `http://www.teradata.com/t/products-and-services/database/teradata-13`.

[TH04] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 539–550. ACM, 2004.

[TR03] D. Tsoumakos and N. Roussopoulos. A Comparison of Peer-to-Peer Search Methods. In *Proceedings of the 6th International Workshop on the Web and Databases (WebDB'03)*, 2003.

[TSJ+09] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework.

In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*, volume 2, pages 1626–1629. VLDB Endowment, 2009.

[twia] Twitter. http://twitter.com/.

[twib] Twittercounter. http://twittercounter.com/.

[twi10] Twitter Statistics 2010. http://tweettwins.wordpress.com/2010/06/01/twitter-statistics-2010/, 2010.

[TXD03] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay NSetworks. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 175–186. ACM, 2003.

[VEP09] A. Vaisman, M. Espil, and M. Paradela. "p2p olap: Data model, implementation and case study". *"Information Systems"*, 34(2):231–257, 2009.

[ver] Vertica. http://www.vertica.com/.

[vol] Project Voldemort. http://project-voldemort.com/.

[wea] Kx Systems. http://www.kx.com/.

[weba] WebAbacus. http://www.foviance.com/.

[webb] WebTrends Analytics. http://www.webtrends.com/Products/Analytics/.

[Wei11] Kevin Weil. Rainbird: Realtime Analytics at Twitter. Presentation in Strata 2011, http://assets.en.oreilly.com/1/event/55/Realtime%20Analytics%20at%20Twitter%20Presentation.pdf, 2011.

[WLFY02] W. Wang, H. Lu, J. Feng, and J.X. Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*. IEEE Computer Society Press, 2002.

[woo] Woopra. http://www.woopra.com/.

[XWP+06] J. Xu, W. Wang, J. Pei, X. Wang, B. Shi, and A.W.C. Fu. Utility-Based Anonymization Using Local Recoding. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'06)*, page 790. ACM, 2006.

[Zho09] S. Zhong. On Distributed k-Anonymization. *Fundamenta Informaticae*, 92(4):411–431, 2009.

APPENDIX A

Exploitation of HiPPIS for k-anonymity

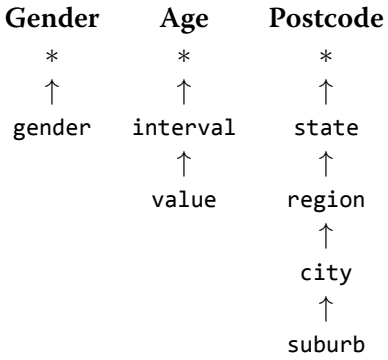
In this Appendix, we present a use case application for the *HiPPIS* system, related to the anonymization of distributed data.

Numerous companies, scientific organizations as well as specialized enterprises produce ever-growing amounts of data and heavily rely on their continuous analysis in order to identify behavioural patterns and discover interesting trends and associations. In order to cope with these needs, distributed data-warehouse-like systems have been created, deployed on a shared-nothing, commodity hardware architecture, giving the advantage of scalability and robustness at low cost.

However, the wide accessibility to vast amounts of data, often originating from many different sources, raises issues of individual privacy protection. Even if some identifiers such as Name or Social Security Number are eliminated from the dataset, the combination of certain existing attributes (called *quasi-identifier attributes* or just *QID*) with external, publicly available data (e.g., voter registration lists) might uniquely identify individuals and release sensitive information about them ([Swe02]).

K-anonymity has been proposed as a remedy for such attacks. Its goal is to ensure that individuals are unidentifiable in released data. In a *k-anonymous* table, each value of the quasi-identifier set appears at least k times. The most common way to produce k identical tuples is to generalize values within the attributes, e.g., by dropping the least significant digit from the Zip code domain. At the same time, the utility of the published data should remain as high as possible.

Figure A.1: *Concept hierarchies for* `Gender`, `Age` *and* `Postcode`



Various approaches for generalization dictate the mapping of a set of attribute values to another set of values that belong to a more general domain. This mapping can be done either globally, by mapping the whole domain to a more general one (*global recoding*) or locally, by mapping each tuple individually to a generalized one (*local recoding*). There are many works in the literature concerning global ([LDR05, LDR06]) and local recoding ([LWFP08, XWP+06]). More recent works use attribute hierarchies in order to achieve k-anonymity with the less possible information loss by "climbing up" in the domain hierarchy ([LWFP08]).

However, these methods refer to the anonymization of one centralized database and do not deal with distributed data. Some works ([JC06, Zho09]) attempting to propose distributed k-anonymity algorithms do not concern data horizontally partitioned and distributed among multiple network nodes. As more and more systems choose a horizontal partitioning of relational data, we believe that their efficient anonymization is of great importance in order to guarantee privacy.

We believe that *HiPPIS* and its algorithms can be applied to the problem of continuously anonymizing fully distributed data in a way that balances the data distortion and the communication and computation overhead without affecting the efficiency and performance of data indexing and querying operations. To that end we propose the proper modifications to *HiPPIS* to provide an always on DHT-based system, that, besides storing and indexing multidimensional, hierarchical data, can guarantee near real-time k-anonymization of the shared data during updates. Although here only k-anonymity is discussed, the proposed system can be extended to support other privacy principles. The method focuses on global recoding, as we believe that for statistical analysis it is more important to maintain the domain consistency in each attribute. Indeed, when values are drawn from more than one domain, values from a more general domain do not provide the same detailed information as values from a more specific domain. The system nodes actively monitor the privacy of the data they are responsible for, in order to adjust

Table A.1: *The raw table of our motivating scenario*

No.	Gender	Age	Postcode	Problem
1	male	middle	4350	Flu
2	male	middle	4350	Ulcer
3	male	middle	4351	Ulcer
4	female	old	4353	Flu
5	female	old	4353	Ulcer

Table A.2: *The 2-anonymized version of Table A.1 through the use of hierarchies*

No.	Gender	Age	Postcode	Problem
1	male	middle	435*	Flu
2	male	middle	435*	Ulcer
3	male	middle	435*	Ulcer
4	female	old	435*	Flu
5	female	old	435*	Ulcer

the indexing level to the one that guarantees k-anonymity after the insertion of new tuples. Furthermore, the system does not invalidate the semantics of the stored hierarchies and allows for distributed knowledge mining. To our knowledge, this is the first attempt towards the support of distributed k-anonymity in DHTs.

A.1 Definitions

The goal of k-anonymization is to make every tuple of a published table identical to at least $k-1$ other tuples with respect to a set of attributes. As a motivating example, let us assume a table of patient's data (Table A.1). Record No. 3 is unique with respect to the attribute set {`Gender, Age, Postcode`}, hence the medical problem of this patient may be revealed if the table is published. To preserve his privacy, we may generalize the `Postcode` attribute values such that each tuple has at least two occurrences. Assuming each domain is analyzed in the hierarchies `gender`$<*$, `value<interval`$<*$ and `suburb<city<region<state`$<*$ respectively, we can achieve 2-anonymity by climbing up one level in the `Postcode` hierarchy (see Table A.2).

Definition 1 (Quasi-Identifier Attribute Set) *A quasi-identifier attribute set (QID) is a minimal set of attributes in a table that can be joined with external information to potentially identify individual records.*

The QID sets are selected by experts based on the specific knowledge of the domain they refer to.

Definition 2 (Equivalence Class) *An equivalence class (EC) of a table with respect to an attribute set is the set of all tuples that contain identical values for the attribute set.*

Definition 3 (Frequency Set) *The frequency set of a table with respect to an attribute set is a mapping from each equivalence class to the total number of tuples (counts) that belong to it.*

Definition 4 (K-Anonymity) *A table satisfies k-anonymity with respect to a quasi-identifier set if its frequency set contains counts greater than or equal to k.*

Example For our motivating example, the QID set is {`Gender, Age, Postcode`}. Tuples 1 and 2 from Table A.1 form an EC with respect to the QID, with frequency count equal to 2. k-anonymity requires that every tuple occurrence for a given QID set has a frequency of at least k. For example, Table A.1 does not satisfy 2-anonymity since the tuple ⟨male, middle, 4351⟩ occurs once.

There exist various metrics to evaluate the quality of a k-anonymous dataset. A general criterion should be the *distortion* of a table. A simple measurement of distortion is the *modification rate*. For a k-anonymous view V of table T, the modification rate is the fraction of cells being modified within the quasi-identifier attribute set. For example, the modification rate from Table A.1 to Table A.2 is 33.3%. However, this criterion does not consider hierarchical structures. For example, the distortion caused by climbing up from `suburb` to `city` in the `Postcode` hierarchy is not as important as the one introduced by the generalization from `gender` to $*$. A more suitable metric is based on the *weighted hierarchical distance (WHD)*, introduced in [LWFP06].

A.2 Necessary Notation

Our data spawn the d-dimensional space. Each dimension i is organized along $L_i + 1$ hierarchy levels: $H_{i0}, H_{i1}, \ldots, H_{iL_i}$, with H_{i0} being the special *ALL (*)* value. We assume that our database comprises of fact table tuples of the form:
$\langle tupleID, D_{11} \ldots D_{1L_1}, \ldots, D_{d1} \ldots D_{dL_d}, fact_1, \ldots, fact_k \rangle$,
where $D_{ij}, 1 \leq i \leq d$ and $1 \leq j \leq L_i$ is the value of the j^{th} level of the i^{th} dimension of this tuple and $fact_i, 0 \leq i \leq k$ are the numerical facts that correspond to it (we assume that the numeric values correspond to the more detailed level of the cube). Our goal is to efficiently insert and index these tuples so that we can answer queries of the form: $q = \langle q_1, q_2, \ldots, q_d \rangle$, where each query element q_i can be a value from a valid hierarchy level of the i^{th} dimension, including the $*$ value (dimensionality reduction): $q_i = D_{ix}, 0 \leq x \leq L_i$.

Definition 5 (Hierarchy Ordering) *A level combination $C = \langle H_{0L_0}, H_{1L_1}, \ldots, H_{dL_{d-1}} \rangle$ lies above (below) a level combination $C' = \langle H'_{0L_0}, H'_{1L_1}, \ldots, H'_{dL_{d-1}} \rangle$, denoted*
$C \prec C'$ ($C \succ C'$) if $H_{iL_i} \leq H'_{iL_i}$ ($H_{iL_i} \geq H'_{iL_i}$), $\forall 0 \leq i \leq d-1$.

Example For the data of our motivating example, $\langle$`gender`, `interval`, `city`$\rangle \prec \langle$`gender`, `value`, `suburb`$\rangle$, while $\langle$`gender`, `interval`, `city`$\rangle \succ \langle *$, `interval`, `state`$\rangle$.

Property 1 *If a table T satisfies k-anonymity with respect to a level combination C, then it satisfies k-anonymity $\forall C'$, where $C' \prec C$.*

Property 2 *If a table T does not satisfy k-anonymity with respect to a level combination C, nor does it satisfy k-anonymity $\forall C'$, where $C' \succ C$.*

Example The data of our motivating example are 2-anonymous when P is $\langle$`gender`, `interval`, `city`$\rangle$. Therefore, the data are also 2-anonymous when $\langle *$, `interval`, `state`$\rangle$ is selected as P. On the contrary, since $\langle$`gender`, `value`, `city`$\rangle$ does not ensure 2-anonymity, nor does $\langle$`gender`, `value`, `suburb`$\rangle$.

A.3 The System

In previous work we presented *HiPPIS*, a DHT-based system to efficiently store, index and update multidimensional, hierarchical data. In short, *HiPPIS* peers initially choose a level of the suggested hierarchy for each dimension and index all tuples according to that default level combination, called *pivot* (P). This means that each tuple receives an ID that that equals the hashed value of the attribute combination corresponding to P. The DHT then assigns each tuple to the node with ID numerically closest to its ID. Inserted tuples are internally stored in a hierarchy-preserving manner (tree-like form). Query misses are followed by soft-state pointer creations so that future queries can be served without re-flooding the network. Peers maintain local statistics which are used in order to decide if a re-indexing (to a different combination of hierarchy levels) is necessary, according to the current query trend. Besides answering point queries at different levels of granularity, HiPPIS can answer group-by queries.

A.3.1 Insertion

Before the data are initially inserted to the system, we assume the fact table undergoes global recoding centrally (e.g., using [LDR05]) and the appropriate P is selected so that the dataset is k-anonymous. The data are parsed tuple by tuple, hashed according to the selected P and inserted to the corresponding network nodes. Inserted tuples are internally stored in a hierarchy-preserving manner: The data of Table A.2 would be stored as seen in Figure A.2. P would be $\langle$`gender`, `interval`, `city`$\rangle$. Since for the sample data there exist 2 distinct value combinations for pivot, two different trees are created and stored in the corresponding overlay nodes after the insertion process is over. In this system, only the values above the pivot level (the yellow area) are visible to the users, in order to ensure k-anonymity.

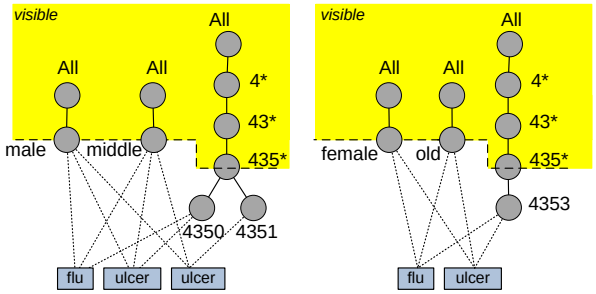


Figure A.2: *HiPPIS system for the motivating example*

A.3.2 Updates

Here we should clarify that by updates we mean the insertion of new tuples, since in data warehousing applications tuples are commonly considered as read-only. Given that P is known to all nodes participating in the overlay, hashing the newly inserted tuple according to it and storing the new item is trivial. However, open issues related to the preservation of k-anonymity with low data distortion arise.

On one hand, new tuples might break the k-anonymity constraint of existing data. Suppose that a tuple $\langle$`female, middle, 4352, Flu`$\rangle$ is inserted in our k-anonymous distributed system. Since the *pivot* is $\langle$`gender, interval, city`$\rangle$, a new tree will be created, as shown in Figure A.3. However, this new tuple is unique for the QID set and thus jeopardizes the privacy of the individual. In this case, a new global P must be selected in order to generalize the data and ensure k-anonymity.

On the other hand, new tuples that arrive and load the existing trees with new values might result in an overgeneralized dataset with high distortion. If tuple $\langle$`male, middle, 4351, Flu`$\rangle$ is inserted, we observe that drilling down one level in the Postcode hierarchy preserves k-anonymity while significantly decreasing data distortion (see Figure A.4).

Both cases require the reindexing of the system's data according to a new P. Our system supports near real-time updates of the distributed data by dynamically adjusting its indexing to the incoming tuples without assuming any prior knowledge, solely relying on locally maintained information. By shifting to a different P we aim at guaranteeing k-anonymous data while causing the least possible distortion.

Every time a new tuple or a batch of new tuples is inserted to the system, the receiving nodes check the modified trees. Note here that each tree corresponds to an EC. If the number of tuples belonging to a tree t (the count of the EC), denoted Count$_t$, is less than k, then the k-anonymity constraint is violated and the *roll-up* anonymization strategy must be followed. If, on the other hand, $Count_t > 2 \cdot k$ then the *drill-down* anonymization strategy further investigates whether a P with less distortion could be chosen.

Roll-up Anonymization During this procedure, the node where the privacy breach has occurred must select an alternative global P in order to ensure k-anonymity. To do so, the node requires information from the rest of the network nodes. To that end, it floods a CollectStats message over the network, which contains the values of all hierarchy levels above *pivot*. Upon reception, each node collates these values with each of its trees and calculates the frequency set of all possible ECs that lie above P. This frequency set is returned to the initiator.

With this process we aim to find all possible ECs that can be merged with the non-anonymized one in order to result in an EC with size of at least k. Since the new P will always be a generalization of the old one, the already anonymized ECs will remain anonymized after the reindexing.

After collecting all the node statistics, the initiator chooses among the possible roll-up level combinations the one, P_{new}, that will result in an EC of k or more tuples and will cause the minimum distortion. A Reindexing is then initiated with P_{new}.

Drill-down Anonymization This procedure is performed in order to check if there exists a level combination that preserves k-anonymity while reducing distortion. It is divided in two phases, the local and the global one. During the local phase, for the specific tree t where $Count_t > 2 \cdot k$ the node calculates the frequency set for all possible level combinations that lie below P. P_{cand} is the set of level combinations that result in ECs with Count$> k$. If the set is empty, the process stops. Otherwise, the global phase begins with P_{cand} being flooded to all network nodes. Upon reception, each node n checks for each level combination of P_{cand} if the resulting ECs are k-anonymous and sends back those that satisfy this constraint ($P_{cand,n}$). After collecting all the answers, the initiator calculates the intersection of the returned sets $\bigcap_{i=1}^{N} P_{cand,n}$ and choses the level combination P_{new} with the minimum distortion. A Reindexing is then initiated with P_{new}.

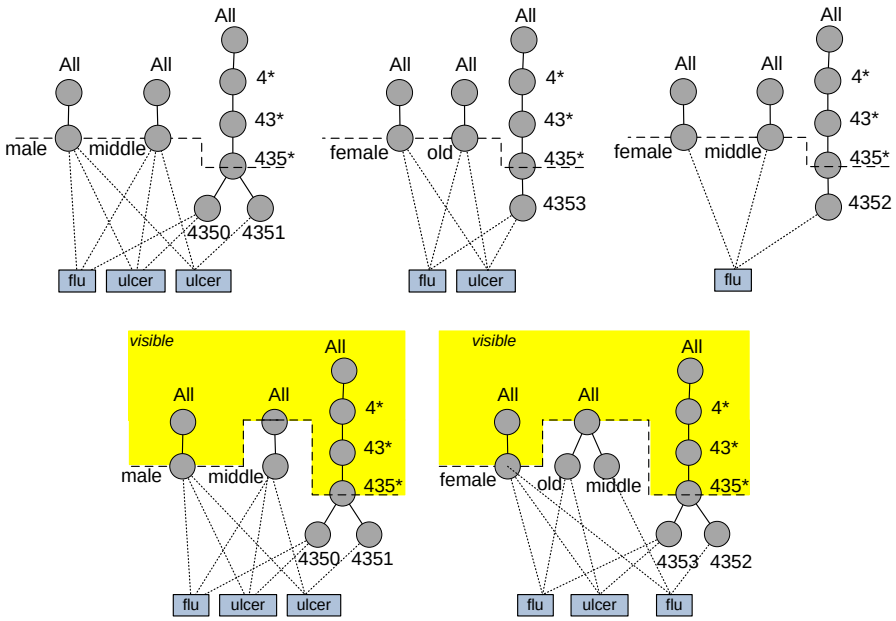


Figure A.3: *Insertion of* ⟨`female, middle, 4352, Flu`⟩ *causes roll-up.*

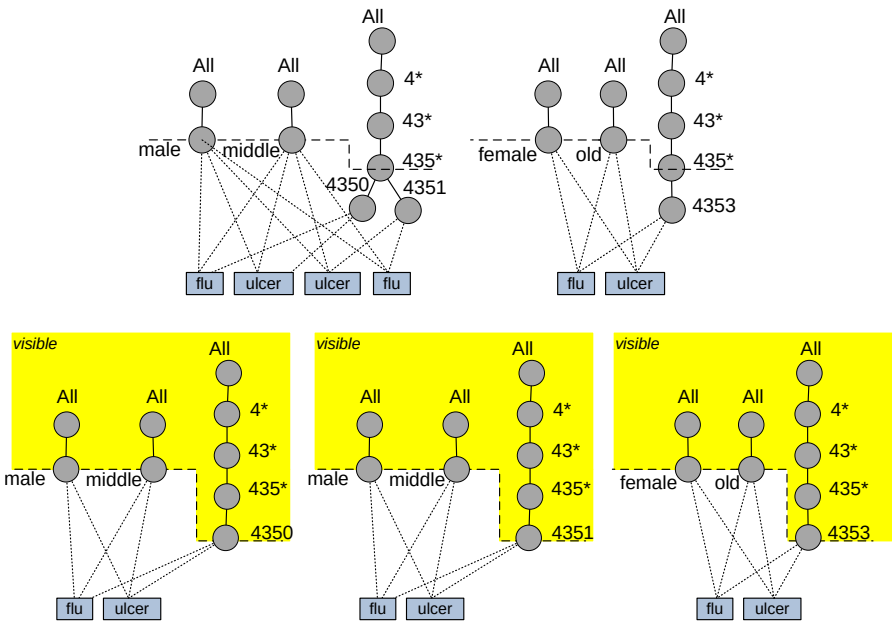


Figure A.4: *Insertion of* ⟨`male, middle, 4351, Flu`⟩ *causes drill-down.*

Index