Τεχνικές Ευρετηριοποίησης και Συσταδοποίησης για την Αποδοτική Αποτίμηση Ερωτημάτων σε Συστήματα Ομότιμων Κόμβων

Η ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από την

Γεωργία Κολωνιάρη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΔΙΔΑΚΤΟΡΙΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

Ιούλιος 2009

Τριμελής Συμβουλευτική Επιτροπή:

- Ευαγγελία Πιτουρά, Αναπληρώτρια Καθηγήτρια του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων (επιβλέπουσα)
- Δημακόπουλος Βασίλειος, Επίκουρος Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Χριστοφίδης Βασίλειος, Αναπληρωτής Καθηγητής του Τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης

Επταμελής Εξεταστική Επιτροπή:

- Ευαγγελία Πιτουρά, Αναπληρώτρια Καθηγήτρια του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων (επιβλέπουσα)
- Δημακόπουλος Βασίλειος, Επίκουρος Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Χριστοφίδης Βασίλειος, Αναπληρωτής Καθηγητής του Τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης
- Παναγιώτης Βασιλειαδης, Επίκουρος Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- Ιωάννης Μανωλόπουλος, Καθηγητής του Τμήματος Πληροφορικής του Αριστοτέλειου Πανεπιστημίου Θεσσαλονίκης
- Τιμολέων Σελλής, Καθηγητής του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πανεπιστημίου
- Παναγιώτης Τριανταφύλλου, Καθηγητής του Τμήματος Μηχανικών Ηλεκτρονικών Υπολογιστών του Πανεπιστημίου Πατρών

Thanks

I would like to express my sincere gratitude and thanks to my advisor Dr. Evaggelia Pitoura, Associate Professor at the Department of Computer Science, University of Ioannina for the valuable guidance and advice she has offered while supervising my dissertation. Her continuous encouragement, patience and insightful suggestions were immensely helpful in overcoming any difficulties that I encountered during my research.

I would also like to thank Assistant Prof. V. Dimakopoulos and Associate Prof. V. Christophides members of my supervising committee for their suggestions and helpful comments. Also, I would like to thank Assistant Prof. P. Vasileiadis, Prof. Y. Manolopoulos, Prof. T. Sellis and Prof. P. Triantafillou for their remarks and useful suggestions. Many thanks also, to all my friends and colleagues in the distributed data management laboratory, for the continuous feedback and support they provided.

Finally, I would like to thank my parents, Michalis and Stella, for their help, love and constant support and encouragement throughout the years of my studies.

Contents

List of Figures

List of Tables

List of Algorithms

Εκτενησ Περιληψη στα Ελληνικα

Γεωργία Κολωνιάρη του Μιχαήλ και της Στέλλας. PhD, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος, 2009. Τίτλος Διατριβής: Τεχνικές Ευρετηριοποίησης και Συσταδοποίησης για την Αποδοτική Αποτίμηση Ερωτημάτων σε Συστήματα Ομότιμων Κόμβων. Επιβλέπουσα: Ευαγγελία Πιτουρά.

Σήμερα, εξαιτίας της ευρείας διάδοσης που παρουσιάζουν εφαρμογές όπως τα συστήματα ομότιμων κόμβων (peer-to-peer systems) και τα κοινωνικά δίκτυα (social networks), σημαντικό ενδιαφέρον επικεντρώνεται στη μελέτη κατανεμημένων συστημάτων μεγάλης κλίμακας, τα οποία αποτελούνται από αυτόνομους δυναμικούς κόμβους που διαμοιράζονται περιεχόμενο. Σε τέτοια συστήματα, βασική πρόκληση αποτελεί η αποδοτική αποτίμηση ερωτημάτων. Στόχος της διατριβής αυτής είναι να παρέχει νέες τεχνικές που βελτιώνουν την αποδοτικότητα της αποτίμησης ερωτημάτων όσον αφορά τόσο το επικοινωνιακό όσο και το επεξεργαστικό κόστος που απαιτείται. Για τον σκοπό αυτό, παρουσιάζουμε λύσεις βασισμένες σε δυο κεντρικούς άξονες: (i) τον ορισμό και τη χρήση κατάλληλων δόμων ευρετηρίου και (ii) την αυτό-οργάνωση των κόμβων του λογικού δικτύου επικάλυψης (logical overlay network) σε συστάδες.

Επικεντρωνόμαστε σε κατανεμημένα συστήματα στα οποία οι κόμβοι διατηρούν και διαμοιράζονται ημιδομημένα δεδομένα, όπως για παράδειγμα XML έγγραφα, που αναπαριστούν την πληροφορία ακλουθώντας μια δενδρική δομή. Εισάγουμε μια νέα δομή ευρετηρίου βασισμένη στον κατακερματισμό, το *πολύ-επίπεδο φίλτρο Bloom*, το οποίο αποτελεί μια επέκταση του γνωστού φίλτρου Bloom σχεδιασμένη έτσι ώστε να διατηρεί τις ιεραρχικές ιδιότητες των ημιδομημένων δεδομένων που συνοψίζει. Εισάγουμε μια δεύτερη δομή που συνδυάζει το πολύ-επίπεδο φίλτρο Bloom με ιστογράμματα, το *πολύ-επίπεδο ιστόγραμμα Bloom*, έτσι ώστε να παρέχει επιπλέον και εκτιμήσεις επιλεκτικότητας. Οι δυο δομές χαρακτηρίζονται από χαμηλό σφάλμα αποτίμησης ενώ έχουν πολύ χαμηλές απαιτήσεις αποθηκευτικού χώρου. Ακόμη, προτείνουμε ένα κατανεμημένο ευρετήριο συστάδων για ημιδομημένα δενδρικά δεδομένα του οποίου η κατασκευή βασίζεται στη χρήση του πολύ-επίπεδου ιστογράμματος Bloom ως κύριου δομικού στοιχείου.

Επειδή τα δεδομένα σε τέτοια κατανεμημένα περιβάλλοντα είναι συνήθως ετερογενή, παρέχουμε τεχνικές για την υποστήριξη προσεγγιστικής επεξεργασίας ερωτημάτων. Συγκεκριμένα, προτείνουμε ένα σύνολο αλγορίθμων για τη δομική χαλάρωση (structural relaxation) XPath ερωτημάτων, οι οποίοι εφαρμόζονται πάνω στο κατανεμημένο ευρετήριο και άρα είναι πολύ πιο αποδοτικοί από αντίστοιχους αλγορίθμους που εφαρμόζονται πάνω στα πραγματικά

δεδομένα. Παρουσιάζουμε τα πλεονεκτήματα της χρήσης του κατανεμημένου ευρετηρίου συστάδων εξετάζοντας δυο σημαντικά σενάρια επεξεργασίας ερωτημάτων: την ανάκτηση των κορυφαίων K (top-K) αποτελεσμάτων και την αυξητική (pay-as-you-go) αποτίμηση ερωτημάτων, συγκρίνοντας την απόδοση που επιτυγχάνουμε με το ευρετήριο συστάδων σε σχέση με ένα τυχαία διαμοιρασμένα κατανεμημένο ευρετήριο. Επιπρόσθετα, θεωρούμε μια παραλλαγή των πολύ-επίπεδων φίλτρων Bloom για την υποστήριξη ερωτημάτων με λέξεισ-κλειδιά στα πλαίσια μιας τεχνικής για την επιλογή κατανεμημένων συλλογών XML εγγράφων, η οποία δε χρειάζεται να προσπελάσει τα δεδομένα αλλά λειτουργεί πάνω στα ευρετήρια που τα συνοψίζουν. Υιοθετούμε σημασιολογία ελάχιστου κοινού απογόνου (LCA-based) και χρησιμοποιούμε το ύψος του ελάχιστου κοινού απόγονου ως ένα μέτρο του πόσο σχετικά είναι τα αποτελέσματα σε σχέση με το ερώτημα. Προτείνουμε έναν προσεγγιστικό τρόπο εκτίμησης του ύψους και σε συνδυασμό με τη χρήση του πολύ-επίπεδου φίλτρου Bloom, παρέχουμε μια κατάταξη των συλλόγων πολύ κοντά σε αυτήν που θα παίρναμε αν επεξεργαζόμασταν αναλυτικά κάθε έγγραφο αλλά με πολύ μικρότερο κόστος.

Αναφορικά με το δεύτερο άξονα, μελετάμε την αυτό-οργάνωση των κόμβων στο λογικό δίκτυο επικάλυψης σε συστάδες με βάση παρόμοιο περιεχομένο και ενδιαφέροντα όπως αυτά εκφράζονται μέσα από τα ερωτήματα που θέτουν οι κόμβοι. Η βασική καινοτομία της προσέγγισης μας είναι ότι μοντελοποιεί τη διαμόρφωση των συστάδων ως ένα στρατηγικό παίγνιο στο οποίο οι κόμβοι (παίκτες) καθορίζουν την στρατηγική τους επιλέγοντας σε ποιες συστάδες θα συμμετάσχουν. Στόχος κάθε παίκτη είναι να επιλέξει τη στρατηγική (συστάδες) που θα βελτιστοποιήσουν μια συνάρτηση ωφέλειας η οποία εξαρτάται από το ποσοστό ανάκλησης των ερωτημάτων (query recall) και το κόστος συμμετοχής σε μια συστάδα. Θεωρούμε παίκτες εγωιστές που προσπαθούν να βελτιώσουν το ποσοστό ανάκλησης των δικών τους ερωτημάτων, καθώς και παίκτες αλτρουιστές που στοχεύουν στη βελτίωση της ανάκλησης των ερωτημάτων των άλλων και ορίζουμε κατάλληλες συναρτήσεις ωφέλειας. Μελετάμε θεωρητικά τις ιδιότητες του παιγνίου και εξετάζουμε συγκεκριμένα σενάρια στα οποία το σύστημα καταλήγει σε κατάσταση ισορροπίας. Βασισμένοι στο μοντέλο μας, προτείνουμε ένα πλήρως κατανεμημένο πρωτόκολλο για το σχηματισμό και την συντήρηση των συστάδων που βασίζεται σε τοπικές αποφάσεις που παίρνονται από κάθε κόμβο ανεξάρτητα για να βελτιώσει τη συνάρτηση ωφέλειας του. Με μια εκτενή πειραματική μελέτη, δείχνουμε ότι μέσα από τις τοπικές αποφάσεις βελτιώνεται η συνολική απόδοση του συστήματος, και το πρωτόκολλο μας επιτυγχάνει βελτίωση στο συνολικό ποσοστό ανάκλησης ερωτημάτων του συστήματος εφάμιλλη με ένα αντίστοιχο συντονισμένο (coordinated) πρωτόκολλο ενώ μειώνει δραστικά το επικοινωνιακό κόστος που θα απαιτούνταν για το συντονισμό και διατηρεί την αυτονομία των κόμβων.

Abstract

Koloniari Georgia. PhD, Computer Science Department, University of Ioannina, Greece. July, 2009. Title of Dissertation: Indexing and Clustering for Efficient Query Evaluation in Peer-to-Peer Systems. Thesis Supervisor: Evaggelia Pitoura.

Nowadays, the popularity of applications such as file-sharing peer-to-peer systems and social networks has drawn significant attention to large-scale distributed systems consisting of dynamic, autonomous nodes (peers) that share their content. An issue of great importance and a considerable challenge in such systems is efficient query evaluation. To this end, the goal of this thesis is to provide novel techniques for improving the efficiency of query evaluation in peer-to-peer systems and large-scale distributed systems, in general. The solutions we propose are centered around two basic axes: (i) the design and deployment of appropriate index structures, and (ii) the self-organization of the nodes in the logical overlay network into clusters.

Since XML has evolved as the de facto standard for data representation and exchange in the Internet, we focus on distributed systems in which the peers maintain semi-structured data, i.e., XML documents that represent data in a hierarchical form. We introduce a novel path-index structure, the *multi-level Bloom filter*, which is based on the well-known Bloom filters that are compact hash-based structures designed to represent a set of elements. Multi-level Bloom filters support path expressions look ups by extending the basic Bloom filters so as to preserve the hierarchical relationships of the data they summarize. We define a second index structure, the *multi-level Bloom histogram*, that combines the multi-level Bloom filter with histograms so as to support selectivity estimations for path expressions. Both structures require low space overhead and result in low evaluation and estimation errors. In addition, we propose a distributed clustered index for semi-structured data which is constructed by using the multi-level Bloom histogram as its basic building block.

Since the data is usually heterogeneous in large-scale distributed systems like the ones we consider, we propose an approach for supporting approximate query processing over the XML data. In particular, we propose a set of structural relaxation algorithms that do not process the actual data but operate on top of the distributed clustered index, thus, significantly reducing the required processing cost. To illustrate the benefits of using the distributed clustered index, we describe two popular query evaluation scenarios, a top-K query evaluation and an incremental pay-as-you-go approach, and compare their performance when using the distributed clustered index, and when using a randomly partitioned

distributed index. We also propose a more efficient organization for the clustered index, which is based on an incremental hierarchical clustering algorithm and the resulting distributed index consists of a set of hierarchies. Through our experimental evaluation, we show that in both query evaluation scenarios the use of the clustered index radically reduces both the communication and processing costs involved as compared to a random index, and the hierarchical organization manages to reduce these costs even further.

We also consider a variation of the multi-level Bloom filters for supporting keyword-queries and use it to address the problem of efficient selection of distributed XML document collections. Based on the multi-level bloom filter, we present an approach that does not require accessing the actual collections of data but instead, it is applied on top of the indexes that summarize it. We adopt lowest common ancestor semantics (LCA-based) for query evaluation, and determine the relevance of a result to the query based on the height of the lowest common ancestor of the keywords that form the result. We introduce an approximate algorithm for estimating the height of the lowest common ancestor and combined with the use of the multi-level Bloom filters for summarizing the required information, we provide a ranking for the XML document collections that is very close to the actual ranking we would get if we had processed each document in the collections separately, but with a significantly reduced processing cost.

Regarding the second axis of solutions we consider, we study the self-organization of nodes in the logical overlay network in clusters based on the similarity of their content and interests as they are expressed through their query workload. The main novelty of our approach is that we model the problem of *cluster evolution* as a strategic *game* in which the peers are the players that determine their strategy by selecting which clusters to join. The goal of the game is for each peer to select the strategy that minimizes its utility function that is defined based on query recall and the cost involved in belonging to clusters. Based on real-life behavior of users in peer-to-peer systems, we model both selfish and altruistic behavior and define appropriate utility functions for both. Selfish peers aim at maximizing the recall of their own queries, while altruistic peers aim at maximizing the recall of the queries of the other peers in their clusters. We study theoretically the properties of the cluster evolution game and consider specific scenarios in which our game reaches a stable state. Furthermore, we introduce a fully-distributed protocol for cluster maintenance that is based on local decisions made by each peer independently so as to improve its utility function. An extensive experimental evaluation shows that our protocol manages through local decisions to improve the overall query recall in the system as a corresponding coordinated protocol would but without requiring any additional communication cost for coordination and while preserving the autonomy of the peers.

Chapter 1

Introduction

Peer-to-peer systems have grown dramatically in recent years. Within a few months of the introduction of Napster [98] in 1999, the system had spread widely. Nowadays, the popularity of file sharing systems (such as Kazaa [62] and Gnutella [51]) and social networks (such as Facebook [38] and Flickr [43]) has resulted in attracting increasing attention and research on large-scale distributed systems that go beyond the traditional client-server model and follow the peer-to-peer computing paradigm.

1.1 Overview and Goal

Peer-to-peer (p2p) computing [93] refers to a form of distributed computing that involves a large number of autonomous computing nodes (the peers) that cooperate to share resources and services. Typical p2p systems are built on top of the Internet or ad-hoc networks. The peers form logical overlay networks on top of the physical network, by establishing links to some other peers they know or discover.

Although the best-known application of p2p systems is file sharing (for example, music files in Napster), p2p system applications go beyond data sharing. Peer-to-peer computing is also a way of implementing systems based on the notion of increased decentralization and self-organization of systems, applications, or simply algorithms. By leveraging vast amounts of computing power, storage, and connectivity from personal computers distributed around the world, p2p systems provide a substrate for a variety of applications such as network monitoring and routing, web search and large-scale event/notification systems.

An increasing number of users chooses peer-to-peer systems as a means to efficiently share their resources, data and services, and exploit the resources that the other participating users provide. Therefore, a central issue in p2p systems is locating the appropriate data among the available, huge, massively distributed data collections. Users in p2p systems issue queries that describe the data they are interested in. The queries are propagated through the overlay network to locate peers that provide relevant data and any matching results are returned to the users that issued the queries. Thus, an important challenge in p2p systems and other large-scale distributed systems that lack centralized control is to support the efficient evaluation of queries both in terms of the communication cost required in locating relevant data and in terms of the processing cost involved in evaluating the queries.

The goal of this thesis is to provide novel techniques for efficient query evaluation in p2p systems and in large-scale distributed systems, in general. Taking into account the distinctive properties of p2p systems that differentiate them from traditional distributed systems that follow the client-server model, such as the autonomy and dynamic nature of the peers, the complete decentralization and lack of central administration, we aim at providing techniques appropriate for such systems. That is, techniques that are distributed and decentralized, do not require global knowledge or control, scalable to support the large volume of available data and users, and efficiently maintainable under dynamic conditions.

To this end, we consider two basic axis on which we focus our solutions: (i) the design and deployment of appropriate index structures, and (ii) the self-organization of peers in the overlay into clusters. In particular, we design index structures appropriate for use in distributed environments that have low storage requirements, reduce the communication costs involved in query evaluation, and also support efficient updates to cope with peer dynamics. Furthermore, we want the peers to form appropriate overlay networks that will improve the performance of query evaluation by enabling efficient location of relevant data.

1.2 Thesis Contributions

Regarding our first goal, the design of appropriate index structures, we focus on systems in which the users share semi-structured data and propose index structures for querying and maintaining such data efficiently. We specifically consider semi-structured data represented in the Extensible Markup Language (XML) [137] that allows dealing with heterogeneous, varied data that may or may not be mapped onto a fixed schema. We consider XML an appropriate choice for use as the underlying data model in p2p systems, since most data shared in p2p systems do not follow predefined schemas and are heterogeneous and XML has become the de facto standard for data representation and exchange in the Internet.

With respect to our second goal, we focus on the self-organization of peers in the

overlay into clusters. We study the formation and evolution of *clustered overlay networks*, in which peers with similar content or interests as they are expressed through their queries, establish logical links between them, thus forming groups (clusters) in the overlay network. Our motivation is based on many real life paradigms which demonstrate the appearance of such groups (clusters) of users that share the same interests and maintain similar content, both in the case of p2p file sharing applications and in social networking applications.

Our main contributions can be summarized as follows:

- We proposed two novel index structures for summarizing XML data that are appropriate for use in distributed systems. Both structures are based on hashing and require low space overhead. The first structure [67], [68], [69], [70] supports boolean queries, while the second [73], [75] provides selectivity estimations for path expressions.

- We addressed the problem of efficient approximate query evaluation over distributed XML document collections. We propose building a distributed clustered index over the documents, and show that we can improve both top-K and pay-as-you-go approximate evaluation [73], [75].

- We use a variation of the index structures to address the problem of database selection over distributed XML document collections [78].

- We provided a new approach on modelling clustered overlay formation in p2p systems based on a game-theoretic approach that fits the autonomous dynamic nature of the peers and the selfish behavior they exhibit in real life applications. In addition, we proposed a completely decentralized protocol for cluster formation and maintenance that although based on local decisions made independently by each peer, it manages to improve system performance and cope with the changing conditions satisfactorily [74], [76].

Next, we describe our main contributions in more detail.

Novel Index Structures for Large Distributed XML Collections

One of our primary contributions is the design of two novel index structures for XML data. We first introduced the multi-level Bloom filter, a hash based index structure that supports boolean queries, that is, the structure queries that check whether a path expression belongs to the document the structure summarizes. We also proposed a second structure, the multi-level Bloom histogram, that combines multi-level Bloom filters with histograms and provides selectivity estimations of path expressions. Both structures require low space overheads with respect to the original data but still achieve a high accuracy in their evaluations. For example, the multi-level Bloom histogram presents a false positive ratio of below 5% while occupying less than 10% of the space of the original data. Another important feature of the structures is that they maintain the structural properties of the

XML documents they summarizes. This feature can be exploited in many applications for example for evaluating structural queries (based on path expressions) over the index or for evaluating the structural similarity of two documents by evaluating the similarity of their indexes.

Approximate Query Evaluation over Distributed XML collections

In accordance with one of our basic assumptions that the data is heterogeneous and its schema is unknown, we provided techniques for supporting approximate query evaluation. In particular, we proposed a set of structural relaxation algorithms for XPath queries that operate on the summarized data maintained in the multi-level Bloom histograms and thus are much more efficient than relaxation techniques that require accessing the actual data and also appropriate for use in a distributed setting.

Another significant contribution of this thesis is illustrating the importance of clustering in improving the cost of query evaluation in distributed systems. This is achieved through a novel distributed clustered index that uses the multi-level Bloom Histogram structure as its building block. The proposed distributed clustered index is constructed by assigning documents to clusters based on their structural similarity which is derived by the similarity of their corresponding summaries. Due to the use of the multi-level Bloom histogram as its building block, the distributed clustered index requires low construction cost and is incrementally updatable. We showed how the clustered index can improve system performance in distributed environments by addressing two important query evaluation scenarios, namely top-K and pay-as-you-go query evaluation. For top-K query evaluation, combining our relaxation algorithms and the use of the clustered distributed index, we showed how by deploying a threshold-based top-K algorithm with clustering improves performance up to 250% (in terms of processing and communication cost saved), compared to using a randomly partitioned distributed index. Similarly, we showed how the recall of a pay-as-you-go approach increases radically when a clustered index is deployed compared to a random index where the recall increases linearly to the percentage of the data evaluated.

Database Selection for Distributed XML Collections

We addressed the problem of database selection over distributed collections of XML documents. In this case, we consider keyword queries and adopt lowest common ancestor semantics in evaluating them to maintain the expressiveness of the structural properties of the XML documents. Our main contribution in this problem is that we proposed a technique for database selection that does not need to access the actual document collections but operates on appropriate index structures.

We use the height of the lowest common ancestor as a measure of how specific a result is with respect to the query and use an approximation algorithm to estimate the height of the lowest common ancestor of a set of keywords by the largest height value of the lowest common ancestor of any pair of keywords in the set. By maintaining this information

for each pair of keywords in an XML document in a preprocessing phase, we reduce the space overhead required from $O(2^n)$ to $O(n^2)$ and provide an approximation with small estimation error (20% in the worst case for complex tree structures).

We further improved this approach (both processing and space overhead) by using a hash-based index structure similar to the multi-level Bloom filter to maintain the lowest common ancestor information about the keyword pairs. By deploying this approach to estimate the relevance of an XML document to a query and then aggregating to estimate the goodness of a collection, our solution provides a ranking of the databases very close to the one we would get by evaluating the query against each document in the collections.

Cluster Reformulation as a Game

With regards to the cluster formation and evolution in overlay networks, we modelled cluster formation as a strategic game where the players are the peers that determine their strategy by choosing which clusters to join. The goal of each player is to choose the strategy (i.e., the clusters) which optimizes a utility function that depends on query recall and the cost of belonging to a cluster. Much like in real life, we discern between selfish peers that aim at improving the query recall of their own queries and altruistic peers that try to improve the recall of the queries of the other peers in the clusters they belong to. We defined corresponding utility functions, namely, an individual peer cost function that selfish peers try to minimize and an individual peer contribution function that altruistic peers try to maximize. Both functions depend on the query workload of the peers as well as their content and thus provide a more complete approach to the clustering problem than previous approaches that mostly relied on content similarity and ignored the query workload.

Based on our modelling, we proposed a completely decentralized protocol for cluster formation that relies on local decisions made by each peer to improve either its cost or contribution. As shown through an extensive experimental analysis, our protocol performs comparably to a corresponding coordinated protocol while reducing the communication costs that coordination would require and enabling the peers to retain their autonomy.

Furthermore, we consider the problem of clustering maintenance. Since we are dealing with dynamic peers that may frequently update their content, change their workload or leave and join the system, a static clustered overlay is not suitable, i.e., the peers recall is decreased and so is the overall system performance. Thus, we use our protocol for coping with changes in the system conditions by allowing the peers to re-evaluate their strategies so as to detect changes in their cost or their contribution and re-evaluate their strategy to reflect the current system conditions more appropriately. We define both an event-based and a trigger-based protocol to detect changes and adjust the clustered overlay to changing conditions. Both protocols work almost as well as applying re-clustering from scratch although with significantly lower communication and processing cost.

1.3 Thesis Layout

The rest of this thesis is structured as follows.

In Chapter 2, we briefly describe the basic characteristics of p2p systems and the XML data model we assume and then present related research regarding XML data management issues in p2p systems. We focus on current approaches especially concerning our solutions axes, that is, the index structures deployed and the logical overlay organizations followed.

In Chapter 3, we introduce the hash-based index structures for XML data. We first present the multi-level Bloom filter for supporting boolean queries and then, the multi-level Bloom histogram that provides selectivity estimations.

Chapter 4 presents the distributed clustered index we construct based on the multi-level Bloom histogram and two popular query evaluation scenarios that demonstrate its benefits, namely a top-K query evaluation procedure and an incremental, pay-as-you-go approach.

Chapter 5 describes another deployment scenario for our multi-level Bloom filter, in which we define a variation of the initial structure and deploy it to improve the performance of the problem of selection of distributed collection of XML documents.

In Chapter 6, we move to our second solution and introduce the strategic game model we use for cluster formation and evolution along with the corresponding cluster reformulation protocols.

Chapter 7 summarizes this thesis and offers directions for future work.

Chapter 2

Related Work on XML Processing in Peer-to-Peer Systems

In this chapter, we present research related to the management of XML data in peer-to-peer systems. We first provide an overview of the characteristics of p2p systems (Section 2.1) and briefly describe the XML data model (Section 2.2). Next, we present how current approaches deal with some important issues of data management such as indexing (Section 2.3), clustering (Section 2.4) and query evaluation (Section 2.5). This chapter offers a general overview of the XML management issues with emphasis on the research topics covered by this thesis, such as indexing and clustering. Summing up our conclusions, we provide a comparative summary of the basic ideas behind the solutions offered by current research (Section 2.6).

2.1 The Peer-to-Peer Computing Paradigm

Peer-to-peer (p2p) computing [93] refers to a form of distributed computing that involves a large number of autonomous computing nodes (the peers) that cooperate to share resources and services. Typical p2p systems reside on the edge of the Internet or in

ad-hoc networks. The peers form logical overlay networks on top of the physical one, by establishing links to some other peers they know or discover.

A set of characteristics found in most peer-to-peer systems differentiate them from traditional distributed systems that follow the client-server model and make the application of previous techniques deployed in those systems problematic.

Some distinctive characteristics of a p2p system are:
Scalability: While even in large traditional distributed systems, the number of participating nodes is in the order of hundreds; p2p systems must achieve scalability at the Internet-level.

Decentralization: Peer-to-peer computing is an alternative to the centralized and client-server models of computing, where there is typically a single or small cluster of servers and many clients. In its purest form, the peer-to-peer model has no concept of server; rather all participants are equal, with each node given both server and client capabilities. Between the centralized and the pure peer-to-peer approach, there are hybrid p2p systems, in which some of the participants, called superpeers, have extended responsibilities and control over the others. The superpeers are often peers that have increased capabilities (storage and processing) and good stability properties.

Autonomy: We distinguish four kinds of autonomy, (i) storage, (ii) execution, (iii) lifetime and (iv) connection autonomy. *Storage autonomy* refers to the freedom of what a node in the system stores. In traditional distributed systems with central administration, the system enforces to the nodes which data items or indexes to store. P2p systems are self-configured: each peer stores its own data according to its interests and needs. Storage autonomy has another dimension related to *ownership* of data. This kind of autonomy allows a peer to determine which other peers in the system can store its own data or index information about its data.

Execution autonomy refers to the ability of a node to answer queries and change its own data. *Lifetime autonomy* refers to the freedom of each node to join and leave the system arbitrarily. In contrast to traditional distributed systems, p2p systems support this kind of dynamism and many issues concerning fault-tolerance, self-maintenance, ad-hoc connectivity and data availability arise from this requirement. Since the peers leave the system very frequently, a p2p system has to provide mechanisms to cope with these disconnections without causing significant problems in the system operation. Furthermore, self-maintenance techniques should be used to deal with the frequent changes in the network. Finally, *connection autonomy* refers to the form of the overlay network in a p2p system. It enables a peer to select with how many and which peers it will connect to, based for example on trust or friendship with other system users.

Because of the various form of autonomy, many peers may exhibit selfish behavior. Selfish peers may refuse to evaluate or propagate queries or store index information or data copies. Such selfish peers try to exploit the resources and services provided by the system, without being willing to offer anything back to the peer community. To prevent this kind of behavior, p2p systems must provide incentives to peers for sharing their data

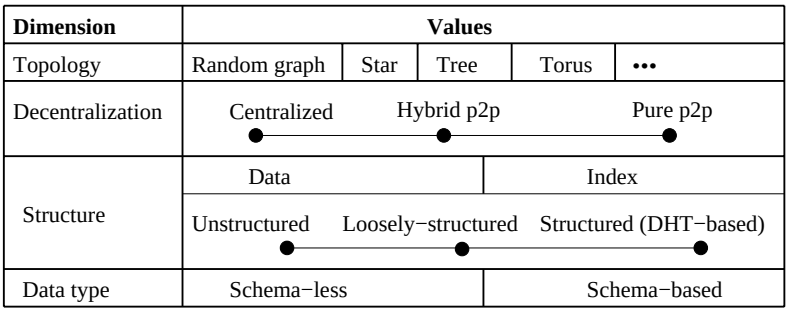


Figure 2.1: Classification of p2p systems

and participating in query processing.

Other challenges that p2p systems need to cope with include anonymity, security and administration transparency. Although, these characteristics may appear in traditional distributed systems as well the way p2p systems have to deal with them is completely different. However, we focus on the data management issues in p2p systems that are not influenced by these characteristics, so we only provide a brief description of each of them. *Anonymity* aims at allowing people to use systems without revealing their identities. In the client-server model, the server identifies the client, at least by its Internet address, whereas in the p2p model where actions are performed locally, users can avoid having to provide any information about themselves to anyone else. Although *security* requirements in p2p systems contradict both anonymity and peer autonomy. One way to minimize threats in p2p systems is to use community-based reputations to help estimating the trustworthiness of peers [138]. Finally, while in traditional distributed systems there are experts that handle the administrative tasks, in p2p systems each user has to administer its own software and devices.

A nice introduction to p2p systems and their basic goals and characteristics can be found in [93]. Some research challenges with emphasis on search are presented in [112], while search and security issues are discussed in [34]. Some initial research ideas on data management in p2p systems are described in [55]. Finally, a comparison of p2p systems with more established decentralized systems models, such as distributed, federated and multi-databases is found in [17].

Considering their distinctive characteristics, we classify peer-to-peer data management systems based on (i) the degree of decentralization, (ii) the topology of the overlay network, (iii) the way information is distributed among the nodes and (iv) the type of data they store. Regarding the degree of decentralization, this varies from pure p2p systems, where all peers have equal roles, to hybrid architectures, where specific peers (the superpeers) are assigned different roles. Topology refers to the way the nodes in the p2p system are interconnected. Example topologies include the star, ring and the grid topology. In hybrid architectures, the topology of the superpeers may differ from that of the other peers. For instance, the superpeers may be fully-connected with each other, while each simple peer is only connected with a single superpeer.

With regards to the way information is distributed among the peers, we distinguish between structured and unstructured p2p systems. In *unstructured* p2p systems, there is no assumption about the way data are distributed among the peers. Unstructured p2p systems can be further distinguished between systems that use indexes and those that are based on flooding and its variations. In the case of indexes, these can be either centralized (as in Napster [98]), which quickly become a bottleneck, or distributed among the peers (as in routing indexes [31]) providing for each peer a partial view of the system. Gnutella-like approaches that use flooding algorithms for query routing do not compromise storage autonomy, while routing indexes based approaches force peers to maintain routing indexes for a number of neighboring peers. Topologies in unstructured p2p systems are usually not restricted to some regular structure, however, they may be regulated, for instance, by setting limits on the number of neighbors each peer can have.

In *structured* p2p systems, data items (or indexes of data items) are placed at specific nodes. Usually the distribution of data items at the peers is based on distributed hashing (DHTs) (such as in CAN [110] and Chord [121]). With distributed hashing, each item is associated with a key and each peer is assigned a range of keys and thus items. We make an additional distinction regarding structured p2p systems based on whether they assign to peers actual data items or indexes of items. Most DHT-based structured p2p systems follow a strict topology (such as a ring or torus) in which each peer has a specific number of neighbors. For example, in Chord, a hash function creates an m-bit identifier space. Identifiers are ordered on an identifier circle modulo 2^m, that forms the Chord virtual ring. As new peers join the system their identifier is produced by hashing their IP address and port. The identifier is used for mapping the new peers in the virtual ring. Data keys are also hashed and distributed to the peers according to their hash value, such as each key is assigned to its successor peer, which is the peer with the nearest hash-value traveling the ring clockwise. When a node n joins the network, certain keys previously assigned to n's successor now become assigned to n. When node n leaves the network, all of its assigned keys are reassigned to n's successor. Each peer maintains a finger table in which it stores its successors. Query routing proceeds by consulting these finger tables to locate the peer with the identifier closer to the search key. Other DHT-based systems exploit a less strict topology. For instance, P-Grid [1] builds a virtual distributed search tree which may be unbalanced.

DHT-based p2p systems support efficient key lookup (e.g., of order O(log$N$) in Chord, where N is the number of peers). However, in most structured p2p systems, both storage and connection autonomy is compromised. The peers are forced by the system to store information for data items assigned to them through distributed hashing and also to follow a regulated topology. In addition, structured p2p systems require sophisticated load balancing procedures to cope with the increased demand for popular items and the system dynamic behavior. There are also systems that are not based on distributed hashing (non-DHT), but do however impose some structure. Such systems organize the overlay network into groups of peers with similar properties. We call such systems *loosely-*

structured or *clustered* p2p systems.

Note that structured, unstructured and loosely structured p2p systems can use either pure or hybrid p2p architectures. For example in the case of structured DHT-based p2p systems, the DHT may be built only upon the superpeers, which have to follow the strict topology imposed by the system, while simple peers connect to one or more superpeers without following any particular structure. Each superpeer is responsible for the routing of the queries of the peers connected to it, and also for forwarding to them the queries that are relevant to their own data.

Finally, we distinguish between schema-less and schema-based p2p systems. In schema-based p2p systems, the peers use explicit schemas to describe their content. Schemas can be heterogeneous. A potential candidate for describing resources in p2p is the Resource Description Framework (RDF). RDF [136] is used to annotate resources on the Web, thus providing the means by which computer systems can exchange and comprehend data. RDF schemas are flexible and can evolve over time by allowing the easy extension of schemas with additional properties, thus being a suitable choice for the representation of resources in dynamic p2p systems. RDF usually uses an XML-based syntax to represent the metadata of the described resources. Apart from representing RDF descriptions in XML, RDF schemas can be used to provide semantic meaning for XML documents by using ontologies that are also described in RDF [65].

Figure 2.1 summarizes our taxonomy.

2.2 XML Data Model and Query Languages

XML [137] has evolved as the standard for the representation and exchange of semi-structured data in the Internet. Several application domains for XML already show that XML is inherently distributed in the Web, for example, Web services that use XML-based descriptions in WSDL and exchange XML messages with SOAP, e-commerce and e-business, collaborative authoring of large electronic documents and management of large-scale network directories. All these applications demonstrate that much of the traffic and data available in the Internet are already represented in XML format. Thus, it is natural to assume that much of the data a user in a p2p system shares is already represented in XML format.

An XML document comprises a hierarchically nested structure of elements that can contain other elements, character data and attributes. Thus, XML allows the encoding of arbitrary structures of hierarchical named values. In our data model, an XML document is represented by a tree. Figure 2.2 depicts an XML document describing a printer and a camera provided by a peer and the corresponding XML tree.

Definition 2.1. (XML tree): An XML tree is an unordered node-labeled tree $Tree(V, E)$ that represents an XML document. Each node $e_i \in V$ corresponds to an XML element with a label assigned from some string literals alphabet that captures element semantics.

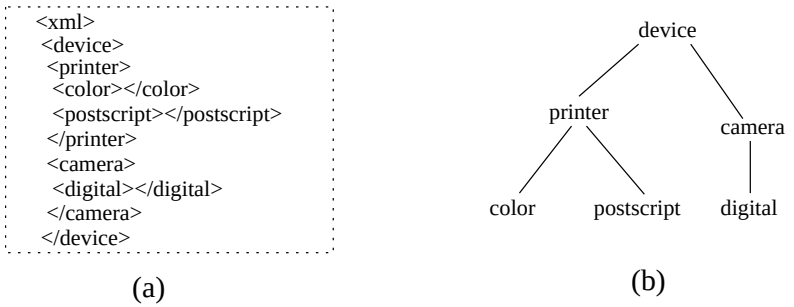


Figure 2.2: Example of (a) an XML document and (b) the corresponding tree

Edges $(e_i, e_j) \in E$ capture the containment of element e_j under e_i.

Any subtree of the XML tree is called an *XML fragment*.

The deployment of XML as the underlying data model for p2p systems can provide a solution to two important issues in current p2p systems: the limited expressiveness of the available query languages and the heterogeneity of data.

In most file-sharing p2p systems, users specify the data they are interested in through simple keyword-based queries. The system matches these keywords against the names of the shared files and returns to the user a list with any matching results. Often, most results returned are not relevant to what the user is interested in. Thus, new more expressive languages than simple keywords are needed to describe and query the shared data. XML seems to be a promising candidate in this direction, since it enables more precise searching because it provides structural and self-describing metadata that allow for context-based and category-based searching.

With regards to heterogeneity, in p2p systems different users and applications use different formats and schemas to describe their data. A user is usually unaware of the schemas remote peers use, so there is a need for a query language that can work with incomplete or no-schema knowledge but also capture whatever semantic knowledge is available. The flexibility of XML in representing heterogeneous data that follow different schemas makes it suitable for distributed applications where the data are either native XML documents or XML descriptions of data or services that are represented in different formats in the underlying sources (i.e. in relational databases). Furthermore, some application domains such as organizations or health-related applications that use sensitive data that must not be exposed to all the system users for privacy reasons allow only partial knowledge of other users' data. A complete schema knowledge may be impossible to achieve even in a centralized scenario [82], thus leading to the need of schema-free XML querying.

Although using XML as the underlying format for describing and querying data in a p2p system addresses some important functional restrictions of current systems, it also imposes some new challenges that p2p systems using keyword-based retrieval do not have to deal with. Since XML query languages exploit both the content and the structure of documents, new indexing and query processing techniques are required. Traditionally,

research work in the area of effective and efficient querying of XML data has been following one of the two paths: the structured query approach (XQuery [23]) and the keyword-based approach (XKeyword [60], XSEarch [28]). While structured queries work effectively with the inherent structure of the XML data and can convey complex semantic meaning, they require from the user to know the schema of the XML data to write the right query. The problem becomes even more difficult when the user has to deal with data from different schemas where the query requires rewriting before it can be evaluated against different schemas. On the other hand, keyword-based searches do not require any knowledge about the underlying data schema but their main drawback is that they do not allow the users to convey semantic knowledge in their queries.

2.3 Distributed Indexes for XML Data

In a p2p system, queries are initiated at various peers. These queries may require data that are located at a large number of peers distributed over the system. Traditionally, distributed systems use centralized or distributed indexes (catalogs) to store information about the location of data. For query processing, the indexes are consulted and the queries are sent to the appropriate nodes and evaluated there. Maintaining indexes in p2p systems poses additional requirements. In particular, indexes in p2p systems must support frequent updates, as p2p systems are very dynamic with peers joining and leaving the system constantly. Furthermore, the indexes need to be scalable since the number of peers reaches Internet-scale, while in traditional distributed systems, the number of participating nodes is much smaller and controlled.

2.3.1 Types of P2P Indexes

There are three basic approaches, maintaining (i) no index, (ii) a centralized index and (iii) a distributed index. When there is *no index* (such as in Gnutella [51]), some form of flooding is used for routing: the peer where the query is initiated contacts its neighbors in the overlay network, which in turn contact their own neighbors until the requested items are located or some system-defined bound is reached. Flooding incurs large network overheads. In the case of a *centralized index* (such as in Napster [98]), information about the contents of all peers in the system is maintained at a single peer. Peers that enter the system publish information about their data in this central index and the central index is consulted when a query is submitted. The drawback of this approach is that the central index server becomes a bottleneck and a single point of failure. Maintaining replicas of the centralized index may increase reliability and scalability but still fails to handle efficiently the huge number of updates.

The distribution of the index depends on the overlay topology and on whether the system is structured or unstructured. In *structured p2p systems*, where distributed hashing is used, each peer stores index information for the data assigned to it by the hash function.

For the evaluation of a query, its hash value is computed and the query is routed through the peer overlay network towards the peer that is responsible for storing the corresponding value. In *unstructured p2p systems*, a popular form of distributed indexing is routing indexes [31]. The *routing indexes* of a peer summarize information about the contents of other reachable peers; they are used during routing to direct the queries towards the peers that are expected to hold relevant data.

In unstructured p2p systems, routing indexes can follow different distribution strategies. Since knowledge about all the peers in the network is infeasible for scalability reasons, *horizons* are used to limit the number of other peers for which each peer stores information. Each index of a peer summarizes information about the data of all the peers that can be reached at a maximum distance Rd, where Rd is called the *radius* of the horizon. A peer may have just one such index or one for each of its links, summarizing the information of all peers on the path starting from this link and at a maximum distance Rd. Another distribution strategy is based on a hierarchical topology. In this case, the peers form one or multiple hierarchies. Each peer in the hierarchy stores information about the peers in its subtree, and the roots of the hierarchies are interconnected. The peers in the upper layers of the hierarchy assume most of the load and use their indexes to forward the queries to parts of the hierarchy where relevant data may be found. By splitting or merging the hierarchies, load-balancing can be achieved.

Distributed indexes can also vary according to the degree of decentralization of the p2p system. In hybrid p2p systems, each superpeer is responsible for a number of other peers for which it stores index information. The superpeers are interconnected with each other to form the network backbone, which can follow either a structured or an unstructured architecture. Each query is forwarded to a superpeer who is responsible for propagating it to the relevant peers or superpeers using its indexes. Query routing follows different protocols among the superpeers and the peers of the system and different types of indexes are used between superpeers and between superpeers and peers.

When using XML as the underlying data format, additional requirements arise for p2p indexes. In particular, for XML documents, we need both value and *path indexes* for addressing the content as well as the structure of documents. We can discern three types of path indexes used in centralized applications. Firstly, there are indexes that assume an unordered tree structured data model. They consist of a tree structure that summarizes path information (path tree indexes). Evaluating a query consists of traversing the path tree and matching the path expression against the tree nodes [52], [30], [25], [11]. Another indexing technique relies on region algebra [2]. Here, the data tree is assumed to be ordered. Regions are defined as contiguous segments of text in an XML document. Region sets are sets of regions, such that any two regions are either disjoint, or one is included into the other. The region algebra is the algebra of the region sets of a document. Different approaches define various operators to manipulate the region sets of a document to efficiently evaluate path expression queries. Finally, the last category assumes that the data follow an arbitrary graph model. These indexes construct reduced

n1	d1: device/printer/postscript, d2: device/camera
n2	d3: device/printer/postscript, device/local/printer
n3	d4: device/camera
n4	d5: device/local/printer, d6: device/network/printer
n5	d7: printer/laser/color, device/camera
n6	d8: network/printer/laser
n7	d9: book/databases/greek
n8	d10: book/databases/english

Figure 2.3: Example of XML data distributed at eight peers

graphs [104, 105, 106] that summarize all paths in the original data graph, by collapsing nodes that are equivalent (two nodes are equivalent if the paths from the root to them are the same). The evaluation of queries proceeds as in path tree indexes. Path indexes and reduced graph-based synopses are also used besides the exact query evaluation also to provide selectivity estimations for path expressions. In [6], path trees and Markov tables are used to support selectivity estimations for simple path expressions. This approach still requires traversing a tree structure for query evaluation, albeit smaller than the original data. [135] introduces Bloom histograms, which are constructed by first splitting the paths in an XML tree according to their frequency and building a histogram for them, and then summarizing the paths that fall in each bucket with a Bloom filter [15] that is a compact hash-based structure with low controllable evaluation error. Finally, [44] defines a tool for extracting statistics from XML schema to construct compact and accurate structural summaries. The approach is based on schema knowledge and cannot be deployed for data that follow heterogeneous unknown schemas.

We shall use the simple example depicted in Fig. 2.3 of eight peers and their documents, where for example, peer n_1 owns two documents d_1 and d_2 that contain paths "device/printer/postscript" and "device/camera" respectively.

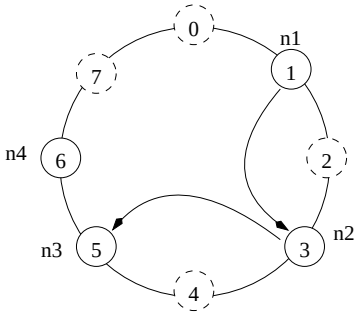
2.3.2 XML Indexes in Structured P2P Systems

The use of XML as the format for data representation introduces additional problems in structured p2p systems. In particular, most current structured p2p systems use document names as the data keys that are mapped to the underlying virtual space formed by the peers. Recent research [118], [125] has extended structured p2p systems by exploiting the content of documents for determining the keys. In particular, a vector describing each document is extracted and used as the key to map the documents to the virtual multi-dimensional space of the network. These vectors, used in information retrieval applications, typically consist of the keywords found in the documents weighted by the frequencies of their appearance in the document. The extracted vector is of much higher dimension than the dimension of the virtual space of the network. Thus, dimensionality reduction is required. This reduction should cause a minimal distortion, that is, the distance of the initial vectors should be approximated by the distance of the new vectors

with the reduced dimensions. Vectors consisting of keywords and their corresponding weights are not suitable for representing XML data, since they do not capture the relationships between the elements (hierarchical structure). Thus, the challenge in this context is mapping the appropriate index keys for XML (both value and path indexes) to the multi-dimensional space provided by the underlying overlay network created by distributed hashing.

In [48], a fully distributed catalog framework based on a structured p2p system is proposed. The system uses Chord [121] as the overlay network. The distributed catalog stores sets of key-summaries information for each peer in the system. The *keys* for XML documents are either element or attribute names, while the *summaries* corresponding to a key are all the possible paths leading to that key. Apart from these structural summaries, value summaries are also used depending on the domain of the key, for example histograms for arithmetic keys. By inserting this additional catalog information into the Chord ring, the system extends Chord's functionality from supporting only keyword-based searches to supporting XPath queries. Each new peer that enters the system constructs its catalog information and each key-summary pair is inserted in the system by the Chord protocol according to the hash values of the keys. The class of XPath queries that the system supports are of the form: $p = a_1[b_1]/a_2[b_2]/.../a_l[b_l]\ op\ value$, where each a_i is a key and each b_i a path. The structural part of the query is handled using the structural catalog information, while the value predicates use the value summaries. The algorithm for query routing proceeds as follows: first all the simple paths ($sp_i = /a_{i1}/a_{i2}/.../a_{i,mi}\ op\ value$) are extracted from the query. The algorithm finds the peer responsible for the next $a_{i,mi}$ and retrieves the set of candidate peers for sp_i using the catalog on that peer. The intersection of all the candidate peers that are produced after all the simple paths are processed is the set of peers that should receive the query.

Figure 2.4 shows an example of how routing is performed in Chord and its extension presented in [48]. The circles correspond to peers in the Chord ring, while dotted circles correspond to logical positions (nodes) in the ring that are not currently occupied by actual peers. The numbers within the circles correspond to the identifier of each logical node in the virtual space while the labels next to them show which actual peers correspond to the logical positions. The system has four peers n_1 to n_4 whose data are presented in Fig. 2.3. The table presents the index information that each peer holds for both Chord and its extension. In particular, the first column presents the finger tables of each peer, for example n_1 stores the information that the successor of identifier 2 is peer 3, of identifier 3 again peer 3 and of identifier 5 peer 5. The finger tables are the same for both systems. For Chord, the index keys are the names of the files shared by the peers. Therefore, the second column of the table shows which files are assigned to each peer after the hash function is applied to their names. In the extended Chord (third column of the table), the indexed keys are the elements found in the shared documents along their corresponding summaries denoted by S_i. Each S_i contains all the possible paths that lead to the corresponding indexed key in the contents of peer n_i. Let us assume that n_1 issues



Peer	Finger Table	Keys (Chord)	Keys−Summaries (extended Chord)
n1	2:3, 3:3, 5:5	d1	camera−S1{device}, S3{device}
n2	4:5, 5:5, 7:1	d2, d3	printer−S1{device}, S2{device, device/local} S4{device/network, device/local}
n3	6:6, 5:5, 7:1	d4, d5	–
n4	7:1, 0:1, 1:1	d6	postscript−S1{device/printer}

Figure 2.4: Indexes in structured p2p systems

the query: "local/printer", searching for all local printers. This query is not supported in Chord, therefore to retrieve this information the user must know the names of the files that contain this information and pose a query using the names of these files. On the other hand, the extended Chord is designed to support exactly this kind of queries. The arrows show the route of the query in the extended Chord.

XP2P [16] also extends Chord to support XML data. The system assumes that each peer stores a set of XML fragments (subtrees of XML documents). Chord is extended by having each peer store the following additional information: the local content of the user's fragments and their related path expressions that are the lists of each fragments child fragments (path expressions stored as PCDATA within sub tags in the fragment) and their super fragment (a path expression of the fragment which is the ancestor of the current fragment). These expressions are hashed into the Chord virtual space and exploited in the finger tables. The hashing technique used is different from that used in Chord. In particular, a fingerprinting technique is proposed based on [107]. The produced fingerprints are shorter than the hash keys used in Chord. Furthermore, a supported concatenation property allows the computation of each short token associated to each path expression to proceed incrementally. In particular, the updates that are performed locally at the fragment roots do not require the recomputation of the modified fragment from scratch; the fingerprint of the parent can easily be concatenated to the new root. The system supports partial and full match lookups where in the first case, we have a partial match to a fragment without unfolding its child fragments, while in the second case, all the sub tags of the fragment are unfolded and the corresponding child fragments are retrieved. The queries are fingerpinted as well and when the fingerprint of a query (either in full or partial lookup) matches the fingerpint of a data fragment, the results are located by using the lookup functionality of Chord. If the system cannot find a match, for instance if some peers are temporarily unavailable, additional techniques based on gradually pruning the query path are deployed to provide the user with at least a partial match.

psiX [109] is another system built on top of Chord. In psiX, a new signature scheme is used based on irreducible polynomials over a finite field. In particular, both documents and queries are mapped onto algebraic signatures using these polynomials. An advantage

of this signature scheme is that it can test if a query matches a document by exploiting the divisibility between the corresponding algebraic signatures of the document and the query, thus performing holistic query evaluation without requiring each path expression to be mapped on the DHT separately as with most of the other approaches and it also allows flexibility in handling the wildcard operator. The signatures of the documents are organized into a forest of hierarchical indexes, so that the divisibility tests can be performed on subsets of documents and not all documents in the system. In addition, the documents are placed into the hierarchical index according to their similarity, which is also derived from the similarity of their signatures more efficiently than if we required access to the actual documents. The hierarchical index is stored into Chord by treating the identifier of each index node as a key, and the entire content of the index node as a value. Thus, the hierarchical index can be accessed by using the Chord lookup functionality.

RDFPeers [20] are based on an extension of Chord, MAAN (Multi-Attribute Addressable Network), which efficiently extends Chord to answer multi-attribute and range queries. Each RDF is viewed as a (subject, predicate, object) triple. Each triple is hashed and stored for each of its values in the corresponding positions in the Chord ring. Furthermore, for arithmetic attributes, MAAN uses order preserving hash functions so as to place close values to neighboring peers in the ring for the evaluation of range queries. Each query is transformed into triples and each value is searched as in Chord.

Another approach that exploits Chord functionalities to achieve efficient management of data described by RDF/S (Resource Description Framework and Schema Language) schemes is presented in [119]. This approach assumes that the RDF/S schemes that appear in the system conform to a number of community schemes, thus implicitly defining groups of peers that maintain data that conform to the same scheme. An RDF/S schema is defined as a set of triplets $(domain(x), x, range(x))$, where x is a property, $domain(x)$ denotes the domain class, i.e., the class the property belongs to and $range(x)$ denotes the range class, i.e., the value of the property. Through the subsumption relationships that exist between properties and classes, the set of triplets are organized into a graph that is then used to describe the different group schemes. To achieve scalability, the peers only advertise part (a fragment) of the group schema their data belongs to, by defining an appropriate view of the RDF/S schema graph. Queries are also defined similarly through RDF/S schema fragments by RQL and query evaluation applies sophisticated pattern matching facilities between views and queries. The RDF/S schema fragments are encoded by a scheme that maintains information about the relationships between nodes in the graph and also the susbsumption relationships between classes and properties. The scheme uses a conceptual cube structure that extends the notion of an adjacency table by adding a third dimension for storing the properties, and through this cube assigns a unique binary number to each RDF/S schema fragment. These unique numbers are then treated as keys in Chord and mapped by order preserving hash functions to the DHT network. The lookup service locates not only the views that exactly map a query, but in a second step by exploiting the subsumption relationships that are supported by the

encoding scheme, it also identifies the views that are subsumed by the query.

A DHT-based approach based on CAN is presented in [134]. The system presumes that the schema of the data is known by all participating peers. An overlay network similar to CAN is built where each dimension in the virtual multi-dimensional space corresponds to either a path level (a level of the path expression corresponding to some element name) or a unique attribute name on a specific path level. The dimensionality of the virtual space depends on the maximum depth of the path expressions in the data and the number of distinct attributes each path level has. The virtual space is viewed as a hyper-rectangle and each distinct path corresponds to a logical node in the overlay network. The overall hyper-rectangles are disjointly partitioned among sub hyper-rectangles with exactly one logical node corresponding to each one of them. Each piece of XML data is mapped to a logical node in the network according to its coordinates that are derived by hashing each element name and attribute that corresponds to each of the dimensions. Each peer in the overlay network keeps catalog information about all the paths that are mapped to it along with its own coordinates and its corresponding hyper-rectangle. Additionally, each peer keeps a routing table where it stores tuples of the form (coordinate, hyper-rectangle, address) for its neighbors in the virtual space. When a query is issued, it is also hashed to provide the query coordinates. When the queries consist of absolute location paths with only parent-child axis the routing can be easily done by using the CAN routing mechanism which is enhanced in order to find the closest neighbor for propagating the query (finding the closest hyper-rectangle). If the query is more complex, it is transformed into one or more absolute location paths and the same mechanism is deployed.

KadoP [4] is another system that relies on a DHT architecture to support complex query functionality for XML data. KadoP indexes the XML data in the form of postings, where each posting encodes information on an element or a keyword. The use of the DHT allows KadoP to assign usually a single peer as responsible for all the postings regarding a term (either keyword or element name). This peer can be efficiently identified by the other peers through the DHT lookup functionality. Thus, given a query, the system combines the postings of the terms that appear in the query in order to locate the peers that maintain relevant data. In particular, for each query the system retrieves the posting lists of all the terms in the query and performs a holistic twig join. The peers that appear in the result of the structural join are then contacted to attain their data and construct the final result. The main limitation of KadoP is the creation of possibly very long posting lists about popular terms that deteriorate system performance. Some techniques addressing this problem were recently proposed in [5]. The first technique splits the posting lists for popular terms horizontally among the peers based on range conditions, similarly to B-trees, thus constructing a type of hierarchical distributed index for each term. Furthermore, the system uses another approach to reduce the size of the data that need to be transferred among the peers to perform the holistic twig join, based on summarizing the posting lists with a Bloom filter based structure that supports checks on whether one element is the ancestor or descendant of another. This information is

then exploited to design a modified join algorithm that is applied on top of the filters thus reducing the required bandwidth consumption.

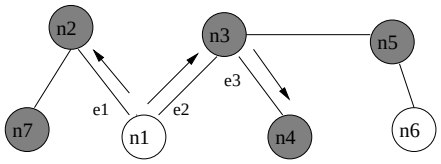
In [99], a hybrid structured (non-DHT) p2p architecture is presented. The superpeers are organized into a hypercube topology that supports efficient broadcasting, while (non-super) peers connect to superpeers in a star-like fashion where each peer connects only to one superpeer. RDF metadata are used to describe the content of peers and to build routing indexes. Queries and answers to queries are also represented using RDF metadata. Two kinds of indexes are maintained at the superpeers: superpeer/peer indexes (SP/P) and superpeer/superpeer (SP/SP) indexes. SP/P indexes at a superpeer store information about metadata usage at each peer connected to it. This includes schema information such as schemas or attributes used, as well as possibly conventional indexes on attribute values. SP/SP indexes contain the same kind of information as SP/Ps, but refer to the direct superpeer neighbors of a superpeer. Queries are forwarded to superpeer neighbors according to the SP/SP indexes and then sent to the connected peers based on the SP/P indexes.

2.3.3 XML Indexes in Unstructured P2P Systems

In unstructured p2p systems, research efforts focus on building space efficient routing indexes for XML documents. Most approaches build path indexes with the use of aggregation and suitable encoding schemes for the paths.

An approach for processing containment queries in p2p networks is presented in [47]. Containment queries exploit the structure that is inherent in XML documents (i.e. book contains author contains name = "John Smith"). XML elements and text words appearing in a document are treated uniformly as keywords that are used as index keys. Local indexes at each peer consist of inverted lists, which map keywords to XML documents stored at the peer. In addition to its local inverted lists, each peer also maintains routing indexes that are called peer inverted indexes. The peer inverted index maps keywords to the identifiers of remote peers. A query is forwarded to remote peers by using the peer-inverted index and set operations are used to minimize the number of relevant destinations. The indexes are constructed when a peer joins the system by exchanging information with other peers. These indexes are smaller than the local indexes because it is expected that a peer would only exchange a small subset of its keywords, such as words that are often found in queries or that are representative of its local data. The result is a p2p system in which each peer has a summary of important data of all other peers. Horizons are used to limit the number of peers for which a peer has summarized information. A peer maps keywords outside of its horizon to peers on the boundary of the horizon that are closer to them.

[79] proposes a decentralized architecture under two models, the open and the agreement model, that differ in the degrees of shared knowledge among the peers. In the open model, each peer is allowed to know about and potentially communicate with every other peer in the system, while in the agreement model, each peer enters into bilateral



Peer	Edge	Keys	Keys (XML−based)
n1	e1	d3, ...	device/local/printer, device/camera, ...
	e2	d4, d5, d6, ...	device/camera, device/local/printer, device/network/printer ...
n3	e3	d5, d6, ...	device/local/printer, device/netwrok/printer, ...

Figure 2.5: Indexes in unstructured p2p systems

agreements with some other peers called its neighbors. Path indexes which treat paths in the XML data tree as the index keys are used as the internal organization of the routing indexes which maintain pointers from each path to the corresponding peers that contain it. Since the information included in a path index can grow excessively, aggregation techniques are proposed to accommodate the routing index in a given space overhead. A compact data structure, called the forwarding table, is derived at each peer from its routing index by aggregating paths with common prefixes.

Figure 2.5 shows an example of on query routing when using simple routing indexes such as in [31] and when XML-based routing indexes are used. The peers n_1 to n_4 hold the same data illustrated in Fig. 2.3. Let us assume that the horizon is of radius 2. Peer n_1 issues the same query of the example of Fig. 2.4, which as in Chord cannot be evaluated when using simple routing indexes and requires from the user to know the names of the files that contain the path expression she is interested in. The gray circles represent the peers that are within peers n_1 horizon. The table shows the routing indexes of peers n_1 and n_3 for edges e_1, e_2 and e_3 respectively, for both simple routing indexes and XML-based ones. The only difference between the two approaches is that filenames are used as index keys in the first case while paths are used in the second. The arrows show the route the query follows when XML-based routing indexes are used.

In our previous work [69], a hierarchical scheme of routing indexes is deployed for query routing and evaluation. The hierarchies are based on the aggregation of the local indexes of the peers and are such that they enable each peer to maintain merged information about its neighbors. The structure used as the local index is the multi-level Bloom filter, which because of its importance in the techniques described in this thesis, will be described in detail in later chapters.

A secure service discovery protocol for p2p systems is described in [33]. Service providers use the Service Discovery Service (SDS) to advertise descriptions of services and services metadata both expressed in XML. Clients use the SDS to locate the services they are interested in. The SDS servers are organized into multiple hierarchies, which can be modified according to each server's workload. Each server is responsible for a particular domain; it receives advertisements and queries from a specific part of the network which form its domain. If a server becomes overloaded, its load exceeds a specific threshold, one

or more child servers are spawned and assigned part of their parent domain. Each internal peer of the hierarchies stores summaries of the descriptions of its children, which are used for query routing. Summaries consist of a single Bloom filter. To insert a description in the filter, it is divided to all possible subsets of the elements and attributes up to a certain threshold. Each query is split to all possible subsets and each one is checked in the index. The system emphasizes on security and access control issues and uses cryptography and authentication to ensure them.

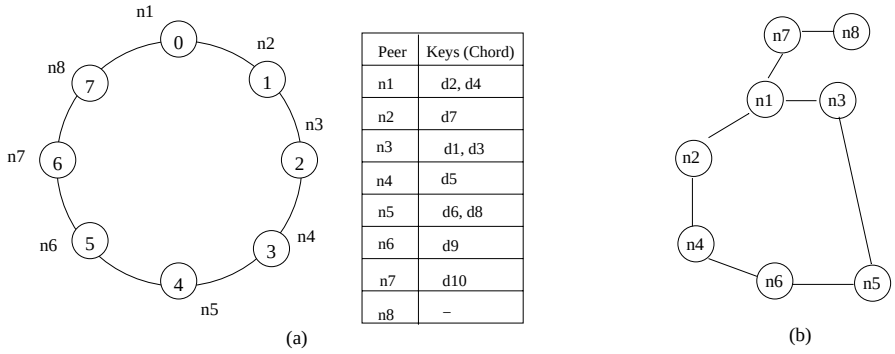
2.4 XML Clustering in P2P Systems

Data clustering refers to grouping data items together to form clusters (groups) of items with common attributes or properties. In centralized systems, query performance may be improved by an appropriate placement of clustered data or indexes in main memory or in disk so that the I/O cost during query evaluation is minimized. In a distributed setting, clustering may be deployed to improve query processing performance by reducing the communication cost by placing similar data at neighboring nodes. In a p2p context, the allocation of similar data to neighboring peers must be dynamic and incremental as new peers enter or leave the system. Furthermore, while clustering algorithms for centralized systems may have global knowledge of all data, in p2p systems, clustering must be distributed among the peers that only have partial knowledge of the system.

In a p2p setting, we further distinguish between (i) clustering similar data items (or indexes of similar data items) so that similar data (or indexes of similar data) are placed in neighboring peers and (ii) clustering peers with similar data items, so that their distance in the overlay network is small. By grouping similar peers, a query that reaches a peer in the cluster finds all other peers with relevant data nearby. Each form of clustering provides a different degree of storage autonomy. Data (and index) clustering violate storage autonomy, since they enforce peers to store specific items.

In structured p2p systems, clustering the data or index can be achieved by using as input to the hash function not just the name of the document but a semantic vector describing its content and structure. If the hash function is order-preserving, similar documents are stored at the same or neighboring peers. Order preserving hash functions are those hash functions that for similar inputs they produce outputs close in the identifier space. For example, the outputs for an order preserving hash function for input “a” and “ab” will differ less than the outputs for “a” and “wb”. pSearch [126] is such a system that maps the documents of the peers on a DHT, based on their term vectors and exploiting only the most important terms. Thus, semantically related documents are “clustered” in the DHT, enabling the search process to limit the space it has to consider. [119] presents another such example in which the encoding scheme deployed and the order preserving hash function that is used places the keys of the RDF/S views with subsumption relationships near each other on the Chord ring.

Clustering the peers affects the topology of the p2p overlay. Issues of interest are how



Peer	Keys (Chord)
n1	d2, d4
n2	d7
n3	d1, d3
n4	d5
n5	d6, d8
n6	d9
n7	d10
n8	–

Figure 2.6: (a) Data (index) clustering and (b) peer clustering

the peers within a cluster are interconnected (*intra-cluster organization*) and how the clusters are connected to each other (*inter-cluster organization*). Usually, query routing proceeds in two steps: first the appropriate cluster is identified and then the query is routed inside the cluster. Different mechanisms for intra-cluster and inter-cluster routing are often required.

Figure 2.6 illustrates the two techniques for clustering in p2p systems, data or index clustering (Fig. 2.6(a)) and peer clustering (Fig. 2.6(b)). The documents stored at each peer is as shown in Fig. 2.3. In Fig. 2.6(a), the peers form a structured p2p system that follows a ring topology. We assume that the vectors extracted from each of the documents are given as input to an order preserving hash function. The output of the hash function maps each document to the virtual ring. The table shows which documents are mapped to each peer. Documents with the same or similar content are mapped to the same or neighboring peers. For example, documents d_2 and d_4 that have the same content are both mapped to logical node 0, while documents d_3 and d_5 that share some common content (i.e., the path expression "device/local/printer") are mapped to neighboring peers in the virtual ring. On the other hand, in n Fig. 2.6(b) where peers clustering is performed what is affected is the created overlay network. In particular, two peers that have similar content are connected in the overlay network. For example, peer n_3 is connected to both n_1 and n_5 with which it has similar contents (document d_2 of n_1 has the same content with d_4 of n_3, while document d_7 of n_5 shares a common path expression with d_4).

With regards to peer clustering, in most current research, the number or the description of the clusters is fixed and global knowledge of this information is required. With Semantic Overlay Networks (SONs), peers with semantically similar content are logically linked to form overlay networks based on a classification hierarchy of their documents, which is defined a priori [32]. Queries are processed by identifying which SONs are better suited to answer them. In [131], clusters of peers are again formed based on the semantic categories of their documents. Sophisticated procedures are proposed for both inter-cluster and intra-cluster load balancing. Similarly in [13], peers are partitioned into topic segments based on their documents. A fixed set of C clusters is assumed, each one corresponding to a topic segment. Knowledge of the C centroids is global. Instead of predefined categories, [41] use a learning approach that based on generalizing the data

the peers share, learns the semantic categories they belong to and then uses those to form the appropriate clusters. In GrouPeer [61], the peers also deploy a gradual learning process through which they discover other peers that may share similar interests. The learning process keeps track of interactions established by two peers for query evaluation. The interactions consist of query rewriting and providing feedback to evaluate the results quality. Clustering in [85] is based on the schemes of the peers and on predefined policies provided by human experts. Besides clustering based on peers content, clustering based on other common features, such as the interests of peers [63, 64], is possible. In [27], peers maintain sets of guide rules, which are formed by the users either explicitly based on their interests, or implicitly through query history, thus defining semantic clusters. In [49], a superpeer-based architecture is proposed under which peers with common interests are organized based on the similarity of their caches. In [36], clustering is first applied on the documents of each peer, and then recursively on the derived feature vectors by selected peer representatives. While this approach does not assume predefined categories, it still requires the use of cluster representatives unlike our uncoordinated protocol. Queries are then first routed through relevant guide rules.

To perform clustering in a p2p system that uses XML as the underlying model, we need to identify which peers have similar content or which XML documents are similar and therefore should be grouped together. Thus, we need to define appropriate similarity measures for XML documents. Clustering for XML documents in centralized applications mostly relies on structural information. For the classification of schema-less data, the authors of [129] combine text terms, structural information in the form of twigs and paths and also ontological knowledge (WordNet [42], [92]) to construct more expressive feature spaces that are then used for the classification. XRules [142] assigns the documents to categories through a rule based classification approach that relates the presence of a particular structural pattern in an XML document to its likelihood of belonging to a particular category. S-GRACE [84] is a hierarchical algorithm for clustering XML documents with a distance metric based on the notion of a structure graph, which is a minimal summary of edge containment in the data. Finally, XClust [81] addresses clustering when schema information in the form of DTDs is available in contrast with the previous methods that can be applied to schema-less data. XClust clusters DTDs based on the semantics, immediate descendants and leaf-context similarity of DTD elements. These centralized methods for XML clustering cannot be directly applied in p2p systems since they require global knowledge of the data or of the schema the data follow. We need to adapt these methods so as to work with incomplete local knowledge acquired by the cooperation of the peers.

In [69], a form of clustering is applied to an unstructured p2p system of XML peers. The peers are organized into hierarchies according to their content similarity, which is derived from the similarity of their routing indexes.

When schema information is available, it may be used to provide for an appropriate mapping of data items or peers to the virtual space of structured p2p systems. If the

schema is known a priori, the virtual address space can be split to sub-spaces each one corresponding to a different part of the global schema. Then, upon entering the system, each peer (or data item) can be mapped to the sub-space of the virtual space that corresponds to the schema of the peer, thus creating clusters of peers that follow the same schema. In [117], the system assumes that all peers share a common topic ontology, and organizes them into concept clusters that are described by a logical combination of ontology concepts. These concept clusters are organized into a hypercube topology. A hypercube topology is followed within the concept clusters as well. In particular, a peer in an ontology-based hypercube carries an address, which concatenates a set of concept coordinates (outer hypercube) and a set of storage coordinates (inner hypercube). In [102], groups of peers choose what data to host based on their own interests, thus defining their interest areas. The interest areas describe index coverage of other groups' data. The assumption is that the data of the peers belong to some categorization hierarchies relevant to a domain, called a multi-hierarchic namespace. Each hierarchy is called a dimension. The coordinates of a data item in the system are expressed as n-tuples. Interest areas are defined as subsets of the cross product of some dimensions and are divided into interest cells. Interest areas are encoded into URNs and the associated index servers are contacted to find relevant data.

In hybrid clustered p2p systems, superpeers act as cluster representatives. Each cluster contains at least one superpeer, which is in charge of the management of the cluster: query processing and peer information management. The superpeers collect their peers' schemas and communicate with each other for query evaluation. In SQPeer [66], peers are grouped based on their RDF-schema similarity. The system uses active schemas that are fine-grained schema advertisements that contain only information about what is actually stored in a peer. The peers that hold RDF descriptions conforming to the same RDF schema are clustered together. In XPeer [115], peers are logically organized into clusters that are also formed on a schema-similarity basis, whenever this is possible. Superpeers are organized to form a tree, where each peer hosts schema information about its children; superpeers having the same parent form a group. In rule-based clustering [86], peers are registered and grouped in clusters based on cluster specific rules that describe the properties that each peer in the cluster should possess. These rules are provided by a cluster's administrator.

2.5 Query Evaluation for XML Data

When dealing with query evaluation over distributed XML data, we can distinguish different issues that are important and determine the efficiency of the evaluation. These issues include, the actual distributed query processing, the type of evaluation that depends on the querying model, such as keyword querying or approximate query evaluation and replication or caching techniques that are used to accelerate the evaluation.

2.5.1 Distributed Query Processing

Query processing in traditional distributed systems can be divided into four phases: query decomposition, data localization, global and local query optimization. Firstly, a relational query is decomposed into an algebraic query on global relations using techniques from centralized databases. Secondly, data distribution information is exploited to localize the algebraic query. The global query optimization phase receives an algebraic query on data fragments and finds an optimal strategy for its execution taking into account selectivity estimations and communication costs. At the last phase, each node receiving a query subplan tries to optimize it locally using centralized algorithms. The first three phases are performed centrally with global knowledge, while the fourth one is executed on each participating node with the knowledge available locally.

Query processing in a p2p system strongly depends on the type of the index. If there is a central index, the traditional technique can be deployed for query processing, otherwise there is no clear distinction between the four phases of query processing as in traditional distributed systems. Query processing must be coordination free and the execution plan must be constructed dynamically.

We consider two generic approaches to this issue. The first approach is based on evaluating the query incrementally. A peer initiates the execution of a query, and the query is routed among the peers using the indexes. The execution plan evolves by accumulating partial results evaluated at the local peer and locating the next peer with relevant data that needs to process the query. The other approach resembles the traditional approach with the known four phases but localization of data is achieved by using the indexes placed at each peer. In particular, the initiator of a query transforms the query into a path list and sends this list to remote peers where with the use of index-joins (similar to semi-joins) between the list and the peers' path indexes, the peers that store data relevant to the query are retrieved. In a way, this method relies on *index shipping* rather than query or data shipping that is common in traditional distributed systems.

If the system is a hybrid p2p system, the queries are sent to the superpeers who are responsible for constructing the execution plan and coordinating the query evaluation.

In non p2p distributed XML repositories, the schema of the data is known. In [123], distribution is implemented by having links from the local XML data to XML objects (i.e. by using XLink [35]) at remote system nodes. The data is represented by a rooted labeled graph. The model distinguishes between local links that point to local objects and cross-links that point to remote objects. Every node determines which of its data have incoming edges from other nodes (input data nodes) and which have outgoing edges to remote objects (output data nodes). Copies of the external data nodes are added to the node's graph. Given a query, an automaton is computed and sent to every node. Each node traverses only its local graph starting at every input data node and with all the states in the automaton. When the traversal reaches an output data node, it constructs a new output data node with the given state. Similarly, new input data nodes are also constructed. Once the result fragments, which consist of an accessibility graph that has

the input and output data nodes and edges between them if and only if they are connected in the local fragment, are computed they are sent to the origin of the query. The client at the origin of the query assembles these fragments by adding missing cross-links, and computes all the data nodes accessible from the root. The algorithm requires only four communication steps and the size of the data exchanged depends only on the number of cross links and the size of the query answer.

Optimizing the cost of communication in answering XPath queries over distributed data is considered in [124]. The approach is based on the client-server model. The main idea is to use minimal views that describe a query's results which are sent to a client, such that we can avoid the redundancy met in such results where the same data may appear many times. The system leaves part of the evaluation of the query to the client who may have to extract all the answers from the minimal view to obtain the results to the initial queries.

Query processing in [18], [50] is based on shipping index entries among nodes and efficiently evaluating chains of local joins of indexes that can be viewed as the problem of the efficient evaluation of a chain of semi-joins. The node that issues a query determines the partitions of the query into path indexes lists that need to be sent to other nodes and instructs these nodes to compute the intermediate results. Then, the index results produced at the first node are sent to the second one to compute the join, and the resulting index is send to the third and so on. The final result is sent to the initial node where with the help of the RepositoryGuide the nodes that store the XML fragments defined by the resulting path index are retrieved. Similarly in the distributed RDF repository of [122], the main issue is to find the optimal ordering in which to perform a chain of joins of simple path expressions that are extracted from each query.

Mutant Query Plans (MQPs) [102] extend the weak query capabilities and limitations in index scalability and result quality of current p2p systems. A mutant query plan is an algebraic query plan graph, encoded in XML that may also include verbatim XML-encoded data, references to resource locations (URLs), and references to abstract resource names (URNs). Each MQP is tagged with a target, the peer where the results need to be sent. An MQP starts as a regular query operator tree at the client peer and is then passed around from peer to peer accumulating partial results, until it is fully evaluated into a constant piece of XML data and returned to the client. A peer can choose to mutate an incoming MQP in two ways: it can resolve a URN to one or more URLs, or a URL to its corresponding data. The peer can also reduce the MQP by evaluating a sub-graph of the plan that contains only data at the leaves, and substituting the results in place of the subgraph. Resolving URLs is done either by connecting to the specified peer or by forwarding the MQP to it. For the URN the system's catalog is used. The cost model introduced in [3] for dynamic XML documents, is also used by a peer to decide on a query execution plan that minimizes its own observed cost. Each query is split to subqueries. A peer tries to satisfy locally as much of the query as it can and then forwards the remaining subqueries to the associated peers that follow the same procedure.

The same ideas are exploited in [19] for Boolean queries and [29] for more general XPath queries. The approaches do not make any assumptions about how the XML documents are distributed and support query evaluation over both horizontally and vertically fragmented XML trees. In [19], the whole query is sent to each site which locally partially evaluates the query and sends the results in the form of compact boolean functions to a site that plays the role of the coordinator and which combines these partial results to derive the final result. [29] extends the previous ideas for generic queries and while requiring more visits of each site and increased data traffic that however only depends on the number of fragments in the result set, maintains the efficiency the original approach achieved.

In [127], data is modelled in XML and peers represent their schemas in XML Schema. Query evaluation is incremental with an additional logical-level search where data are located based on schema-to-schema mappings. In particular, instead of a local index, each peer maintains mappings between its own schema and the schemas of its immediate neighbors. Mappings are described as query expressions using a subset of XQuery. Peers are considered as connected through semantic paths of such mappings. Peers may store mappings, data or both. Query processing starts at the issuing peer and is reformulated over its immediate neighbors; in turn, these neighbors reformulate the query over their immediate neighbors and so on. Whenever the reformulation reaches a peer that stores data, the appropriate query is posed on that peer, and additional results may be appended to the query result. Various optimizations are considered regarding the query reformulation process such as pruning semantic paths based on XML query containment, minimizing reformulations and pre-computing some of the semantic paths.

In XPeer [115], peers export a description of their data in the form of a tree-shaped DataGuide, which is automatically inferred from the data by a tree search algorithm. The query language of choice is the FLWR subset of XQuery without universally quantified predicates and sorting operations. Query compilation is performed in two phases by the superpeers. In the first phase, a peer issues a query and translates it into a location-free algebraic expression. In the second phase, the query is sent to the superpeer network for the compilation of a location assignment. After the location assignment is completed, the query is passed back to the peer that issued it for execution, minimizing the load of the superpeer network. The peer applies common algebraic rewriting and then starts the query execution: the query is split into single-location subqueries that are sent to the corresponding peers. Subqueries are locally optimized and the results are returned to the initial peer which executes operations such as joins involving multiple sources.

In SQPeer [66], queries are posed in RQL, according to the RDF schemas that are known to each peer. When a peer receives an RQL query, it parses the query and by obtaining the involved path creates the corresponding query pattern graph, which describes the schema information employed by the query. For each path pattern, the algorithm discovers the peers that contain the required data to answer it. If the schema of the peer is subsumed by the selected query path, then this query path is annotated with the name of the corresponding peer. This is done for all known schemas that belong to remote peers.

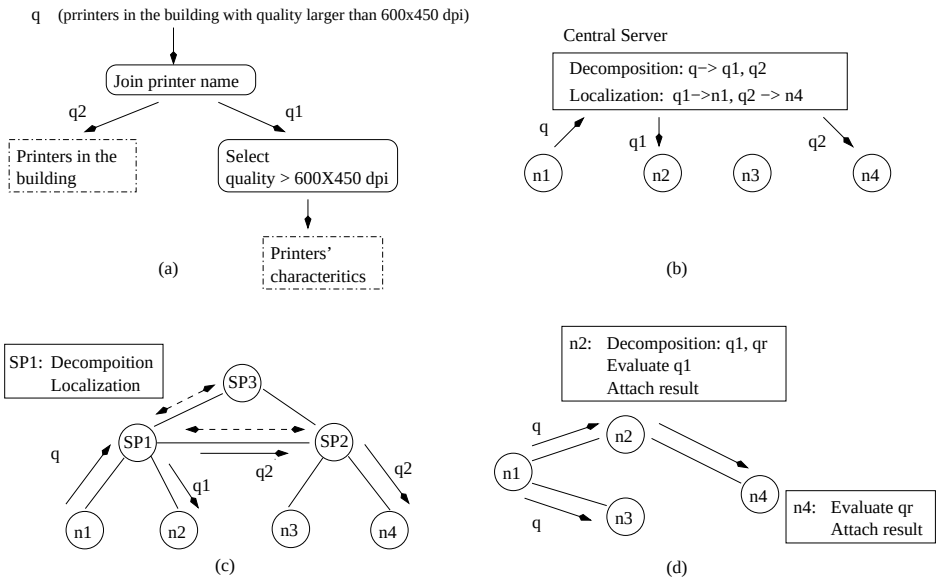


Figure 2.7: Distributed query processing scenarios

The result is an annotated query pattern graph with information for all peers that need to be contacted to answer the query. The query processing algorithm receives as input the annotated query graph and outputs the execution plan according to the underlying data distribution. Query processing uses channels through which the peers exchange query plans and results.

Figure 2.7 illustrates the different scenarios for distributedly processing the query illustrated in Fig. 2.7(a) issued by peer n_1. The query asks for all the printers in a building that have printing quality larger than 600x450 dpi. Let us assume that the information about the available printers in the building is stored in peer $N4$ and the information about various printers characteristics in peer n_2. The query can then be decomposed in q_1 retrieving all the printers with the desired quality from n_2 and q_2 that retrieves all the available printers in the building from n_4. The lines in the figure illustrate the overlay network connections while the arrows show the route the query takes. In Fig. 2.7(b), we illustrate the traditional client-server scenario where the query is forwarded to a central server that has the available information for decomposing, localizing and optimizing globally the query. The server constructs the distributed query plan and forwards the two sub-queries to the corresponding peers for local optimization and evaluation. The results are returned to n_1 who performs the final join. In Fig. 2.7(c), the superpeer based scenario is illustrated. Peer n_1 sends the query to the superpeer it is connected to, namely SP_1. SP_1 interacts with the other superpeers in the network (dotted arrows) and gathers the required information to construct the distributed query plan. It then forwards the sub-query q_1 to n_2 that is attached to it, and q_2 to SP_2 who in turn forwards the query to n_4. Again the results are returned to n_1 who performs the final join. Finally, in Fig. 2.7(d) we demonstrate the fully-distributed incremental query evaluation strategy. n_1 issues the query and forwards it to its neighbors, following a flooding-based

approach. When n_2 receives the query, it realizes that it can evaluate a part of it (q_1). It evaluates its part and attaches to the query plan the computed result. It then forwards the query further to its own neighbor n_4. Peer n_4 evaluates the remaining part of the query and it can either perform the final join itself or just attach its own partial result and forward it to n_1 for the evaluation of the final join.

2.5.2 Approximate Query Processing

Approximate processing of queries over XML data has been addressed in centralized scenarios where data is located at a single server and their schema is known. Usually, the approaches are based on generalizing, relaxing, the original XPath query to attain approximate results that are then ranked according to their relevance to the original query.

[88] uses a relaxation technique [8, 9] on XPath queries based on edge generalization, leaf deletion and subtree promotion. For ranking in [88], the tf*idf measure is extended to account for predicates both on content and structure. Each query node is assigned to a server maintaining a priority queue of partial matches. For each match at the head of its queue, a server computes a set of extended matches with their scores and updates the set of top-K matches maintained by the system. TopX [128] is a query engine that can support approximate top-K results using probabilistic score estimations, but unlike our approach relies on relaxing values rather than structure and uses the latter only to reduce the processing cost. Compact structural summaries for XML data such as [106, 105, 6] that provide selectivity estimations for twig queries can be used to support approximate query processing.

2.5.3 Keyword-based Query Evaluation

Keyword queries have been widely studied with respect to XML documents. A common approach to maintain the XML structural semantics through keyword queries is to adopt lowest common ancestor semantics for the interpretation of the results. That is, the result defined by the most specific lowest common ancestor is considered as the most relevant to the query. In [139] and [140] lowest common ancestor semantics are applied to query evaluation to return the most relevant results to a query, i.e., the most specific subtree that contains all keywords and eager, stack-based algorithms are used for query evaluation. XRank [56] is also based on lowest common ancestor semantics. Additionally, the approach considers the proximity of the keywords in a flat text as well as the reference (IDREF) elements that are included in a document, to compute the rank of the result. It also requires traversing the XML graph for query evaluation. Schema-free XQuery also relies on lowest common ancestor semantics, defining the meaningful lowest common ancestor and incorporating this functionality into XQuery in order to eliminate the need for knowing the exact schema of a document to pose and XQuery. Instead, the MLCA takes the structural part of the query when no schema knowledge is available. In contrast,

XKeyword [60] is an approach for supporting pure keyword queries that however relies on the existence of a schema the documents in the collection follow. In particular, based on this schema XKeyword builds appropriate path indices summarizing the data information and then evaluates queries on top of them. It is evident that if the documents in a collection are heterogeneous and we need a schema and the indices for each document, the approach cannot be applied when dealing with large collections. Besides keyword-based querying that focuses mainly on content and incorporates the structure through the LCA semantics we described, other approaches define different ways for querying XML based on IR principals [45]. In [28], XSEarch a search engine for XML based also on IR principles is defined. This engine satisfies queries by identifying fragments that are meaningfully related across different XML documents.

2.5.4 Replication and Caching

As usual, in a p2p system, replication is used both to improve performance by balancing the load or placing copies of data close to their requestors and to increase data availability in case of peer failures. In particular, due to the lifetime autonomy of the peers, replication is essential for keeping data available even when some of the peers storing them go offline. Replication refers either to the data items or their indexes. Issues of interest include determining which items to replicate, where to place the replicas and how to keep them consistent. Handling these issues depends on the overlay topology and the routing mechanism in use. With regards to the number of replicas, there are various approaches. At one extreme, *uniform replication* creates the same number of replicas for all data items regardless of their popularity. At the other extreme, *proportional replication* creates for each data item a number of C replicas, where C is proportional to its popularity, i.e. the number of queries that concern the given item. While in proportional replication, popular queries are satisfied very efficiently since a large number of copies for the requested data is available across the system, unpopular data require many searching steps thus compromising the overall system performance. On the other hand, with uniform replication, a large portion of the system resources is wasted in replicating data that appear in queries very rarely. Between these two extremes, *square-root replication* [26] creates copies so that for any two data items the ratio of replication is the square root of the ratio of the query rates. Square-root replication provides better results by leveraging the efforts for finding popular and unpopular data items.

Two different strategies are mainly used for placing replicas: path and owner replication. With *owner replication*, whenever a peer issues a successful query about a specific data item, a replica of the item is created at that peer. with *path replication*, copies of the requested data are stored at all peers along the path in the overlay network from the requestor peer to the provider peer [87]. Although path replication outperforms owner replication, it tends to replicate data items to peers that are topologically along the same path, which somewhat hurts the system performance. For updating the copies, the authors in [113] propose an investment policy where each individual peer propagates an

update only if it estimates a benefit from doing so, i.e., an investment return.

For structured p2p systems that map indexes of data items to peers, the index entries must be extended to maintain the identifiers of all the peers that hold copies of the item indexed. Load balancing techniques based on replication are proposed for the structured p2p system of [48]. Since some objects may be more popular, thus creating a considerable load to the peers that store them, two methods are proposed for splitting this load with other peers, namely the split-replicate and the split-toss methods that split catalog information among peers. A peer increases the level of a popular index key, where level is the number of path steps contained in the key (initially set to 1), and either replicates the new keys to the corresponding peers (according to the hash function) in the ring (split-replicate), or only sends them there and then discards them (split-toss). The peer also creates a mapping in its catalog for the new locations of the key, which it hands over to peers that request it. Once they obtain these mappings, they query one of the new locations in a round-robin fashion.

When XML is used for data representation, an important issue is the granularity of replication and distribution. The top-down design of a distributed XML repository is described in [18], [50]. A global conceptual schema is used by a simplified structure called RepositoryGuide, which is a tree-structured index that resembles a DataGuide [52]. The system supports path and tree pattern queries. The fragmentation scheme decomposes the RepositoryGuide into a disjoint and complete set of tree-structured fragments that preserve the data semantics. A sublanguage of XPath is used for data fragmentation that supports vertical fragmentation, which is solely based on the selection of node types through path properties. The language can also be extended to support horizontal fragmentation, which includes conditions and branching, although some consistency issues arise. The allocation phase consists of three steps: determining which fragments to allocate at which system nodes, placing schema structures at local nodes and suitable instances of fragments at each node. For the first step, existing methods from distributed databases are used. For the second step, the RepositoryGuide is fully distributed among the nodes. Finally, for the third step, the global context of each fragment is kept by storing the data path from the global root node to the root of the local fragment. To this end, three indexes are used: a path index that encodes the global context of local fragments, a term index that allows processing of queries that include conditions on terms and an address index that stores the physical addresses of the fragments. Space efficient path indexes are constructed with the use of a path identification scheme. Because of their small size, path indexes are replicated at all system nodes, while term and address indexes are distributed among them.

In [3], distribution and replication of dynamic XML documents in a p2p system is discussed. Dynamic XML documents are XML documents that contain materialized XML data that are part of the document and intentional data that can be produced by service calls. Since dynamic documents may contain calls to services on other peers, some form of distribution is inherently part of the model. External edges are added to the

XML document to point to peers that store other parts of the documents to allow for a higher form of distribution. A cost model used for query processing and also for deciding whether to replicate some parts of other peers' documents to increase the efficiency of query execution is introduced. Replication addresses not only the replication of services but also of the data these services utilize.

2.6 Summary

In this chapter, we provided a brief description of data management issues that arise in the design of p2p systems that use XML as the underlying data model. We described the main characteristics of p2p systems and the properties of the XML data model and studied the issues of indexing, query processing, clustering and self-organization of peers in such systems. For each of these issues we presented solutions proposed by current research in the area. Table 2.1 provides a summarized grouping of the techniques proposed to deal with each of the data management issues.

We will discuss more specific technicals issues with comparison to our proposed solutions in the respective chapters when needed.

A previous version of this chapter appears as a survey article in [71].

Table 2.1: Summary of XML data management issues in p2p systems

<table>
<tr><th></th><th></th><th>Structured P2P</th><th>Unstructured P2P</th><th>Non P2P</th></tr>
<tr><td rowspan="2">Indexing</td><td>Pure P2P</td><td>Mapping of paths onto the multi-dimensional virtual network [48], [4], [20], [118], [125], [16], [5], [134], [109], [119]</td><td>Building of space efficient routing path indexes through aggregation and encoding [69], [47], [79]</td><td rowspan="2">Not required since the data location is known</td></tr>
<tr><td>Hybrid P2P (superpeers)</td><td colspan="2">Superpeers responsible for routing hold all index information and propagate the queries to their peers
Different routing protocols and index structures between superpeers and between superpeers and peers [99], [33]</td></tr>
<tr><td rowspan="2">Clustering</td><td>Pure P2P</td><td>Clustered index
DHT-based clustering of nodes based on their content by adopting the input of the hash function to split the multi-dimensional space into regions of peers with similar content [126], [119]
Non-DHT clustering based on schema similarity [102], [117]</td><td>Formation of peer clusters based on schema or content similarity, with the use of routing indexes
Different mechanism used for inter-cluster and intra-cluster routing [69]</td><td rowspan="2">Centralized techniques used for data allocation</td></tr>
<tr><td>Hybrid P2P (superpeers)</td><td>Clustered index built only upon superpeers</td><td>Superpeers used as cluster representatives
Inter-cluster communication between superpeers [115], [66], [86]</td></tr>
<tr><td rowspan="2">Replication</td><td>Pure P2P</td><td rowspan="2">Data allocation with the use of distributed hashing
Replication requires additional mappings among nodes responsible for the data (or indexes) and the nodes that hold the replicas [48]</td><td rowspan="2">Replication increases data availability and performance [3]</td><td rowspan="2">Fragmentation-allocation of XML [18], [50]</td></tr>
<tr><td>Hybrid P2P (superpeers)</td></tr>
<tr><td rowspan="2">Query Processing</td><td>Pure P2P</td><td colspan="2">Coordination-free based on the type of index
No clear distinction among query processing phases
Dynamic construction of the execution plan
Query passed around among peers accumulating partial results [102], [19], [29],[3], [127]</td><td rowspan="2">Traditional techniques (Distinct phases: query decomposition, query localization, global and local query optimization) [123], [124]
Index shipping [18], [50]
Ordering of joins [122]</td></tr>
<tr><td>Hybrid P2P (superpeers)</td><td colspan="2">Superpeers responsible for the construction and the coordination of the execution plan
[115], [66]</td></tr>
</table>

Chapter 3

Hash-based Indexing for Semi-Structured Data

In this chapter, we introduce two novel index structures for summarizing XML documents. The first structure, the multi-level Bloom filter supports boolean queries, while the second provides selectivity estimations. Both structures are designed so as to be appropriate for use in large scale distributed environments such as p2p systems. Therefore, the structures meet the following requirements to abide with p2p and XML properties and: (a) are scalable, i.e., storage and processing wise efficient, (b) are incrementally updatable, (c) do not assume that the documents follow any particular schema and (d) maintain the structural properties of the XML documents.

We first present the querying model our structures support and the requirements we want to meet with their design (Section 3.1). We explain why earlier path indexes are not suitable for use in our context and present the hash-based structures on which we base our novel indexes (Section 3.2). We then introduce the index structure for boolean queries (Section 3.3) and for selectivity estimations (Section 3.4), along with the corresponding algorithms for their construction, query evaluation and update and a short theoretical analysis of their performance. We also include an experimental evaluation of the structures (Section 3.5) and conclude with a brief summary (Section 3.6).

3.1 Problem Definition

We consider a collection of heterogeneous schemaless XML documents that follow the tree structured model. We focus on queries that belong to an XPath subset, XPath$^{\{/,//,*\}}$, consisting of expressions of the form: $a_1p_1a_2p_2a_3...a_np_n$, where $p_i \in V \cup \{*\}$, $*$ is the wildcard operator and a_i is the child or descendant-or-self axis ("/" and "//"). Such path expressions form the building blocks of more advanced querying. We also handle twig queries (e.g. tree-pattern queries) by decomposing them to appropriate path expressions in XPath$^{\{/,//,*\}}$ and process them separately. A path expression is evaluated sequentially by finding an element p_1 anywhere in the document and nested within it an element p_2, and so on, until p_n is encountered. Its result is the set of fragments rooted at the p_n nodes found in the given XML tree. We say a query q *matches* a collection of documents $\mathcal{D}$, if the result set of the evaluation of q against the documents in $\mathcal{D}$ is non-empty. We denote as *result(q, D)* the fragments included in the result set.

We discern between two type of queries: *boolean* and *estimation* queries. For boolean queries, we are only interested in whether a query q *matches* a collection of documents $\mathcal{D}$, that is, if it has any results in $\mathcal{D}$, and in that case we write *match(q,* $\mathcal{D}$*) = True*. For estimation queries, we are interested in the number of the results q has in collection $\mathcal{D}$ and return *result(q,* $\mathcal{D}$*)* as our result.

Our goal is to define index structures for summarizing XML data that support the evaluation of both types of queries efficiently in p2p systems.

In particular, we want the summaries to be scalable so as to deal with the large volume of data that is available in p2p systems. Thus, the summaries should be storage efficient. Since we are dealing with indexes in a distributed system where one of the main goals for improving system performance is to minimize the communication cost that is involved in query evaluation, the need for storage efficient summaries is crucial as the summaries are often exchanged between peers for query evaluation and overlay maintenance. Furthermore, the summaries should support incremental updates efficiently to cope with p2p system dynamics, as the content and the population in p2p systems changes frequently. Thus, we want the structures to be maintained up-to-date without requiring that they are reconstructed for each update.

Besides storage and maintenance efficiency, processing efficiency is also important. The summaries should support efficient query evaluation and process boolean queries and provide selectivity estimations without requiring access to the actual data. The summaries should maintain the structural properties of the data they summarize, i.e., preserve the hierarchical relationships between the tree elements so as to support queries belonging in XPath$^{\{/,//,*\}}$. Finally, the summaries should handle heterogeneous documents without requiring that they conform to any schema or DTD.

3.2 Preliminaries

Indexing methods for XML provide efficient ways to index XML, support complex queries and offer selectivity estimations. However, most such structures were designed for use in centralized settings, thus, they cannot be directly applied in p2p systems.

3.2.1 Path Indexes and Summaries

The method in [114] encodes paths as strings, and inserts them into an index that is optimized for string searching. Evaluating queries involves encoding the query as a search key string, and performing a lookup in the index. The XSKETCH synopsis [104] relies on a generic graph-summary where each node only captures summary data that record the number of elements that map to it. Emphasis is given on the processing of complex path queries. APEX [25] is an adaptive path index that utilizes frequently used paths to improve query performance. It can be updated incrementally based on the query workload. The path tree [6] has a path for every distinct sequence of tags in the document. If it exceeds main memory space, nodes with the lowest frequency are deleted. In [103], a signature is attached to each node of the XML tree, in order to prune unnecessary subtrees as early as possible while traversing the tree for a query. In [83] signatures are used to construct a lightweight tree index where components extracted from the documents are used as keys. A query signature is constructed, based on the components, and used as a key to traverse the tree and retrieve all document signatures that contain it. [46] is an attempt to reduce the index space overhead by either using compression or auxiliary data structures (XS tree) that add structural information to inverted lists so as to support efficient information retrieval for XML documents.

The focus of all these structures is on answering complex queries efficiently. Some of them, though use reduced storage requirements with respect to the original data, still are not storage efficient enough to be used in p2p systems where index shipping is often deployed in query evaluation. Furthermore, though some were designed to support incremental updates, these updates are mostly in the scope of a document; the structures have no efficient way of supporting the aggregation operation between two structures that summarize different documents to produce a single merged one, a property we require to build a single index for a collection of XML documents that is incrementally updated.

Therefore, we study next index structures that are appropriate for use in distributed systems and examine whether they can be used for summarizing XML documents.

3.2.2 Bloom Filters

Bloom filters [15] are a data structure that has received much attention as a means to summarize information in distributed systems and has seen many uses such as web cache sharing [40], query filtering and routing [54, 59, 95, 111] and free text searching [108].

Bloom filters are compact data structures for probabilistic representation of a set that support membership queries ("Is element x in set X?"). Consider a set $x = \{x_1, x_2, ...,$

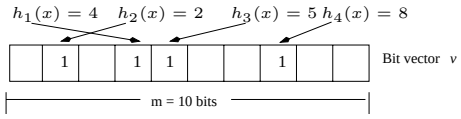


Figure 3.1: A (simple) Bloom filter with $k = 4$ hash functions

$x_N\}$ of N elements. The idea is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, h_1, h_2, ..., h_k, each with range 1 to m. For each element $x \in X$, the bits at positions $h_1(x)$, $h_2(x)$, ..., $h_k(x)$ in v are set to 1 (Fig. 3.1). A particular bit may be set to 1 many times. Given a query for y, the bits at positions $h_1(y)$, $h_2(y)$, ..., $h_k(y)$ are checked. If any of them is 0, then certainly $y \notin X$. Otherwise, we conjecture that y is in the set although there is a certain probability that we are wrong. This is called a "false positive" and it is the tradeoff for Bloom filters' compactness. The parameters k and m should be chosen such that the probability of a false positive is acceptable.

To support updates of the set X we maintain for each location i in the bit array a count $c(i)$ of the number of times that the bit is set to 1 (the number of elements that hashed to i under any of the hash functions). All counts are initially set to 0. When a key a is inserted or deleted, the counts $c(h_1(x))$, $c(h_2(x))$, ..., $c(h_k(x))$ are incremented or decremented accordingly. When a count changes from 0 to 1, the corresponding bit is turned on. When a count changes from 1 to 0 the corresponding bit is turned off.

Bloom filters are appropriate as an index structure for p2p systems in terms of scalability, supporting incremental updates and distribution. However, they are not suitable for XML data summarization as they have no means of supporting the hierarchical relationships between the elements they summarize.

Bloom Histograms [135] extend Bloom filters to summarize XML data. They are designed for supporting selectivity estimations for XPath expressions. The idea is to use a histogram to summarize the frequency of all paths of an XML tree, while using a Bloom filter to summarize the corresponding values, i.e. the paths that fall into each bucket of the histogram. To construct a Bloom Histogram BH with b buckets for an XML tree, the set of paths in the tree is partitioned into b disjoint sets of paths, $path_i$. The *Bloom Histogram* is a two-column table $BH(BF_i, val_i)$, $1 \leq i \leq b$, where BF_i is a Bloom filter of all paths in $path_i$ and val_i is a representative value of the frequencies of all paths in $path_i$.

Based on Bloom filters we define appropriate structures for summarizing XML documents and combining them with histograms we also present a structure that supports selectivity estimations.

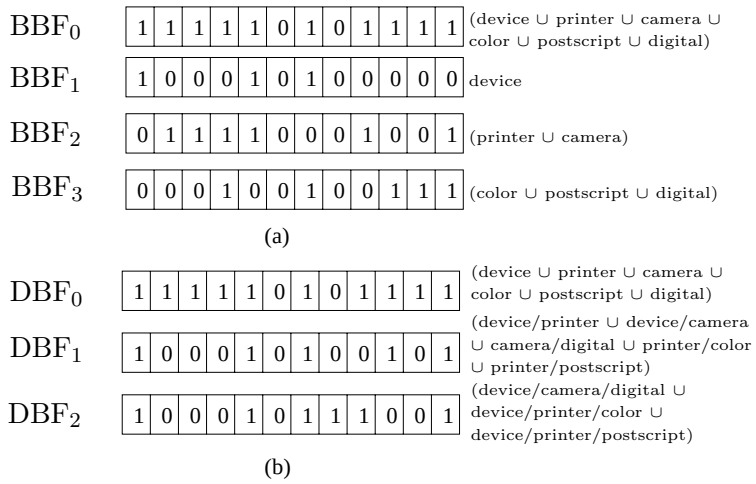


Figure 3.2: The multi-level Bloom filters for the XML tree of Fig. 2.2: (a) the Breadth Bloom filter and (b) the Depth Bloom filter

3.3 Index Structures for Boolean Queries

We introduce two new data structures based on Bloom filters, the *Breadth Bloom filter* and the *Depth Bloom filter* that aim at supporting boolean queries. The two structures are based on two alternative ways of hashing XML trees.

3.3.1 Breadth and Depth Bloom Filters

Let *Tree* be an XML tree with L levels and let the level of the root be level 1. The *Breadth Bloom Filter* (BBF) for an XML tree *Tree* with L levels is a set of simple Bloom filters $\{\text{BBF}_1, \text{BBF}_2, \ldots \text{BBF}_{l_b}\}$, $l_b \leq L$. There is one simple Bloom filter, denoted BBF_i, for each level i of the tree. In each BBF_i, we insert the elements of all nodes at level i. To improve performance and decrease the false positive probability in the case of $l_b < L$, we may construct an additional Bloom filter denoted BBF_0, where we insert all elements that appear in any node of the tree. For example, the BBF for the XML tree in Fig. 2.2 is a set of 4 simple Bloom filters (Fig. 3.2(a)). Note that the BBF_is are not necessarily of the same size. In particular, since the number of nodes and thus keys that are inserted in each BBF_i ($i > 0$) increases at each level of the tree, we analogously increase the size of each BBF_i. Let $|\text{BBF}_i|$ denote the size of BBF_i. As a heuristic, when we have no knowledge for the distribution of the elements at the levels of the tree, we set: $|\text{BBF}_{i+1}| = dgr\ |\text{BBF}_i|$, $(i < l_b)$, where *dgr* is the average degree of the nodes. For equal size BBF_is, BBF_0 is the logical OR of all BBF_is, $1 \leq i \leq l_b$.

Depth Bloom filters provide an alternative way to summarize XML trees. We use different Bloom filters to hash paths of different lengths. The *Depth Bloom Filter* (DBF) for an XML tree *Tree* with L levels is a set of simple Bloom filters $\{\text{DBF}_0, \text{DBF}_1, \text{DBF}_2, \ldots, \text{DBF}_{l_b-1}\}$, $l_b \leq L$. There is one Bloom filter, denoted DBF_i, for each path of the tree with length i, (i.e., a path of $i + 1$ nodes), where we insert all paths of length i. For example, the DBF for the XML tree in Fig. 2.2 is a set of 3 simple Bloom filters (Fig.

3.2(b)). Note that we insert paths as a whole, we do not hash each element of the path separately; instead, we hash their concatenation. We use a different notation for paths starting from the root. This is not shown in Fig. 3.2(b) for ease of presentation.

Regarding the size of the filters, as opposed to BBF, all DBF_is have the same size, since the number of paths of different lengths is of the same order. The maximum number of keys inserted in the filter is order of dgr^L for a tree with maximum degree dgr and L levels.

We collectively call Breadth and Depth Bloom filters as *multi-level Bloom filters*.

3.3.2 Lookup Evaluation

BBF Search: The look-up procedure, that checks if a BBF matches a query, distinguishes between: path queries starting from the root and partial path queries. In both cases, first we check whether all elements in the query appear in BBF_0. Only if we have a match for all, we proceed in examining the structure of the path. For a root query: $p_1/p_2/\ldots/p_{l_q}$ every level i from 1 to l_q of the filter is checked for the corresponding p_i. The algorithm succeeds if we have a hit for all elements. For a partial path query, for every level i of the filter: the first element of the path is checked. If there is a hit the next level is checked for the next element and the procedure continues until either the whole path is matched or there is a miss. If there is a miss, the procedure repeats for level $i + 1$. For paths with the containment operator *, the path is split at the *, and the sub-paths are processed. All matches are stored and compared to determine whether there is a match for the whole path. The procedure is implemented by the algorithms described in detail in Alg.1, Alg. 2 and Alg. 3.

Let l_q be the length of the path and l_b be the number of levels of the filter. (We exclude the Bloom on top-it). In the worst case, we check l_b - l_q + 1 levels for each path, since the path can start only until that level. The check at each level consists of at most l_q checks, one for each element. So the total complexity is $l_q(l_b$ - l_q + 1)=O(l_b l_q). When the path contains the * operator, it is split into two sub-paths that are processed independently with complexity O(l_b $l1_q$) + O(l_b $l2_q$)<O(l_b l_q). The complexity for the comparison is O($l_q{}^2$), since we have at most (l_q + 1) / 2 *. For a path that starts from the root the complexity is O(l_q).

DBF Search: The look-up procedure (Alg. 4), that checks whether a DBF matches with a path query, first checks whether all elements in the path expression appear in DBF_0. If this is the case, we continue treating both root and partial paths queries the same. For a query of length l_q, every sub-path of the query from length 2 to l_b is checked at the corresponding level. If any of the sub-paths does not exist then the algorithm returns a miss. For paths that include the containment operator *, the path is split at the * and the resulting sub-paths are checked. If we have a match for all sub-paths the algorithm succeeds, else we have a miss.

Consider a query of length l_q. Let l_q be smaller than the number of the filter's levels. Firstly l_q sub-paths of length 1 are checked, then l_q - 1 sub-paths of length 2 are checked

and so on until we reach length l_q where we have 1 path. Thus the complexity of the look-up procedure is $l_q + l_q$ - 1 + l_q - 2 +...+ 1 = $l_q(l_q + 1)/2 = O(l_q^2)$. This is the worst case complexity as the algorithm exits if we have a miss at any step. The complexity remains the same with * operators in the query. Consider a query with one *, the query is split into two sub-paths of length $l1_q$ and $l2_q$ that are processed independently, so we have O($l1_q^2$) + O($l2_q^2$) < O(l_q^2).

Store Match

Stores the start and the end point of a match at two arrays,
the start_point and end_point arrays, and returns

Algorithm 1: Store Match

Sub_Match
Input: *pos*: start point of subpath

1: $l_q = pos$
2: **if** Case (a) path is a root path expression **then**
3: **for all** $j = 1$, $i = 1$ to $j = l_b$, $i = l_q$ **do**
4: **if** match(BBF$_j$, a_i) = NO MATCH **then**
5: **RETURN**(NO MATCH)
6: **if** $j + (l_q - i) < l_b$ {check if the query is longer than the remaining levels} **then**
7: **RETURN**(NO MATCH)
8: store MATCH {if this point is reached we had a match for the whole path}
9: **if** Case (b) path is a partial path expression **then**
10: **for all** $j = 1$ to l_b {for every level of the filter check} **do**
11: **if** $j + l_q > l_b$ {if the query is longer than the remaining levels} **then**
12: **RETURN**
13: **for all** $i = 1$ to l_q {for every attribute beginning from the start of the query check by starting from level j} **do**
14: **if** $j + (l_q - i) > l_b$ **then**
15: **RETURN**
16: **if** match(BBF$_j$, a_i) = NO MATCH **then**
17: GOTO 1
18: **else**
19: $j = j + 1$
20: store MATCH {if this point is reached we had a match for the whole path}

Algorithm 2: Subpath Matching

3.3.3 Operations with Multi-Level Bloom Filters

We define two operations between multi-level Bloom filters, namely the construction of the merged filter of two multi-level Bloom filters and the evaluation of the similarity between

Breadth Bloom Filter Lookup
Input: BBF: Breadth Bloom filter with l_b levels, $q = a_1/a_2/\ldots/a_{l_q}$: path expression with length l_q
Output: MATCH/NO MATCH

1: **for all** $i = 1$ to l_q {check for all attributes of the path} **do**
2: **if** $a_i \neq *$ {$a_1, \ldots, a_{l_q}$ in the BBF_0 the * is ignored} **then**
3: **if** no match(BBF_0, a_i) **then**
4: **RETURN**(NO MATCH)
5: $pos = 1$
6: **for all** $k = 1$ to l_q {traverse the path and check for * to split the path} **do**
7: **if** $a_k = *$ **then**
8: $npos = pos + 2$
9: $pos = k - 1$
10: SubMatch(pos)
11: **if** Case(a) AND SubMatch(pos) = NO MATCH **then**
12: **RETURN**(NO MATCH)
13: **for all** stored matches **do**
14: **for all** $i = 1$ to number of paths **do**
15: **if** All start_point(match i) > end_point(match $i + 1$) **then**
16: **RETURN**(MATCH)
17: **RETURN**(NO MATCH)

Algorithm 3: Breadth Bloom Filter Lookup

Depth Bloom Filter Lookup

Input: DBF: Depth Bloom filter with l_b levels, $q = a_1/a_2/\ldots/a_{l_q}$: path expression of length l_q

Output: MATCH/NO MATCH

```
1: for all i = 1 to l_q do
2:   if a_i ≠ * then
3:     if match(DBF_1, a_i) = NO MATCH then
4:       RETURN(NO MATCH)
5: pos = −1
6: for all k = 1 to l_q do
7:   if a_k = * then
8:     npos = pos + 2
9:     pos = k − 1
10: for all i = npos to pos {check all the subpaths of the path of length k − 1} do
11:   for all j = 1 to pos − 1 {until the '*'} do
12:     if i + j ≤ pos {if any of the sub-paths do not exist return failure} then
13:       if match(DBF_j, (a_i/a_{i+1}/.../a_{i+j})) = NO MATCH then
14:         RETURN(NO MATCH)
15:     else
16:       if k = l_q {if there is no path, or for the last part of the path} then
17:         for all i = pos+2 to l_q {after the last * we repeat the above procedure} do
18:           for all j = 1 to l_q − 1 do
19:             if i + j ≤ l_q then
20:               if match(DBF_j, (a_i/a_{i+1}/.../a_{i+j})) = NO MATCH then
21:                 RETURN(NO MATCH)
22: RETURN(MATCH) {if this statement is reached we have a match}
```

Algorithm 4: Depth Bloom Filter Lookup

two filters, which are important for the deployment of the indexes in a distributed setting.

Efficient merging of two multi-level Bloom filters enables us to support incremental updates in a p2p system as for each new document that enters the system its multi-level Bloom filter only needs to be merged with the filter that summarizes the current document collection. To calculate the merged multi-level Bloom filter of a set of multi-level Bloom filters we take the bitwise OR for each of their levels. In particular, the merged filter, Sum_BBF, of two Breadth Bloom filters BBF^x and BBF^y with l_b levels is a Breadth Bloom Sum_BBF = Sum_BBF_0, Sum_BBF_1,..., $Sum\_BBF_{l_b}$ with l_b levels where: Sum_BBF_i = $BBF^x{}_i$ BOR $BBF^y{}_i$, 0≤i≤l and BOR stands for bitwise OR. Similarly, we define merging for Depth Bloom filters.

The evaluation of the similarity is required so as to exploit the idea of clustering in our index structures. That is, in accordance to clustering as used in databases, we argue that by summarizing similar documents in the same filter we improve the performance of our structures since similar documents match similar queries. Instead of checking the similarity of the documents themselves, we rely on the similarity of their filters. This is more cost effective, since a filter for a set of documents is much smaller than the documents. Furthermore, the filter comparison operation is more efficient than a direct comparison between two sets of documents.

Let B be a simple Bloom filter of size m. We shall use the notation B[i], $1 \leq i \leq m$ to denote the ith bit of the filter. Our similarity measure is based on the *Manhattan distance* metric. Let two filters B and C of size m, their Manhattan (or Hamming) distance, dMan(B, C), is defined as dMan(B, C) = |B[1] - C[1]| + |B[2] - C[2]| + ... |B[m] - C[m]|. We define the similarity, *similarity*(B, C), of two simple Bloom filters, B and C of size m as follows: similarity(B, C) = m - dMan(B, C). The larger their similarity, the more similar the filters are. Fig. 3.3 shows an example for two filters of size 8. In the case of multi-level Bloom filters, we take the sum of the similarities of every pair of corresponding levels. To compute the similarity of two filters, we simply take the *equivalence* (exclusive NOR) of the vectors that correspond to each level of the filter.

Fig. 3.4 illustrates an experiment that confirms the validity of the measure. The measure is valid if when we increase the similarity between two documents (i.e. by repeating an increasing number of element names in both documents); the similarity measure also increases analogously. We used different percentage of element name repetition between documents and measured their similarity. The similarity measure increased linearly with the increase of the repetition between the documents. The same holds for the similarity between multi-level Bloom filters, although in this case, the measure depends on the structure of the documents as well.

3.3.4 False Positives

The probability of false positives depends on the number k of hash functions we use, the number N of elements we index, and the size m of the Bloom filter. The formula that

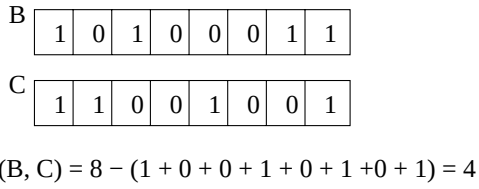


Figure 3.3: Bloom filter similarity.

gives this probability for Simple Bloom filters is [15]:

$$P = (1 - e^{-kN/m})^k \tag{3.1}$$

For the computation of the false positive probability for a multi-level Bloom filter we also have to consider the structure of the XML document, as the number of elements inserted in every level of the filter depends on the document structure, in particular, its depth and out-degrees. Let dgr be the out-degree of the XML tree and l_b the number of levels in the filter. In the rest of the analysis for the false positives, we assume that the filter has the same number of levels as the depth of the tree, although the filter levels can be limited so as to reduce storage requirements.

We denote as N_i the number of elements inserted in each level i of the multi-level Bloom filter, and we have for Breadth Bloom filters: $N_i = dgr^{i-1}$, while for Depth Bloom filters $N_i = \sum_{j=i+1}^{l_b} dgr^j$ assuming that every path expression is inserted once and root paths are not hashed under different notation.

The respective sizes of each level of the filters are denoted as m_i. While we use fixed size for all the levels of the Depth Bloom filter (size equal to m), we may vary the size in Breadth Bloom filters to correspond with the number of inserted elements that are also increased exponentially at each level. Thus, by selecting a minimum size m, the size of each level of the Breadth Bloom filter is: $m_i = dgr^{i-1}m$.

By simple substitutions of the corresponding filter size and number of inserted elements in Eq. (3.1), we derive the false positive probability for a single lookup in any level of the multi-level Bloom filters ($P_{ls}(i)$).

We compute the false positive probability for a path q of length l_q by combining the probabilities for a single lookup. We make the independence assumption for the occurrences of false positives for the different subpaths and between all the levels in the filters. Therefore, to compute the false positive probability (P_{tot}) for q we take the product of the false positive probabilities for each of the single lookups the query evaluation algorithm makes.

We discern between three cases regarding q and the document d which is summarized by the filter: (Case I) none of the elements of q appear in d, (Case II) all the elements of q appear in d but they do not form any subpath that appears in q, and (Case III) r elements of q appear in d.

Simple Bloom Filters. If we assume that we use Simple Bloom filters to summarize our documents, then the false positive probability for (Case I) is $P_{tot} = P^{l_q}$ as it would

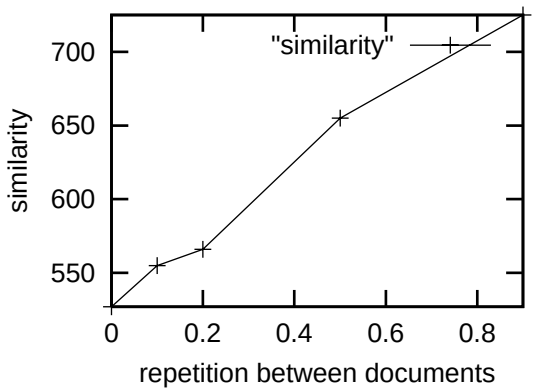


Figure 3.4: Validity of similarity measure.

require a false positive for each separate lookup to falsely identify the query as a match. For (Case II), the false positive probability is equal to $P_{tot} = 1$ as Simple Bloom filters do not maintain the relationships between the inserted elements. Finally, for (Case III), the false positive is equal to $P_{tot} = P^{l_q - r}$.

Breadth Bloom Filters. We exclude the BBF_0 from our analysis as this performs as simple Bloom filter and we are interested in studying the influence of the multiple checks at the various levels of the multi-level Bloom filter. For (Case I), a false positive occurs when all l_q elements in the query appear at some level of the BBF. Each element is checked against the BBF $l_b - l_q + 1$ times, starting from level $i = 1$ to $l_b - l_q + 1$. The checks do not start from lower levels, since the path length would then be greater than the remaining levels of the filter. Each check consists of l_q separate lookups in the corresponding levels. To have a false positive for the whole path, it suffices to have l_q false positives in one of the $l_b - l_q + 1$ checks the algorithm executes. Thus, we have:

$$P_{tot} = \sum_{i=1}^{l_b - l_q + 1} \prod_{j=1}^{l_q} P_{ls}(j) \tag{3.2}$$

For (Case II) and (Case III), we have to observe that some elements may fall into the correct level at each different check. If r of the elements in the path query are at the correct level then the false positive probability is for both cases:

$$P_{tot} = \sum_{i=1}^{l_b - l_q + 1} \prod_{j=1}^{l_q - r} f(j), \text{for } 1 \leq r \leq l_b - l_q + 1, \tag{3.3}$$

where $f(j)$ is a function that is equal to 1, if the j-th element of q is one of the r elements and $P_{ls}(j)$ otherwise.

Depth Bloom Filters. Every query of length l_q has $(l_q - i + 1)$ sub-paths of length i, and thus $(l_q - i + 1)$ checks are performed at every level i of the filter. If none of the elements of the query exists in the filter (Case I), to conjecture that the path exists we must have a false positive for every check at each level, so the false positive probability

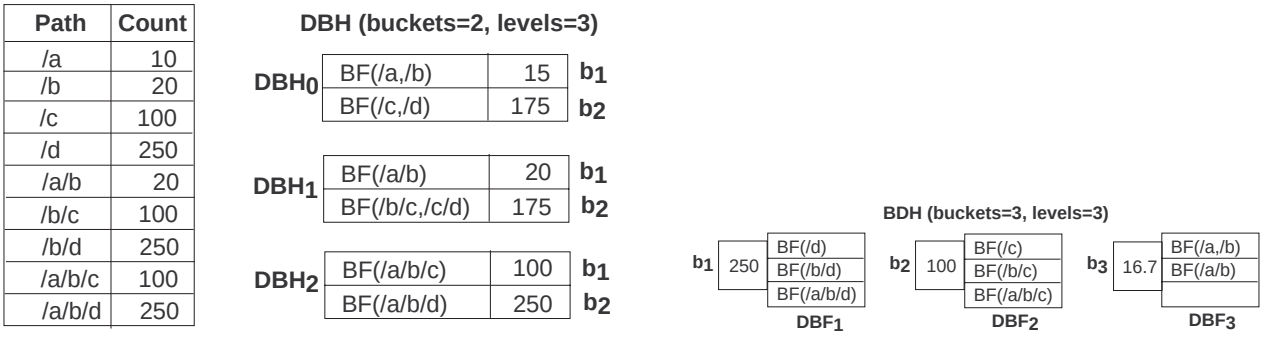


Figure 3.5: (left) Full path count table, (center) the corresponding DBH and (right) BDH.

is:

$$P_{tot} = \prod_{i=1}^{l_q} P_{ls}^{l_q - i + 1}(i) \tag{3.4}$$

In (Case II), we have to discern how many many correct sub-paths are formed from these elements. If no correct sub-paths are formed (Case II), then only the q checks at the first level (DBF_0), that is, checks for sub-paths of length 0, will correctly return a success. Thus, to deduce that the path does exist in the filter we would have a false positive in all other checks at levels 2 to l_q, that is:

$$P_{tot} = \prod_{i=2}^{l_q} P_{ls}^{l_q - i + 1}(i) \tag{3.5}$$

For (Case III), we have at each level i, $r - i + 1$ correct subpaths and $q - r$ incorrect and the resulting false positive probability is:

$$P_{tot} = \prod_{i=1}^{r} P_{ls}^{l_q - r}(i) \prod_{i=r+1}^{l_q} P_{ls}^{l_q - i + 1}(i), \tag{3.6}$$

where the first term of the product corresponds to the falsely $(l_q - r)$ recognized subpaths and the second term of the product corresponds to sub-paths with length greater than r, that also do not exist in the filter, as we assumed that the correct subpath is of length r.
Additional False Positives. Using BBFs, a new kind of false positive appears. Consider the tree of Fig. 2.2 and the partial path query: camera/color. We have a match for camera at BBF_2 and for color at BBF_3; thus we falsely deduce that the path exists. The probability for such a false positive is strongly dependent on the degree of the tree. For DBFs we have a type of false positive that refers to queries that contain the * operator. Consider the paths: a/b/c/d/ and m/n. For the query: a/b/*/m/n, we split it to: a/b and m/n. Both of these paths belong to the filter so the filter would indicate a false match.

3.4 Index Structures with Selectivity Estimations

We combine the multi-level Bloom filters with histograms and derive a new index structure *Multi-Level Bloom Histogram* that supports queries that require selectivity estimations. We select to extend the Depth Bloom filters between the two variations of multi-level Bloom filters since it does not incur the additional type of false positives that are inherent in the Breadth Bloom filter.

ConstructDBH
Input: DBH: DBH with l_b levels with b buckets each, d: XML document
Output: DBH: Updated DBH that includes d

1: $EstSet_i$ = NULL //Set of selectivity estimations for each level of the DBH
2: $EstSet$ = NULL //Set of average selectivity estimations
3: Extract from d all possible subpaths p with length 0 to l_b
4: $freq_p$ = number of occurrences for each subpath p in d
5: Split the set of subpaths p into groups, $group_i$, according to their length i
6: **for all** $group_i$ **do**
7: Build BH with b buckets
8: Sort the BHs according to their path lengths
9: **RETURN** DBH

Algorithm 5: DBH Construction

Instead of using a simple Bloom filter for each level i, we use a Bloom Histogram constructed by taking as input all paths of length i. The resulting *Depth Bloom Histogram* (DBH) with l_b levels and b buckets for an XML tree with L levels is a set of Bloom Histograms $\{\text{DBH}_0, \text{DBH}_1, \text{DBH}_2, \ldots, \text{DBH}_{l_b-1}\}$, $l_b \leq L$, where each DBH_i is a Bloom Histogram with b buckets that includes all paths of length i. An example is shown in Fig. 3.5, where Fig. 3.5(left) depicts a full path count table (PCT) that reports the frequency of each distinct path in the tree and Fig. 3.5(center) the corresponding DBH.

In addition to the DBH, we also consider an alternative way of combining histograms and Depth Bloom Filters (DBFs) (Fig. 3.5(right)). Instead of using a Bloom Histogram for each of the levels of a DBF, we use a DBF for each bucket of the Bloom Histogram. In this case, we first assign all paths of the input XML document into buckets according to their frequencies. Then, we build a DBF for each bucket by splitting the paths in the bucket based on their length. Assume that the paths of the XML tree are split into b sets of paths, $path_i$, $1 \leq i \leq b$. The *Bloom Depth Histogram* (BDH) with l_b levels and b buckets is a two-column table $BDH(DBF_i, val_i)$ where each DBF_i is a DBF with l_b levels of the paths with frequency val_i.

For supporting local updates, as in [135], each site maintains an auxiliary full path count table on which it first applies the update and then reconstructs the DBH or BDH taking it as input.

In the following, we refer to the DBH structure since it is rather straightforward how all algorithms can be adapted for the BDH one.

3.4.1 Selectivity Estimation

Assume a DBH with l_b levels and b buckets and a query q of length l_q. To estimate the selectivity of q using the DBH, q is split to all possible subpaths of length 0 (single elements) up to length $l_m - 1$, where $l_m = minimum\{l_b, l_q\}$. Each subpath of length i is then hashed and checked against the b Bloom filters of the corresponding DBH, that is of DBH_i. If we have a match in more than one filter, the average of the values in the corresponding buckets is returned as the selectivity estimation for the corresponding path. After all subpaths are evaluated, a set of selectivity estimations, one for each of the subpaths, is retrieved. For the query to have a match in the data, it is required that all subpaths appear in the filter. Thus, we take as the selectivity estimation for q the minimum of the set of the selectivity values that are retrieved.

When q contains either the wildcard or the "//" operator, the query is split at its position and the resulting subqueries are evaluated separately by using the procedure described above. The selectivity estimation for q is given by the minimum estimation returned for any of the subqueries. The same applies for queries containing branching.

Complexity. Assume a DBH with l_b levels and b buckets. For a query q with length l_q, firstly, l_q subpaths of length 0 are matched against the b Bloom filters of DBH_0, then $l_q - 1$ subpaths of length 1 are matched against the b Bloom filters of DBH_1 and so on, until we match 1 path of length l_q or the length of the subpaths in question is no longer contained in the DBH ($l_q > l_b$). Thus, we have: $b*l_q + b*(l_q-1) + b*(l_q-2) + \cdots + b*1 = O(b*l_q*l_m)$ lookups for each query, where $l_m = minimum\{l_b, l_q\}$.

3.4.2 Operations with Multi-Level Bloom Histograms

EvaluateSim
Input: H, clH: DBHs with l_b levels and b buckets
Output: sim: similarity score

1: $sim = 0$
2: **for** $i = 0$ to l_b **do**
3: $\quad mBF_i = clH.DBH_i(BF_1)$ BOR $clH.DBH_i(BF_2)$ BOR ... BOR $clH.DBH_i(BF_b)$
4: $\quad sim_i = 0$
5: $\quad$ **for** $j = 0$ to b **do**
6: $\quad\quad sim_i = sim_i + H.DBH_i(val_j) * Jaccard(H.DBH_i(BF_j), mBF_i)$
7: $\quad sim = sim + sim_i$
8: **RETURN** sim

Algorithm 6: Similarity Evaluation

The similarity between two DBHs is computed using the *EvaluateSim* algorithm (Alg. 6), which is based on the Jaccard distance between the corresponding simple Bloom filters of the two DBHs. We describe next how the merging of two DBHs is achieved.

Let the DBHs H and H' with l_b levels and b buckets be the indexes for document d and d' respectively. We want to aggregate (merge) the two DBHs to create a new DBH H''

MergeDBH
Input: H, H': DBHs to be merged
Output: H'': merged DBH
Require: H, H': DBHs with l_b levels and b buckets
Ensure: H'': merged DBH
1: **for** $i = 0$ to l_b **do**
2: **for** $j = 0$ to b **do**
3: Find the two buckets j' and j'' in level i of H and H' with the closest frequency values that have not been selected yet
4: $H''.DBH_i(val_j) = (H.DBH_i(val_{j'}) + H.DBH_i(val_{j''})/2$
5: $H''.DBH_i(BF_j) = H.DBH_i(BF_{j'})$ BOR $H.DBH_i(BF_{j''})$
6: **return** H''

Algorithm 7: DBH Merging Procedure

with l_b levels and b buckets that summarizes both documents d and d'. A nice property of Bloom-based structures is that to construct H'', there is no need to have access to the actual documents d and d'. Instead, there is a simple procedure to construct H'' based on H and H'. The two DBHs H and H' are merged per level, that is level i of H is merged with level i of H' to create level i of H''. For each level i, we merge one bucket of H with one bucket of H'. We start by merging the two buckets that have the most similar frequencies, that is, the ones with the most similar value at the *val* column, then the ones with the next most similar frequencies and so on, until all b buckets of level i are merged. The frequency (i.e., *val* column) of the resulting merged bucket is set equal to the average of the frequencies (i.e., *val* columns) of the buckets being merged. The Bloom filter (i.e., BF column) of the merged bucket is computed by taking the bitwise OR of the Bloom filters (i.e., BF columns) of the two buckets being merged. Clearly, the same merge procedure may be used to merge DBHs that summarize more than one document.

3.4.3 False Positives and Estimation Error

Based on the false positive analysis of the DBFs, we provide a corresponding analysis for the DBH. Assume that the probability that a path in $Tree$ has frequency that belongs to bucket j ($1 \leq j \leq b$) is $P_f(j)$ with $\sum_{j=1}^{b} P_f(j) = 1$. Then, the number of elements inserted at each bucket j of each level i of the filter is at most: $P_f(j) * N_i$. Thus, the false positive probability for a single look-up in a filter corresponding to a bucket j of the i level of DBH is given by: $P_l(i,j) \leq (1 - e^{-m*P_f(j)*n_i/s})^m$. The query evaluation algorithm performs b checks for each subpath of length i. Thus, the false positive probability for a subpath of length i is: $P_{sub}(i) = 1 - \prod_{j=1}^{b} (1 - P_l(i,j))$, since for not having a false positive, all the b checks must not return one.

To estimate the corresponding estimation error for any subpath lookup we rely on [135]. Let $1 < val_j \leq M$, $1 \leq j \leq b$ and let V_* denote the actual number of appearances of a path in the XML tree. The absolute error for a returned value val_j is then given

by $e_j = |val_j - V_*|$. For $V_* > 0$, the estimated error for a subpath of length i that was inserted in bucket b_* of DBH_i ($1 \leq b_* \leq b$) is bounded by:

$$E[e] < \prod_{j=1, j\neq b_*}^{b} (1 - P_l(i,j)) E[|val_{b_*} - V_*|] + (1 - \prod_{j=1, j\neq b_*}^{b} (1 - P_l(i,j))) \; M$$

The first term refers to the error due to the histogram when the correct bucket b_* is located, while the second term is the error for falsely reporting the existence of the path in multiple buckets. When $V_* = 0$, the error is bounded by: $E[e] < P_{sub}(i)M$.

Let us consider the simplest case (Case I) where none of the subpaths in the query belongs to documents in the DBH. Then, for a false positive to occur, at each level i of the filter, we must have a false positive match against all $l_q - i + 1$ subpaths of the query, in at least one of the b buckets it is checked against. If we consider these probabilities of false matches to be independent, then this probability is: $P(i,q) = \prod_{q' \in q} P_{sub}(i)$, where q' are all subpaths of length i extracted from q. Taking into account all levels of the filter, the overall false positive probability is: $P(q) = \prod_{i=1}^{l_m} P(i,q)$, where $l_m = minimum\{l_b, l_q\}$ and l_b the number of levels of the DBH. Let us now also consider the case in which $r+1$ elements $0 \leq r \leq l_q$ exist in the document and form a correct subpath of length r (Case II - Case III). Then, at each level i of the filter, the correct subpaths that exist are at least: $r - i + 1$ and the ones that do not exist are at most $l_q - r$. To compute the false positive, $P(i,q)$ for $i \leq l_q - r$, we should consider only these subpaths.

3.5 Experimental Evaluation

We present an experimental evaluation of the performance of the two new structures with respect to the false positives and estimation errors they incur and study the influence of the various tuning parameters. We first present a comparison of the DBF and BBF to a simple Bloom filter to show how our enhancement is appropriate for handling boolean path queries. Then, we also present a comparison of the Bloom Histogram structure with the two variations of the multi-level Bloom histogram to show how the multiple checks imposed by adding levels to the structure improves its performance.

3.5.1 Multi-Level Bloom Filter Evaluation

We implemented both the BBF and DBF data structures in C. We also implemented a Simple Bloom filter (SBF) that just hashes all elements. We limited the inserted path expressions in Depth Bloom to be at most of length 3, that is, the Depth Bloom only has three levels. Also, we excluded the bloom on top that is only used for performance reasons since it requires more space and it would deteriorate Breadth's performance for a given space overhead. For the hash functions, we used MD5 [89]: a cryptographic message digest algorithm that hashes arbitrarily length strings to 128 bits. The k hash functions are built by first calculating the MD5 signature of the input string, which yields

Table 3.1: Input parameters for the evaluation of the multi-level Bloom filters

Parameter	Default Value	Range
# of XML documents	200	-
Total size of filters	78000 bits	30000-150000 bits
# of hash functions	4	-
# of queries	100	-
# of elements per document	50	10-150
# of levels per document	4/6	2-6
Length of query	3	2-6

128 bits, and then taking k groups of 128/k bits from it. For the generation of the XML documents, we used the Niagara generator [101] to generate tree-structured XML documents of arbitrary complexity. The repetition on the names of the elements was set to 0 between the elements of a single document as well as between all the documents. Queries were generated by producing arbitrary regular paths, with 90% elements from the documents and 10% random ones. All queries were partial paths and the probability of the containment operator at each query was set to 0.05. Table 3.1 summarizes our parameters. We have chosen as our metric the percentage of false positives, since the number of nodes that will process an irrelevant query strongly depends on it.

Varying the Filter Size. We examine the influence of the size of the filter with respect to false positives. The document's structure is fixed with 50 elements and 4 levels. The queries are of length 3. The size of the filters varies from 30000 bits to 150000 bits. The lower limit was chosen from the formula that gives the number of hash functions $k$that minimize the false positives probability for a given size m and n inserted elements for a Simple Bloom filter: $k = (m/N)ln2$. We solved the equation for m keeping the other parameters fixed. The goal of this experiment is to show that even if we increase the size of the filter significantly, Simple Bloom filters cannot correctly recognize path expressions.

The results show that both Breadth and Depth Bloom filters outperform Simple Bloom filters even for only 30000 bits (Fig. 3.6(left)). In addition, in contrast with Simple Bloom filters where the increase in the size results in no improvement in their performance, the multi-level structures exploit the extra space. Simple Bloom filters are only able to recognize as misses paths that contain elements that do not exist in the documents. Breadth Bloom filters perform very well even for 30000 bits with an almost constant 6% of false positives, while Depth Bloom filters require more space since the number of the elements inserted is much larger than that of Breadth and Simple Bloom filters. However, when the size increases sufficiently, Depth Bloom filters outperform even Breadth Bloom filters and produce no false positives.

Using the result of the first experiment, we choose as the default size of the filters for the rest of the experiments, a size of 78000 bits, where both our structures showed

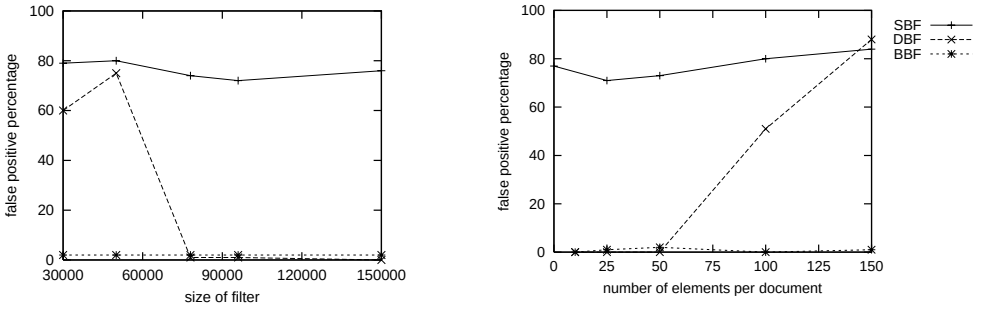


Figure 3.6: Varying (left) the filter size and (right) the number of elements per document

reasonable results. For 200 documents of 50 elements, this represents 2% of the space that the documents themselves require. This makes Bloom filters a very attractive summary to be used in a peer-to-peer context.

Varying the Number of Elements per Document. We compare the filters with respect to the number of elements per document. The size of the filter is fixed to 78000 bits, and the documents have 4 levels. Queries have length 3 and the elements vary from 10 to 150.

Once again, Simple Bloom is only able to recognize path expressions with elements that do not exist in the document (Fig. 3.6(right)). Even for 10 elements where the filter is very sparse, Simple Bloom filters have no means to recognize hierarchies. When the filter becomes denser as the elements inserted are increased to 150, Simple Bloom filters fail to recognize even some of these expressions. Breadth Bloom filters show the best overall performance with an almost constant percentage of 1 to 2% of false positives. Depth Bloom filters require more space and their performance rapidly decreases as the number of inserted elements increases, and for 150 elements they become worse than Simple Bloom filters because the filters become overloaded (most bits are set to 1).

Varying the Number of Document Levels. In this experiment, we compare the three approaches with respect to the number of levels of the documents. The size of the filter is fixed to 78000 bits, and the documents have 50 elements. The levels vary from 2 to 6. The queries are of length 3, except for the documents with 2 levels where we conduct the experiment with queries of length 2.

The behavior of Simple Bloom filters is independent of the number of levels of the documents, since they just hash all their elements irrespectively of the level that they belong to (Fig. 3.7(left)). So they only recognize path expressions with elements not in the documents that account for about 30% of the given query workload. Both Breadth and Depth Bloom filters outperform them with a false positive percentage below 7%. Breadth Bloom filters perform better for 4 to 5 levels. This is because the elements are more evenly allocated to the levels of the filter, while for fewer levels the filter has also less levels and it becomes overloaded.

Also false positives of a new kind appear for Breadth Bloom filters. If we had a tree that had the following paths: /a/b/c and /a/f/l then Breadth Bloom would falsely

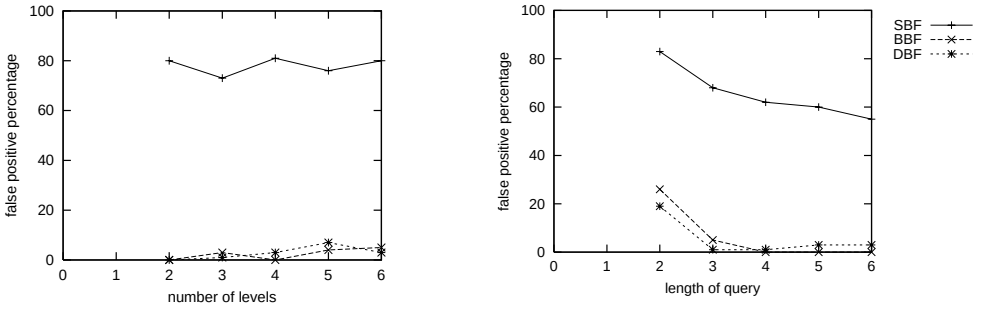


Figure 3.7: Varying (left) the number of levels and (right) the length of the queries

recognize as correct the following path: /a/b/l. Depth Bloom filters do not have this problem as they would check for all possible sub-paths /a/b/l, /a/b, /b/l, and would find a miss for the last one. That is why they perform very well for documents with few levels. Their performance decreases for more levels but remains almost constant since we insert only sub-paths up to length 3, while Breadth Bloom filters deteriorate further for 6 levels.

Varying the Length of the Queries. The parameter examined in this experiment is the length of the queries. The structure of the document is fixed, with 4 levels and 50 elements, for queries of length 2 to 4 and 6 levels for queries with length 5 and 6. The size of the filter is also fixed to 78000 bits.

Once again, both multi-level Bloom filters outperform Simple ones (Fig. 3.7(right)). The Simple Bloom filters performance slightly improves as the query length increases but this is only because the probability for an element that does not exist in the documents increases. Both structures perform better for large path expressions, since if one level is sparse enough it is sufficient to filter out irrelevant queries. Depth Bloom filters show a slight decrease in performance for a length of 5 and 6 since for documents with 6 levels the number of inserted elements increases and the filter becomes denser.

The last two experiments also show that though we limited the number of filters to three for the Depth bloom, it is still able to show a very good performance although all the elements (all possible sub-paths with length larger than 3) are not inserted. The checks of all possible sub-paths up to length 3 are able to recognize most of the misses, so we conclude that we can limit the number of levels of the filter without a significant loss in performance if we have limited space.

Types of Workload. In most of our experiments, Breadth Bloom filters seem to outperform Depth Bloom filters for a fraction of the space the second require. The reason for using Depth Bloom filters became evident in the experiment regarding the influence of the levels in the document, in which the Breadth Bloom filters fail to recognize a new kind of false positives. To clarify this, in this last experiment, we created a workload with queries consisting of such path expressions, that is, a workload that favors Depth Bloom filters. The percentage of these queries varied from 0% to 100% of the total workload. The size of the filter is fixed to 78000 bits; the documents have 4 levels and 50 elements. We included the Simple Bloom in the experiment only for completeness.

Breadth Bloom filters fail to recognize these misses and their percentage of false pos-

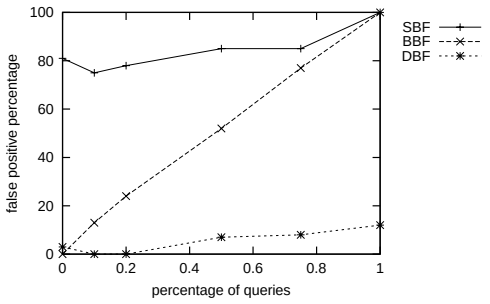


Figure 3.8: Influence of the type of workload

Table 3.2: Input parameters for the evaluation of the multi-level Bloom histograms

Parameter	Default Value	Range
# of documents	500	50 - 500
# of data categories	8	-
Size of document	2KB-120KB	-
DBH size	320KB	5 - 320KB
DBH depth	3	-
DBH hash functions	4	-

itives increases linearly to the percentage of these queries in the workload (Fig. 3.8). However, Depth Bloom filters have no problem of recognizing this kind of false positives and show much better results. The slight increase of the percentage of false positives in Simple and Depth Bloom filters is because as the number of these special queries increases, the number of queries with elements that do not exist in the document decreases. When all queries are of this special form (thus, there are no queries with elements that do not exist in the documents), the Simple Bloom filters has a percentage of 100% false positives and Depth of about 10%. Thus, we can conclude that one may consider spending more space in order to use Depth Bloom filters, so as to avoid these false positives, while, when space is the key issue, Breadth Bloom filters are a more reasonable choice.

3.5.2 Multi-Level Bloom Histogram Evaluation

In this set of experiments, we evaluate the performance of our DBH and BDH structures compared to the performance of a Bloom Histogram in terms of storage efficiency and estimation accuracy. We use real data sets from the Niagara Project [100] that belong to

Table 3.3: Data sets for the evaluation of the multi-level Bloom histograms

Set	Sigmod Record '03	bibliographical data	movies	actors	linux docs	company personnel	nasa	club members
Doc Size	100K	5K	2.2K	2-10K	20-120K	2K	2-14K	8K

eight predefined categories as shown in Table 3.3. There are two pairs of categories that share structural similarities, namely, the Sigmod Record data with the bibliographical data and the actors with the movies data, while there is little to none overlap among the rest. For query generation, we use the zipf distribution to select paths from the documents.

The queries have a minimum length of 4 location steps. To tune the number of buckets and their boundaries in the multi-Level Bloom Histograms, we apply the techniques presented in [135]. We use 4 buckets per level. We configure the Depth Bloom filters parameters (i.e, the BF size and the number of hash functions) based on previous results and limit the levels to 3 since we showed that 3 levels are adequate for a false positive ratio below 5%. Table 3.2 summarizes our parameters.

Furthermore, we assume both randomly selected documents in our index structure and documents selected from a single category, that is, documents that are structurally similar, so as to show that grouping similar documents together improves the performance of our structures.

Clustered Documents. We compare the false positive ratio and the estimation error of the DBH and BDH indexes with those of a Bloom Histogram (BH) index of the same size. The space that each index occupies is expressed as a percentage of the space occupied by a full path count table (PCT) for the same documents (this is between 2% to 10% of the PCT size). All documents summarized by each of the indexes belong to one of our categories, i.e. they are structurally similar and therefore they are expected to satisfy similar queries. Figure 3.9(right) shows that both DBH and BDH improve the error of the BH up to 30% for the same space overhead.

Randomly Selected Documents. We repeat the same experiment using documents selected uniformly at random from all categories in the system. For comparison, we use the same sizes we used for the summary when documents of a single category were selected. However, note that these do not represent the same percentage of the path count table that would be constructed in this case, since this path count table is larger due to the variety in the path expressions. As Fig. 3.9 and Fig. 3.10 show, the comparative performance of the three indexes is similar with that in our first experiment. However, all three indexes perform worse with random documents than with similar ones. For instance, the simple Bloom Histogram does not achieve a false positive ratio lower than 40% (Fig. 3.10(left)), while for the clustered index, it achieves a ratio below 20%.

Comparison of DBH and BDH. The two enhanced indexes (DBH and BDH) perform similarly for both cases. In general, their performance depends on the data characteristics. The first structure splits paths primarily according to their length, while the second one according to their frequencies. Therefore, for data with paths with equal frequencies, BDH is expected to perform worse than DBH. For the data sets in our evaluation, their performance is comparable, and we use the DBH as the index structure in the rest of our experiments.

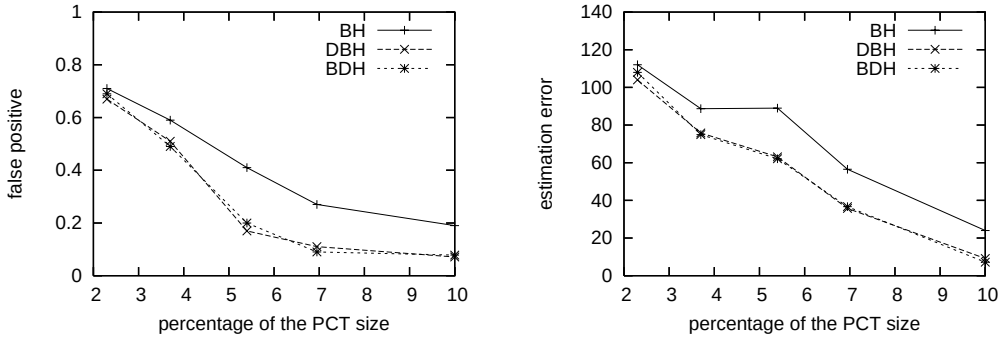


Figure 3.9: (left) False positive ratio and (right) estimation error for clustered documents.

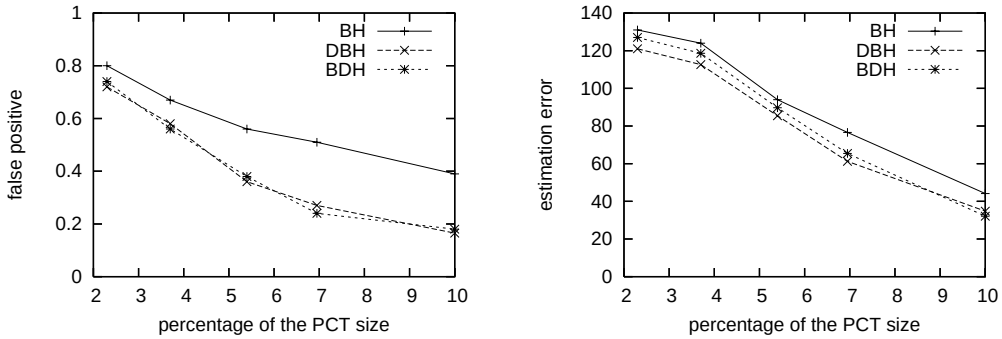


Figure 3.10: (left) False positive ratio and (right) estimation error for randomly selected documents.

3.6 Summary

In this chapter, we introduced the multi-level Bloom filters and the multi-level Bloom histograms, two hash-based index structures designed for summarizing XML documents in distributed environments such as p2p systems. We described algorithms for their construction, maintenance and query evaluation and presented an analysis on the false positives and estimation errors they incur. Our experimental evaluation showed that both structures summarize large volumes of data while requiring limited storage overhead compared to maintaining the actual data and in the same time have satisfying performance, i.e., low false positive ratios for both and a low estimation error for the multi-level Bloom histograms. Finally, we showed that summarizing similar documents together, further improves the performance of our structures.

In the rest of this thesis, we further explore the deployment of the multi-level Bloom histograms in distributed systems for building distributed indexes as a means to improve efficiency with respect to the communication and processing costs involved in query evaluation.

The multi-level Bloom filters were introduced and used as routing indexes for p2p systems in [67], [68], [69], [70], and the multi-level Bloom histograms were introduced and deployed in distributed query evaluation scenarios in [73], [75].

Chapter 4

Structural Relaxation Over Distributed XML Collections

In this chapter, we use the proposed multi-level Bloom histogram to build a distributed clustered index. Furthermore, since we assume that our data is heterogeneous and schema knowledge is not usually available, we present a set of algorithms that operating on top of the multi-level Bloom histograms support approximate query processing by structurally relaxing the original query and ranking the results according to their relevance to it. In this setting, we exploit this clustered index in two popular query evaluation scenarios, a top-K and an incremental, pay-as-you-go, query evaluation, so as to improve their performance with respect to the communication and processing cost required.

We first present the problem of top-K query evaluation and the overall architecture of our system (Section 4.1) and introduce the algorithms we use for structural query relaxation (Section 4.2). Then, we describe the clustered index and how it is constructed as well as an alternative organization that constructs an hierarchical clustered index (Section 4.3). We then exploit the clustered index and the hierarchical clustered index and describe how they can be used for the two query evaluation scenarios (Section 4.4). Finally, we show through our experimental evaluation how the distributed clustered index improves the performance of the query evaluation scenarios (Section 4.5) and conclude with a short summary of this chapter contributions (Section 4.6).

4.1 Problem Definition

We assume a collection of heterogeneous XML documents $\mathcal{D}$ distributed over a number of N different sites or peers. Users issue queries over the distributed collection of documents and are interested in retrieving a sufficient number of results (K) that match their query. We assume queries in XPath$^{\{/,//,*\}}$. Since the number of documents available in the collection that match any given query exactly may not be enough to satisfy the user requirement ($|result(q, \mathcal{D})| < K$), we support approximate matching, returning to the user the top-K most relevant results. To support approximate matching, the queries are relaxed and then evaluated to produce the top-K results which are ranked based on their relevance to the original query.

A problem with distributed processing is that a site cannot determine how much to relax a query without contacting the other sites. A straightforward solution would be to forward the query to all sites. Then, each site would relax the query independently to produce a list with the locally best K results and send it to the query origin, which would produce the final list of top-K results. However, the number of sites involved and thus, the communication overhead would be very large. Furthermore, some of the sites may not support approximate query processing capabilities.

To achieve scalability, instead of dealing with actual data and contacting all sites when processing each query, we summarize the documents using the multi-level Bloom histograms. The summaries of the documents are merged and these merged summaries are assigned and maintained by special sites that play the role of *superpeers*. Furthermore, the summaries are not assigned randomly to superpeers, but summaries of documents with similar structural properties are grouped together, thus forming *clusters*. Each superpeer maintains a *cluster summary* ($clsum$), which is constructed by aggregating the summaries of the documents assigned to it. This set of M cluster summaries forms our clustered index which is built over the distributed document collection and is distributed among the superpeers as shown in Fig. 4.1.

Queries are then handled by the superpeers where they are evaluated, i.e., relaxed against the cluster summaries. The results attained by all clusters are then combined to produce the final top-K list. The sites maintaining the documents in the list are then contacted to retrieve the actual data.

We also consider an alternative evaluation scenario, in which the superpeers are contacted gradually. This incremental query evaluation based on a *pay-as-you-go* premise, enables the user to gradually pay more in communication and processing cost as well as response time while receiving incrementally the results to its query.

Most previous approaches to XPath relaxation are centralized [88, 8, 9, 128, 106, 105, 6]. There has been some previous work on the distributed evaluation of top-K queries, but they mostly assume keyword-based data models and the data structures used as summaries and the algorithms deployed for relaxation are not appropriate for XML data.

In particular, the distributed evaluation of top-K queries has been addressed in the context of distributed web search engines, such as Minerva [14] and Chora [57] that

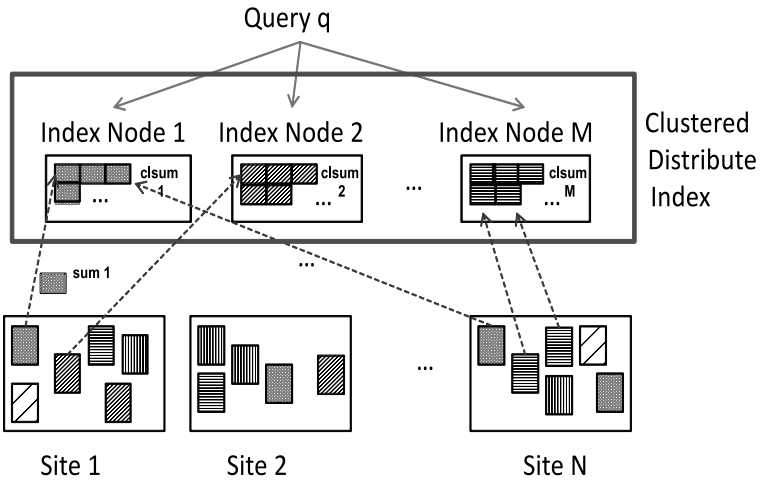


Figure 4.1: A distributed clustered index. The clustered index is built on top of N sites that maintain the distributed XML document collections and is partitioned to M ($N >> M$) index nodes (superpeeers). Each index node maintains a cluster summary $clsum$.

are layered upon a distributed hash table (DHT). In Minerva, the DHT holds compact, aggregated information about the local indexes of the peers. Every peer is responsible for a randomized subset of the global directory, unlike our approach where the index is clustered. Queries are forwarded to a chosen, small set of peers. A routing algorithm (IQN) [90] exploits statistical synopses published in the DHT to iteratively choose the best peers for forwarding the query. In CHORA, a peer's web viewing history is exploited to recommend useful web sites to queries. KLEE [91] extends [22] and allows a peer to trade-off result quality and expected performance. The system assumes that index lists for text terms are distributed across peers. Query evaluation exploits statistical information about the indexed data.

4.2 Selectivity-based Structural Relaxation

There has been a lot of work on defining non exact-match semantics for XML [8], [9], [88]. Generally, approximate query processing and ranking for XML may include other characteristics of the documents such as values or keywords. Since the heterogeneity and lack of schema knowledge, which is inherent in large-scale distributed systems, mostly concerns the structure of the documents, we focus on the problem of structural query relaxation.

4.2.1 Distance Measure and Structural Transformations

To acquire approximate results for a query, we generalize the original query, by applying a number of structural transformations to it. Our transformations are in the spirit of [8], but simplified for linear path expressions.

Definition 4.1 (Structural Transformations)**.** Let P be the set of path expressions. A structural transformation T is a function $T : P \rightarrow P$ that maps a path expression $p = //p_1/p_2/.../p_n$ into a new path expression $T(p)$. In particular, we define the following three structural transformations:

- truncation of the last element of p: $Trunc(p) = //p_1/p_2/.../p_{n-1}$
- replacement of an element tag at position i of p with the wildcard operator: $RepTag(p, i) = //p_1/p_2/.../p_{i-1}/*/p_{i+1}.../p_n$.
- replacement of a "/" axis in position i with the "//" axis: $RepAxis(p, i) = //p_1/p_2/...p_{i-1}//p_i/.../p_n$.

We define next a distance measure for ranking the results. Such a measure should not assume any knowledge about the global data distribution, since this is not realistic in a distributed setting. To this end, we rely on a variation of the edit-distance that counts the number of mismatching element tags between two path expressions of the same length. To compare two path expressions with different lengths, we define a function $ap^*(p, x, j)$, which given a path expression p of length l, and a number x ($x \geq 0$), appends x path steps at the $j + 1$ position in p with the wildcard as their element tag. That is: $ap^*(p, x, j) = //p_1/p_2/\ldots/p_j/p_{j+1}/\ldots/p_x/\ldots/p_l$, where $p_i =$"*", for $j < i \leq j + x$. To compare two path expressions p and p' of length l and l' respectively with $l < l'$, it suffices to use the ap^* function on p, with $n = l' - l$ and $j = l$. If p contains the "//", we set j equal to the position of the "//" and x equal to l'. If $l = l'$, then $ap^*(p, 0, j)$ just substitutes "//" with "/". In the computation of distance, we consider that the wildcard operator does not match any element tag.

Definition 4.2 (Distance Measure)**.** The distance, $dist$, between two path expressions p and p', with lengths l and l' respectively, is defined as:

$$dist(p, p') = \begin{cases} dist(ap^*(p, l' - l, l), p') = \sum_{i=1}^{l'} diff(p'_i, p_i)/l', \text{if } l \leq l' \\ dist(p', ap^*(p', l - l', l')) = \sum_{i=1}^{l} diff(p'_i, p_i)/l, \text{if } l > l' \end{cases}$$

where p_i and p'_i denote the elements in position i in p and p' respectively, and $diff(p_i, p'_i) = 0$ if $p_i = p'_i$ and 1 otherwise.

We associate a *cost* with each structural transformation, evaluated by calculating the distance between the original path expression p and the relaxed path expression $T(p)$.

Definition 4.3 (Transformation Cost)**.** The cost of each structural transformation is defined as:

- $cost(Trunc, p) = dist(Trunc(p), p) = 1/length(p)$.
- $cost(RepTag, p) = dist(RepTag(p, i), p) = 1/length(p)$.
- $cost(RepAxis, p) = dist(RepAxis(p, i), p) = dist(//p_1/p_2/... /p_n, /p_1/p_2/\ldots/p_{i-1}/*_1 /.../*_x /p_i/.../p_n) = x/length(p)$.

The cost of the last transformation is defined so that "//" is equivalent to inserting a specific number of simple steps in the path expression with the wildcard operator.

Note that cost is not defined for all pairs of path expressions. For example, it holds that $dist(a/*/b, a/c/b) = dist(a/c/b, a/*/b) = 1/3$. However, we can determine only the cost for transforming *"a/c/b"* to *"a/*/b"* which is 1/3, and not vice versa.

Let $T^t(p)$ be a sequence of t transformations applied to p. That is, $T^t(p) = T(T^{t-1}(p))$. We define the cost for a sequence of transformations as: $cost(T^t, p) = cost(T^{t-1}, p) + cost(T, p)$. If we denote as q^t the result of a sequence of transformations T^t applied to q, it is straightforward that:

$$dist(q, q^t) = dist(q, q^{t-1}) + cost(T, q^{t-1}) \tag{4.1}$$

where T denotes the t-th transformation in the sequence. A direct consequence is the following property:

Property 4.1 (Monotonicity)**.** The transformation functions are *monotonous* with respect to the distance measure *dist*. That is, $dist(q, q^i) \leq dist(q, q^{i+1})$.
Proof. This is derived from Eq. (4.1), since the cost function of any of the available transformations is a positive integer.□

A second important property guarantees that if a an XML fragment f matches the original query q, it also matches any query that is derived after the application of any possible combination of transformations on q. In particular:

Property 4.2 (No-loss Guarantee)**.** If fragment $f \in result(q^{i-1}, D) \Rightarrow f \in result(q^i, D)$.
Proof. This is a straightforward result of the type of structural transformations that only result in more general queries than the original. That is, if a query q matches a document d, then $q' = T(q)$ also matches d. For example, any fragment that matches a path expression $//q_1/q_2/\ldots/q_n$, also matches $Trunc(q) = //q_1/q_2/\ldots/q_{n-1}$. The same observation is obvious for the $RepTag(q, i)$ and $RepAxis(q, i)$ functions.□

This property ensures that by relaxing a query, we do not lose any results that may exist. Furthermore, it holds that: $|result(q^i, D)| \geq |result(q^{i-1}, D)|$.

4.2.2 Relaxation Algorithms

The *relaxation* algorithm takes as input a query and gradually applies a combination of the available transformations to it so as to attain the user-specified number of results. A central issue is the termination condition, that is, how much we need to generalize the query to attain the required number of results. We propose deriving the termination condition by exploiting the distributed index. In particular, both the order and the number of transformations that the algorithm applies is driven by two factors: the quality of the results and index selectivity estimations. Note that our index ensures that queries with results in the data have non-zero estimated selectivity.

We propose three different variations of the relaxation algorithm that vary on the portion of the total search space that each one explores.

Dynamic Programming Relaxation Algorithm. We introduce an exhaustive relaxation algorithm that applies at each iteration the transformation that will result in a query with the smallest possible distance from the original one. The index is consulted so as to consider only transformations with non-zero estimated selectivity. Furthermore, for transformed queries with the same distance from the original query (e.g. when we apply *RepTag* or *Trunc* to the same query), the index is used to select the one with the largest estimated selectivity.

If all one-step transformations result in queries with zero selectivity, we proceed by applying a second relaxation step to the already relaxed queries. When there are transformed queries that have no results with regards to the index, but have a smaller distance from the original query than that of the best relaxed query with non-zero selectivity, the algorithm continues to apply transformations to them. The process stops only when each candidate query has either non-zero selectivity or no more transformations can be applied to it or its distance is larger than that of a query with non-zero selectivity. Thus, the algorithm ensures that the selected transformations are the ones that will result in the smallest loss of quality, that is, the transformations that will lead to a query with the smallest distance from the original query. It can be shown that:

Property 4.3 (Correctness)**.** The relaxation algorithm chooses at each iteration the transformation that has the smallest possible distance from the original query among those that have not been applied yet.

Proof. Let us consider that the input query list of the algorithm consists of j queries $q_1, q_2, \ldots, q_j$ in the y-th iteration ($j \leq y$) of the algorithm. In lines 3-12, the algorithm considers every possible one-step transformation on each of the queries and computes their corresponding distances with the original query q. It then selects (lines 14-15) the transformed query with the smallest distance, i.e., it chooses to apply the transformation that will results in the smallest loss of quality from all the one-step transformations. Since every possible query is considered and no results are lost as we further relax a query (Prop. 4.2), it suffices to show that there exists no two-step (or more steps) transformation of the same query that results in a smaller loss in quality from any one-step transformation to prove that the algorithm selects the best transformation. Since the transformation function is monotonous (Prop. 4.1), this assumption holds.□

To improve the complexity of the relaxation algorithm, we use a dynamic programming technique (Alg. 8). The intermediate distances calculated at each iteration are stored in auxiliary tables ($qlist, qw, dw, q_1$) to avoid recomputing them at each iteration. An instance of Alg. 8 is shown in Fig. 4.2.

Property 4.3 ensures that at each iteration the returned results have greater (or equal) distance from the input query than the results of the previous iteration. This can be exploited to stop processing before attaining K results. In particular, each superpeer that has computed a number of K/x results, ($x \geq 1$) can forward the distance score

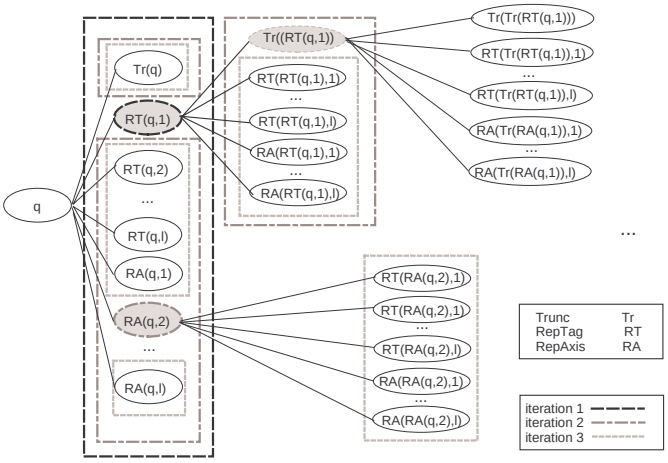


Figure 4.2: Dynamic programming algorithm. The rectangles include the candidate states (transformation sequences) considered at each iteration. The shaded oval is the best transformation at each iteration.

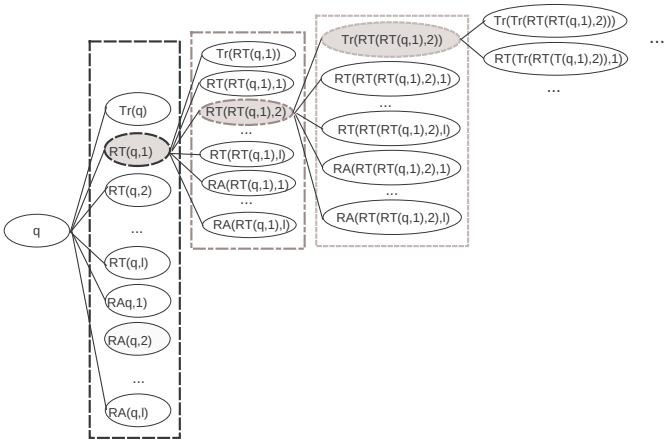


Figure 4.3: Greedy algorithm. The rectangles include the candidate states (transformation sequences) considered at each iteration. The shaded oval is the best transformation at each iteration.

of the last result to the other superpeers. After comparing their distance scores, each superpeer can decide whether it needs to continue the relaxation process or not. This way we move the pruning phase of the query evaluation earlier in the method, thus saving on processing cost at each superpeer. Note that if a random index is utilized, we do not expect to have a large gain, but with a clustered index, superpeers responsible for clusters irrelevant to the query stop their processing much earlier.

Greedy Relaxation Algorithm. We introduce an alternative *greedy* relaxation algorithm. The main difference with the dynamic programming algorithm is that at each iteration, only a single query is considered (Fig. 4.3). In particular, at each iteration the index is consulted to select among all one-step transformations the one that yields the largest number of results. This relaxed query is then used as input to the next iteration, until the required number of results is attained.

Dynamic Programming Relaxation
Input: I: Index, K: # of results, q: Query
Output: $result\_list$: list with top-K results

1: $qlist = \{(q, 0)\}$ //Input query list with pairs of the form $(expr, dist(expr, q))$
2: $q_{MIN} = NULL$ //Selected query for this iteration
3: qw, dw, q_{can}, q_1 //Lists for storing intermediate results
4: $NUM = 1$ //Number of queries in $qlist$ currently
5: $K = 0$ //Number of retrieved results so far
6: $result\_list = NULL$
7: **while** (1) **do**
8: **for** $j = 1$ to NUM **do**
9: $q_1[j].expr = Trunc(qlist[j].expr)$
10: $q_1[j].dist = qlist[j].dist + 1$
11: len =Number of path steps in $qlist[j].expr$
12: **for** $i = 1$ to len **do**
13: $qw[j, i].expr = RepTag(qlist[j], i)$
14: $qw[j, i].dist = qlist[j].dist + 1$
15: $dw[j, i].expr = RepAxis(qlist[j], i)$
16: $qw[j, i].dist = qlist[j].dist + x$
17: q_{can}:$expr$s with min $dist$ value in qw, dw, q_1
18: q_{MIN}:The query in q_{can} with the largest $EstRes$ according to the index
19: $K = K + EstRes(I, q_{MIN})$
20: Add q_{MIN} to $result\_list$
21: **if** $K < k$ **then**
22: $NUM = NUM + 1$
23: $qlist[NUM] = q_{MIN}$
24: **else**
25: **RETURN** $result\_list$

Algorithm 8: Dynamic Programming Relaxation

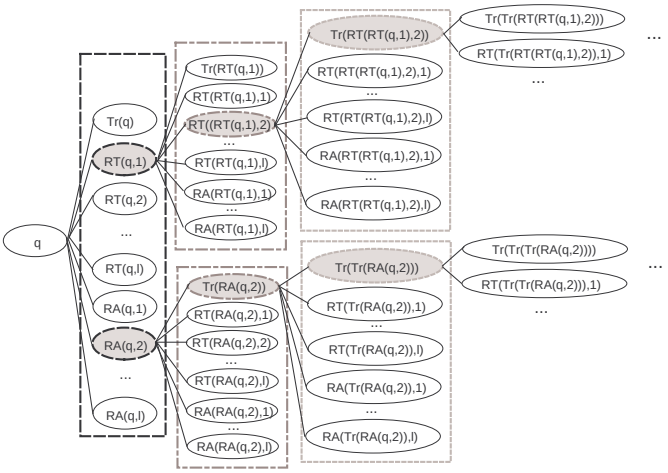


Figure 4.4: Random walks algorithm with $M = 2$. The rectangles include the candidate states (transformation sequences) considered at each iteration. The shaded oval is the best transformation at each iteration.

The algorithm may relax a query up to the most general query (i.e. "//*"), especially for large values of K. To avoid this, we use a simple heuristic in which the algorithm stops relaxing the selected query when its distance from the original query exceeds a threshold ($MaxDist$). It then uses backtracking to return to the iteration after which the largest increase in distance so far was observed. Then, the process continues by applying a different transformation to the relaxed query of that iteration.

Unlike the dynamic programming algorithm, the greedy one does not guarantee that the returned results are the actual top-K results. Thus, the greedy algorithm trades-off correctness for efficiency, since it applies a much smaller number of transformations and index look-ups by reducing the search space it considers.

Random Walks Relaxation Algorithm. To improve the quality of the results of the greedy algorithm without reaching the complexity of the dynamic programming one, we introduce a hybrid technique that considers M relaxed queries at the first iteration and applies the greedy technique in parallel to each of them (Fig. 4.4). After each iteration, the number of all returned results is compared to K to determine the termination of the process. Instead of selecting M queries randomly, we bias the walk by selecting the M queries with the largest pairwise distances. The goal is to drive relaxation towards queries that lead to a larger number of non-overlapping results. The random walks approach also does not ensure correctness. However, we expect that it provides results with better quality, since it explores a larger part of the search space.

4.3 Distributed Clustered Index

In this section, we focus on the creation and maintenance of the clustered index on which the relaxation algorithms are applied without requiring any access to the original data.

The distributed clustered index is constructed by using the multi-level Bloom histogram structure as its building block and thus, maintains all its original properties such as scalability, storage and processing efficiency and support for incremental updates. In brief, each site creates a DBH for each of its documents. The DBHs of the documents of all sites are merged based on their similarity into a number of clusters to produce a single DBH for the cluster, denoted *cluster DBH* or $clDBH$, that provides a summary of all documents in the cluster.

Cluster formation takes place incrementally as sites and documents are inserted, updated or deleted and is coordinated by the superpeers. Superpeers correspond to sites that are stable and have increased capabilities. For simplicity, we assume a one-to-one mapping between clusters and superpeers, that is, there is exactly one superpeer per cluster. However, this mapping is virtual, in that, in practice, one superpeer may be responsible for more than one cluster, or many superpeers may collaborate for the maintenance of one cluster.

4.3.1 Clustered Index Construction

The construction of the clustered index uses the procedures we defined for operations with multi-level Bloom histograms, the MergeDBH (Alg. 7) that merges two DBHs to produce a single merged DBH, and the EvaluateSim procedure (Alg. 6) for evaluating the similarity between two DBHs.

When a site joins the system, it constructs a DBH for each of its documents. The DBHs are compared against the clDBHs of the superpeers. We derive the structural similarity between a new document having a DBH, H, and the content of a cluster having a cluster DBH, clH, from the similarity between the corresponding DBHs, that is, between H and clH by using the $EvaluateSim$ procedure. After the appropriate cluster is selected, the DBH of the new document is merged with the corresponding clDBH using the merge procedure.

Based on this basic algorithm, we define two different construction procedures, one that builds a simple clustered index based on a *K-Means* clustering algorithm and one that builds a hierarchical clustered index.

***K-Means* Based Construction.** We consider as our basic input parameter the number of clusters C. When an appropriate sample of documents D_{in} has been inserted, a single superpeer is selected to gather all DBHs and to apply *K-Means* on them. *K-Means* is very sensitive to the initial sample of documents and a different sample could lead to a completely different result. For instance, even if all documents in D_{in} belong to the same semantic category, *K-Means* may still partition them into C clusters. To deal with this problem, we enforce additional constraints. After finding the C centroids (i.e, clDBHs), we check their pairwise distances and if this is lower than a value $\Delta > 0$, we merge the respective clusters.

When new documents enter the system they are forwarded to all clusters and compare against their clDBHs to locate the most similar one and the DBH of the new document

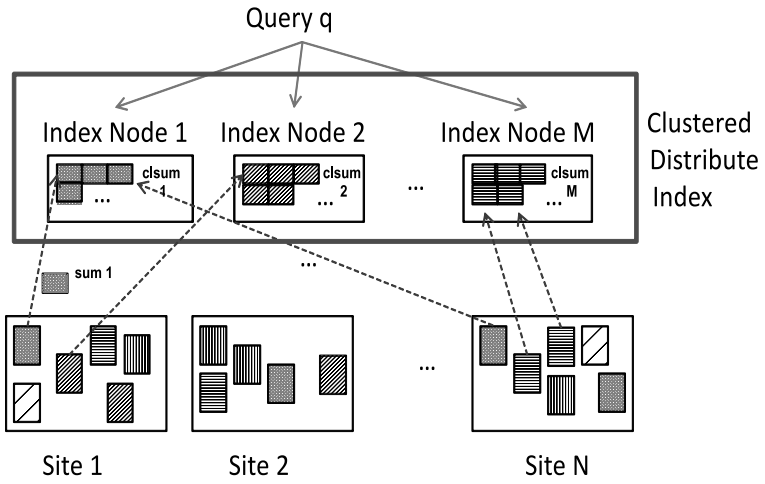


Figure 4.5: A distributed hierarchical clustered index. The clustered index is built on top of N sites that maintain the distributed XML document collections. It consists M ($N >> M$) index (leaf) nodes and a set of routing nodes that reside on top of them in the hierarchy. Each index node maintains a cluster summary $clsum$.

is merged with it. Any empty clusters will be filled with new documents, as they enter the system, if their distance with the existing clusters is large enough.

Depending on the distribution the indexed document follow, *K-Means* may create some large clusters (clusters that are assigned a large volume of documents) for popular documents, and other very small clusters, corresponding to rare documents. This poses a problem for the DBHs used to summarize the documents in each cluster ($clDBH$) as in large clusters larger estimation errors are induced. For improving performance, besides the flat distributed index with C disjoint clusters (index nodes) produced by *K-Means*, we consider also a hierarchical index organization.

Hierarchical Index Construction. The hierarchical index is built incrementally as new documents enter the system using a divisive (top-down) technique. Documents are assigned to clusters according to their structural similarity as in the flat index case, but when a cluster becomes large, we split it into a set of new clusters and partition the documents it indexed among them.

To determine when a cluster has become too large and requires a split, we rely on its clDBH and use its load as an indication of the size of the cluster. To this end, we define the *load factor* of a Bloom filter as the ratio between the number of 1s that are set in the filter and the number of elements that are inserted in the filter. We extend the definition of load factor for multi-level bloom filters as the maximum load factor among the load factors corresponding to any of the simple bloom filters that compose the multi-level one.

Each index node is assigned the same space overhead for its clDBH. We use a system-defined threshold (LF_{over}) to determine when a filter becomes overloaded, i.e., when its accuracy is decreased because of the appearance of a high false positive ratio. We then use this threshold to determine when an index nodes needs to be split in our clustering procedure.

The index is first initialized as a single index node (a single cluster). New documents that enter the system are then merged into the single cluster. After each document is inserted into the clDBH, its load factor is computed. If its value is above the threshold, a split is triggered. The split procedure (Alg. 9) reattains all the DBHs of the documents that are indexed in the clDBH, let us call it H, from their local sites. The two DBHs with the greatest pairwise distance are chosen to initialize the two new clDBHs, H_1 and H_2, that will be produced by the split. The rest of the DBHs are assigned then to their most similar clDBH and merged with it.

Split Procedure
Input: H: clDBH to be split, i: index node that maintains H
Output: H_1, H_2: new clDBHs produced by the split

1: Node i contacts the sites with documents in its clDBH, H, and re-attains their DBHs.
2: The pairwise distances of all DBHs are evaluated.
3: The two with the greatest distance are selected to initialize the two new clDBHs, H_1 and H_2 respectively.
4: The rest of the DBHs are compared to the H_1 and H_2 and merged with the one most similar to them.

Algorithm 9: Split Procedure

The nodes that are produced after the split are assigned the H_1 and H_2 clDBHs respectively, and may comply to one of the following two cases: *(Case I)*: the new index nodes maintain structurally similar documents and the split was triggered because the initial node maintaining the documents was overloaded because the size of the respective cluster became very large. Thus, in this case, the new clusters still share structural similarity and are therefore considered as sub-clusters of the original cluster that was split. The new nodes clDBHs and are either attached as children to the original node thus increasing the depth of the hierarchy, or replace it in the same level as children of its parent. *(Case II)*: the new index nodes maintain documents that do not share a high degree of structural similarity. In this case, the initial cluster became too large because it accommodated documents that may belong to different structural clusters. Then, the new clusters are not suited to become sub-clusters of the initial one and therefore are initiated as different clusters replacing the one that triggered the split.

After a split, we determine which of the two cases is the appropriate one according to the similarity of H_1 and H_2 and follow a different procedure for each. In particular: *(Case I)* If $EvaluateSim(H_1, H_2) > SimT$, where $SimT$ is a predefined similarity threshold, the new index servers that emerged from the split, are logically linked to the original index node that triggered the split as its children. The parent node that initialized the split, still maintains the index structure it had before the split (H), that now corresponds to the structure it would get by merging all of its children's indexes and is used for routing. *(Case II)* If $EvaluateSim(H_1, H_2) < SimT$, one of the index nodes replaces the initial node that caused the split, and the second one is attached as its sibling.

Thus, the nodes in the hierarchy constructed play different roles in query evaluation. The leaf nodes of the hierarchy correspond to the index nodes of a non-hierarchical index, and we call them *index nodes* similarly. The internal nodes of the hierarchy maintain a clDBH that corresponds to a filter constructed by merging the clDBHs of their children. This merged clDBH is used for routing to more efficiently locate clusters that are relevant to new documents and queries and for that we call the internal nodes *routing nodes*. Figure 4.5 shows the form of a hierarchical clustered index. In particular, when a new document needs to be inserted:

- The DBH corresponding to the new document is first forwarded to the nodes of the top-level of the hierarchy.
- The most similar node is selected and the DBH is merged with that clDBH and then forwarded to its own children and so on.
- When the document reaches a leaf node it is inserted into the clDBH and the load factor is examined to check whether a split is required and if so the corresponding procedure is initialized.

Thus, each document is maintained by all index nodes in a path in the hierarchy from the root to a leaf. This is done for routing purposes. By maintaining the property that each node has a clDBH that is the merge of the clDBH of its children we ensure that any document can be located by traversing the hierarchy and we have no information loss. Thus, we are able to find the cluster most similar to a document without needing to compare it against all clusters in the system, but instead by traversing the hierarchy.

Furthermore, the property of the index nodes enables a more efficient query evaluation procedure. Since we have no information loss, we can determine that if a query has no results against a specific index node, it also has no results in any of the index nodes that are located in the subtree rooted at it.

Note, that while we ensure that all leaf nodes have a load factor lower than the system threshold this does not hold for the routing nodes that form the hierarchy.

4.3.2 Clustered Index Maintenance

Local updates are supported by the DBH structure. We describe now how these updates are propagated from the local sites that perform the updates to the clustered index. The application of a local update results in the reconstruction of the local DBH. The site sends both the original DBH H and the updated one, H', to the corresponding superpeer that maintained the original DBH H. The superpeer removes H from its clDBH and inserts the new H' to it through merging. Note, that for the hierarchical organization the update occurs at the leaf index nodes and needs to be propagated from the leaf node of the hierarchy where it is applied to all the nodes that constitute the path leading to the root and include the old document.

Furthermore, the split procedure we have described for the hierarchical index construction may also be used for the non-hierarchical organization when a node becomes too overloaded, i.e., its load factor exceeds the system defined threshold. In this case, the second case of split is deployed in which the two new clDBHs replace the existing one. Similar steps are taken to merge two clusters, when their load becomes too low.

In particular, for the hierarchical organization an update may cause a leaf node to become too sparse. In this case, a second threshold for the load factor is utilized in accordance to the one used in the index construction. This threshold (LF_{under}) is a lower bound for the load factor and if the load factor of the clDBH H of an index node n_i becomes lower than the threshold a merge is triggered.

For the merge process, all siblings of the node that under-flowed are compared to the given index node and ordered according to their similarity to it. Starting from the most similar (we denote as H' its clDBH), the merge proceeds as described in Alg. 10. The procedure checks whether the merge would cause a new overflow and if not it proceeds with the merge. Otherwise, the next sibling in descending order of similarity is checked and so on, ensuring each time that the similarity with the corresponding sibling is above the required threshold. If the procedure does not find a sibling that would not cause an overflow, it proceeds with the most similar node it had selected in the first place and triggers a new split after the merge.

Node Merge
Input: clDBHs H and H'

1: **if** $EvaluateSim(H, H') > SimT$ **then**
2: $H'' = MergeDBH(H, H')$
3: **if** the load factor of H'' is lower than LF_{over} **then**
4: Propagate H'' upwards in the hierarchy along with the clDBHs of its siblings for the parent node to compute its own update merged filter
5: **else**
6: continue with the next sibling node according to similarity
7: **if** no such sibling exists or the next siblings also result in greater load factors **then**
8: proceed with the first selected node and H''
9: Split H'' and propagate the resulting clDBHs up the hierarchy
10: **else**
11: The merge action is cancelled.

Algorithm 10: Node Merge

By basing the merge procedure on the similarity threshold, i.e., no merge is allowed if the similarity between two nodes is lower than this threshold, we favor the creation of clusters of structurally similar documents even if such clusters are unbalanced, rather than building a balanced distribution of the data that would be the case if we ignored the similarity threshold and base our algorithm only on the load factor.

In the case where the merge triggers another split, what we practically accomplish is to redistribute the documents between the two most similar nodes when one becomes too empty aiming to achieve a more balanced distribution. When the process is complete, the updated DBH structures are again propagated up the hierarchy.

4.4 Distributed Query Evaluation

Query evaluation is applied on top of the clustered index (or the hierarchical clustered index) and it is coordinated by the superpeers. In particular, each query is assigned to a coordinator superpeer. All superpeers process each query in parallel by relaxing it against their part of the index until they attain K results. Then, they exchange the distance scores of their K-th result and prune their result list according to these distance scores. Finally, they send the remaining entries in the list to the coordinator, which after receiving all lists, proceeds in constructing the final top-K list by merging them.

Alternatively, we may employ a pay-as-you-go approach, in which the clusters are processed one-by-one based on their relevance to the query. This can be achieved as follows. Each query can be represented as a tree, thus we construct the DBH for the query. We rank the clusters based on the similarity between their cluster index, clDBH, and the query DBH and then visit them in descending order of similarity.

4.4.1 Top-K Threshold-based Evaluation

The process that is followed to retrieve the top-K results proceeds in phases. When a query is issued by a site, the site propagates it to a superpeer. In the first phase (*Local Query Evaluation*), the superpeer forwards the query to the other superpeers. In parallel, each superpeer evaluates the query against its part of the index to attain K results. Note that, with results, we refer to index entries and not to the actual data that are stored at remote sites. These results are sorted locally at each superpeer, in non-decreasing order, according to their distance from the original query. We describe first a flat organization of superpeers (clusters) and then the hierarchical one.

By following a procedure commonly used in a family of distributed top-K evaluation algorithms (*Threshold-based Algorithms* [39]) for reducing communication costs, in a second phase (*Elimination Phase*), superpeers exchange the distance scores of their K-th result (the result in the list with the largest distance from the original query). Upon receiving these distance scores, each superpeer discards the results with distance score larger than the smallest of all the distance scores it has received. In the third phase (*Result Construction*), each superpeer forwards the remaining results to the superpeer from which it has received the query. After the initial superpeer has gathered all lists, it merges them to construct the final result list. Then, the site that issued the query contacts the sites in the ranked list to retrieve the actual data. If some sites lack the required processing capabilities, they forward their data to the initial superpeer to perform the

actual query processing.

The elimination phase of the routing process can consist of more than one round for further pruning the result list and thus reducing the size of the data that needs to be transmitted. In the second round of this phase, the superpeers exchange the distance score and the position in their ranked list of the last result (in ascending distance order). Let m be the smallest distance score among all the distance scores sent. Each superpeer compares m to its own list and finds all results with a distance score larger than m. If the sum of the position of such a result with the position of m is greater than K, then this result can be safely discarded, since it does not belong to the final result list. The elimination phase can then proceed with the next round taking into account the newly acquired pruned lists in a similar manner to further reduce the number of results.

Assume N superpeers, each maintaining a part of the distributed index U_i, $1 \leq i \leq N$. Let us denote each iteration (round) of the elimination phase of the query processing algorithm as $round_j$. We define the *pruning degree* to measure the number of results that are eliminated in each round of the elimination phase as follows:

Definition 4.4 (Pruning Degree)**.** For each part U_i of the distributed index and each $round_j$ of the elimination phase, the pruning degree ($PD_j(U_i)$) is defined as:

$$PD_j(U_i) = \frac{Eliminated_{ij}}{K}, \tag{4.2}$$

where $Eliminated_{ij}$ is the number of eliminated results in the partial results list of superpeer i responsible for part U_i.
We consider the average pruning degree (PD_j) for each $round_j$:

$$PD_j = \frac{\sum_{i=1}^{N} \frac{Eliminated_{ij}}{K}}{N} = \frac{\sum_{i=1}^{N} Eliminated_{ij}}{N * K} \tag{4.3}$$

The larger the pruning degree, the more efficient the elimination phase, since less data needs to be transferred to the superpeer that initiated a query. Furthermore, a large pruning degree indicates that less processing is required for constructing the final result.

Using a clustered index increases the pruning degree and thus reduces the number of documents that need to be considered during relaxation. To see this, consider the following simple example. Assume a set $\mathcal{D}$ of XML documents that can be classified in N categories (C_i) each having x documents, such that if a query q matches a document $d' \in C_t$, $1 \leq t \leq N$, it does not match any documents belonging to any other category. Also, assume that there are more than K exact matches for q. We partition the documents to N superpeers: (a) uniformly at random, where x/N documents of each category fall into each superpeer and (b) by assigning the documents of each category to a different superpeer. In (a), the results of q are expected to be distributed uniformly among the superpeers while in (b), the results reside at a single superpeer. For simplicity, we assume that all non exact matches have the same distance from q. While in (a), all exchanged distance scores have the same value and no pruning is possible, in (b), the superpeer of C_t has a K-th distance score equal to 0 and all other superpeers can prune their entire

lists. This simple example shows that a clustered index can increase the pruning degree, when there is similarity among the documents.

Hierarchical Index based Evaluation When the index servers are organized into a hierarchy, then the elimination phase instead of rounds, proceeds *in levels*. That is, the index servers belonging to the same level of the hierarchy exchange their scores to determine how many results need to be pruned and whether the evaluation needs to continue to their subtree. In particular, a query is first sent to all the nodes in the top-level of the hierarchy. The evaluation proceeds at each internal index node of the hierarchy by computing the local top-K results and in a similar way to the elimination phase, exchange the score of the K-th result with all the other nodes in the same level of the hierarchy that are still active. Nodes that manage to prune their entire local result lists do not forward the query any further into their subtrees, while the other nodes forward the query to their children that repeat the process (Alg. 11).

Hierarchical Top-K Evaluation
Input:q: query, K: number of target results

1: **for all** internal index node i in level j that receive q **do**
2: evaluate q against i's clDBH and derive the local top-K results
3: send K-th score to all other nodes in j
4: **if** node i cannot discard its entire top-K list **then**
5: forward q to its children
6: **else**
7: stop the evaluation in its subtree

Algorithm 11: Hierarchical Top-K Evaluation

The leaf nodes in the hierarchy that may be found at any level since the hierarchical index is not balanced, need to take some different actions. In particular,

- If a node is a leaf node in level i, then it participates in the elimination phase of each level from i to the bottom of the hierarchy (unless it can prune all its results later).
- When all the remaining nodes are leaf nodes, the elimination phase is the one described in the threshold-based algorithm that leads to constructing the final result list. (multiple rounds can be utilized in this case among the leaf nodes).

Based on this evaluation procedure, the hierarchical index enables the query processing to prune whole subtrees in the hierarchy, thus further increasing the pruning degree of the system and additionally avoiding any computation on those sites (which is a benefit not captured through the pruning degree).

4.4.2 Incremental Query Evaluation

Using a distributed clustered index provides another advantage. In particular, it enables an alternative query evaluation strategy that follows the principles of a *pay-as-you-go* approach. That is, the query evaluation proceeds incrementally to reduce the time that a user has to wait to obtain results. The pay-as-you-go strategy exploits the locality of the entries at each index partition. With the clustered index, query processing may start with the most relevant to the query cluster (for instance, with the cluster with the most similar to the query entries) and proceed gradually to access the remaining clusters. This way, we "prune" the number of sites that need to be considered for processing the query by excluding the potentially irrelevant ones. The processing and communication cost increase, while users receive more responses to their queries, i.e, gradually paying for the results they get.

To determine the order in which the clusters are considered, we rely on the similarity of the cluster index clDBH to the query. The query is represented as a tree-pattern, and then inserted into a corresponding DBH (qDBH). Based on the similarity of the qDBH to the clDBH, we determine the order to visit the clusters starting from the most similar one and proceeding in non-ascending order of similarity. To compute the similarity in this case, we consider only the Bloom filters of the clDBH and ignore the information maintained in the histogram buckets.

Thus, the evaluation proceeds as follows. The query is forwarded to all index servers in the system and compared against their index units. The units are then ordered according to their similarity and the query is then evaluated against the index units in this order returning the top-K results to the user every time it finishes processing a unit.

Though the evaluation proceeds gradually on the premises of the pay-as-you-go approach, the process still needs to first compare the query against all index units and order them according to their similarity before it starts the evaluation. Using a hierarchical index organization avoids this step and enables us to probe index units incrementally. In particular, the query is forwarded top-down the hierarchical index, selecting at each node to be propagated to its child node with the most similar clDBH to the input query (Alg. 12).

When a query reaches a leaf node of the hierarchy, it is evaluated against its clDBH to attain the top-K local results that are then send to the user. A type of backtracking is used to determine the next index node that needs to process the query. In particular,

- When a leaf node receives a query for forwarding it evaluates it against its clDBH to obtain the top-K results that are send to the user.

- The node then sends a completion message to its parent that forwards the query to its child that follows in order of similarity as it it was evaluated at the previous steps of the evaluation.

- When there are no children left for the parent node to forward the query, it sends

Incremental Hierarchical Evaluation
Input: q: query

1: **if** an index node i receives q from its parent in the hierarchy **then**
2: it compares the query against its clDBH
3: sends the similarity value to its parent
4: **if** an index node i receives the similarity values from its children **then**
5: It selects the child with the greatest similarity and propagates the query to it for $forwarding$
6: **if** an index node i receives a query from its parent for $forwarding$ **then**
7: it propagates the query to its own children

Algorithm 12: Incremental Hierarchical Evaluation

a completion node to its own parent that repeats the procedure for its respective children nodes.

- The procedure stops either when all the leaves have been visited or when the user determines that it does not require more results.

The index nodes that receive a query from one of their children after it has completed its evaluation have already evaluated the query against all of their children and thus the similarity computations are not repeated. In the backtracking process the query will be propagates also to nodes for the first time and these nodes need to evaluate their clDBH similarity to the query. However, if these values are maintained the comparison needs to be performed only once.

Thus, using a hierarchical index enables us to attain results without accessing and evaluating the similarity of the query against all our index nodes. In particular, to return the first top-K results to the user, we need to access less than (hierarchy depth)*(out degree) nodes.

4.5 Experimental Evaluation

We use the data sets used in the evaluation of the multi-level Bloom histogram (Table 3.3). For query generation, we use the zipf distribution to select paths from the documents. Then, we replace a random element with a foo element, or insert a subpath of random length in them. Our multi-level Bloom histogram tuning parameters are as in the evaluation of the structure (Table 3.2) and we repeat them for clarity in Table 4.1 along with the rest of the parameters used for the evaluation of the distributed clustered index and the structural relaxation algorithms.

Table 4.1: Input parameters for the evaluation of distributed structural relaxation

Parameter	Default Value	Range
# of documents	500	50 - 500
# of clusters	8	2 - 24
K	20	10%-50% of the results
# of data categories	8	-
Size of document	2KB-120KB	-
DBH size	320KB	5 - 320KB
DBH depth	3	-
DBH hash functions	4	-
load factor	0.4	0.1-0.7
similarity threshold	0.7	0.1-0.9

4.5.1 Evaluation of the DBH as Building Block of the Distributed Index

To evaluate the benefits we derive from using the DBH structure as the building block for the distributed clustered index in our system, we focus on the construction cost with respect to the communication cost entailed, as that is the most important overhead that determines efficiency when dealing with distributed systems. In particular, we measure the construction cost of the clustered index, as the total size of messages required, while varying the number of clusters (Fig. 4.6(left)) and the number of documents (Fig. 4.6(right)). The deployment of the DBH reduces the construction cost of the clustered index from GBs to only a few MBs from the case when PCT is used (Fig. 4.6(left)) without damaging performance significantly, since the pruning degree is only slightly decreased (Fig. 4.10(left)).

Scaling. To evaluate the scaling capability of our index, we keep a fixed size for the DBH and increase the number of documents maintained. As Fig. 4.7 illustrates, our index structures scale gracefully, since the pruning degree does not decrease significantly.

Summary of Results.

- The Depth Bloom Histograms reduce the construction cost of the clustered index from GBs to MBs.
- The Depth Bloom Histograms scale well with respect to the number of documents.

4.5.2 Comparison of the Relaxation Algorithms

In this set of experiments, we evaluate and compare the performance of the three relaxation algorithms, the dynamic programming (DP), the greedy (GR) and the random walks (RW) relaxation algorithm. For random walks, we set $M = 3$ and $M = 4$, so that 3 and 4 different queries are relaxed respectively. We apply the algorithms on a DBH

with clustered documents (Fig. 4.8) and on a DBH with random documents (Fig. 4.8). We evaluate the quality of the results the algorithms provide by measuring the average distance of the top-K results produced to the original query, as well the processing cost of each approach measured by the number of index look-ups required.

The dynamic programming approach provides the results with the best quality for both clustered and unclustered documents, which are also the actual top-K results present in the collection so their quality is bound by the actual data relevance to the query. Dynamic programming is also, however, the most expensive approach with a processing cost about 60% greater than the one of the greedy approach. The greedy approach has the worst overall quality in its results, and the difference is greater when the documents are clustered. With clustered documents there are a lot of "good" relaxed queries that are explored with dynamic programming that the greedy approach ignores. For unclustered documents, since the good queries are less the difference is somewhat smaller. The random walks approach is a compromise between the two extremes and produces results with better quality than the greedy and lower cost than the DP approach. M is the parameter that regulates the tradeoff determining whether the random walks approach behaves more like the greedy or like the dynamic programming algorithm.

Summary of Results.

- All algorithms handle both clustered and unclustered documents and provide results at the worst case with a quality about 35% lower than the quality of the actual available results.
- The dynamic programming approach evaluates the actual results, but entails a processing cost 60% greater than the greedy algorithm's one.
- The random walks approach performs in between the two others depending on the value of M.

4.5.3 Benefits of a Clustered Index

In this section, we evaluate the performance of our two query evaluation techniques against a clustered and an unclustered index. We use four approaches for constructing the distributed index. The first two approaches use a simple path count table (PCT) as the building block of the index, while the latter two use the Multi-Level Bloom histogram (DBH). Using the two index structures, we construct both a random index, where the data is assigned to superpeers uniformly ($randPCT$ and $randDBH$) and a clustered index ($clustPCT$ and $clustDBH$).

Pruning Degree. We compare a clustered and a non-clustered (random) index using dynamic programming (DP) relaxation. This clearly depends on the number of clusters. Thus, we measure the average pruning degree varying the number of clusters from 2 to 24. Figure 4.10 shows that for all numbers of clusters, the pruning degree is improved in the case of clustering as expected. Specifically, as we assign multiple categories to the

same cluster (2 and 4 clusters), the pruning degree increases for the clustered index, while it remains almost constant for the random one. This is because if documents from one category are distributed among different clusters, relaxation at each superpeer produces results with similar quality. Using as many clusters as the document categories or less, alleviates this problem.

Increasing the number of clusters, reduces the load at each superpeer, but increases the communication cost among them. To demonstrate this trade-off, we measure the maximum processing cost among all superpeers (Fig. 4.11(left)) and the average communication cost during query processing (Fig. 4.11(right)). As processing cost, we measure the lookups performed in the index during relaxation and the number of entries that are processed for the construction of the final result list. While for a small number of clusters the first cost is the prominent one, i.e., 90% of the total cost, as the number increases the second cost reaches up to 56% of the total cost.

Greedy and Random Walks Relaxation. We repeat the same set of experiments using greedy (GR) and random walks (RW) relaxation with $M = 3$.

Figure 4.12(left) shows that the pruning degree is again larger with clustering for both approaches. Compared to the performance of the DP approach, pruning degree is decreased by 20% for the greedy one, and by 6% for random walks. Due to this decrease in the pruning degree, the communication cost increases correspondingly for both approaches (Fig. 4.13(right)). However, their advantage over the DP approach is that the processing cost required is reduced by around 24% for the GR and 15% for the RW approach (Fig. 4.13(left)). We observe that in this case the gain in processing cost we have with the two approaches is smaller compared to the centralized case. This is because the larger pruning degree results in longer results lists that need to be merged to construct the final result list, thus explaining the larger overhead in processing.

This gain in the processing cost results in a loss in the accuracy. We evaluate this loss by measuring the recall of the greedy and the random walks algorithms against the DP one (i.e., the percentage of results returned by each method that belong to the actual top-K results). For the random walks approach we used two different configurations with $M = 3$ and $M = 4$. As Fig. 4.12(right) shows, the RW approach exhibits the best recall when $M = 4$ almost 99%, while the greedy approach has the lowest recall of all (46% in the worst case).

Multiple Rounds in the Elimination Phase. We want to evaluate the benefits of using multiple rounds in the elimination phase. We measure the pruning degree at each round using all three relaxation algorithms (Fig. 4.16). The largest pruning degree is achieved on the first round, while the other rounds prune only a small number of results. If we consider the time/communication cost trade-off, it is clear that we do not need to use more than two rounds as the response time of the evaluation would only increase significantly without bringing a considerable gain in the communication and processing cost.

Pay-as-you-go Evaluation. In this experiment, we evaluate the performance of a *pay-*

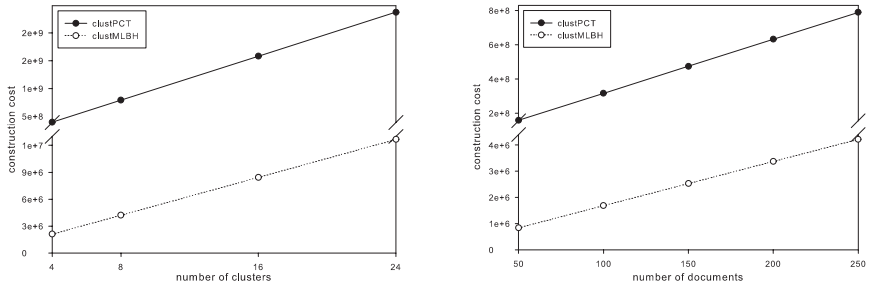


Figure 4.6: Construction cost with respect to (left) varying clusters and (right) number of documents.

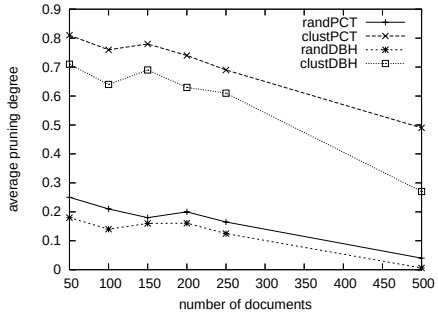


Figure 4.7: Scaling.

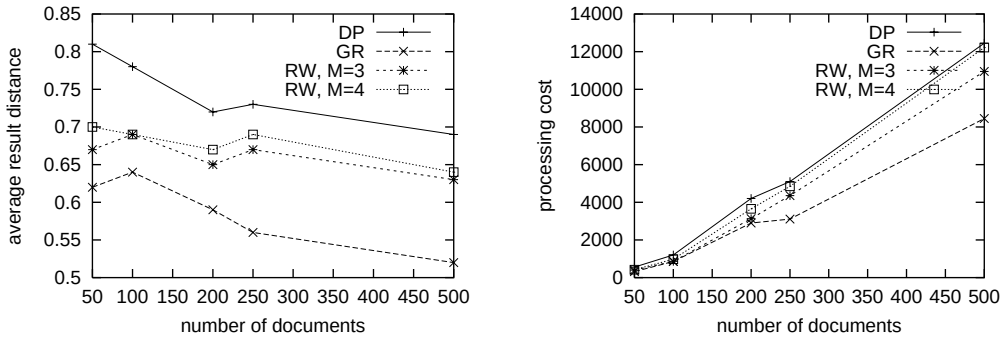


Figure 4.8: (left) Quality and (right) processing cost with clustered documents.

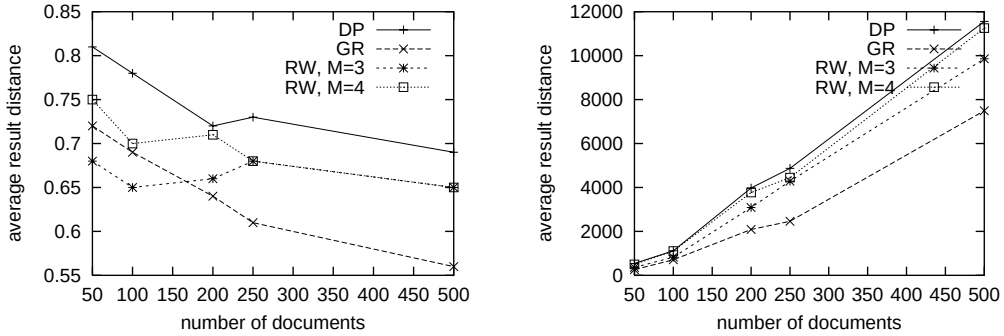


Figure 4.9: (left) Quality and (right) processing cost with randomly selected documents.

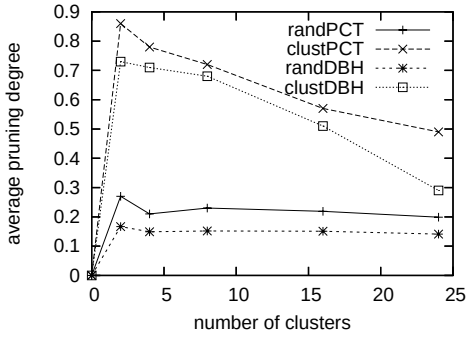


Figure 4.10: Pruning degree when using dynamic programming.

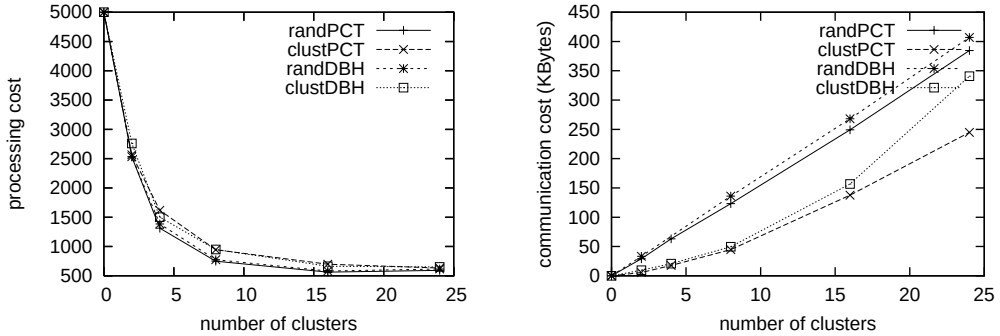


Figure 4.11: (left) Processing and (right) communication cost when using dynamic programming.

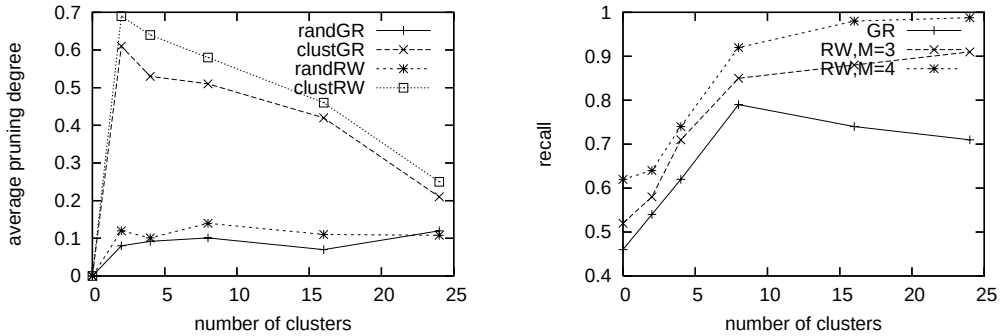


Figure 4.12: (left) Pruning degree with the greedy and random walks approach and (right) comparison of the three methods.

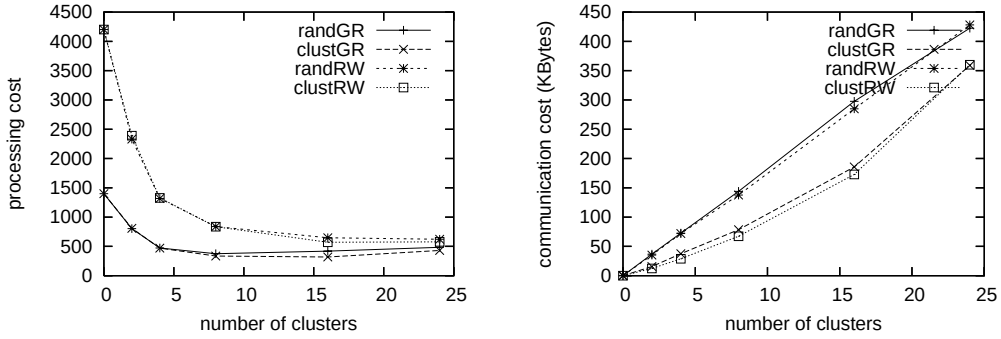


Figure 4.13: (left) Processing and (right) communication cost with the greedy and random walks approach.

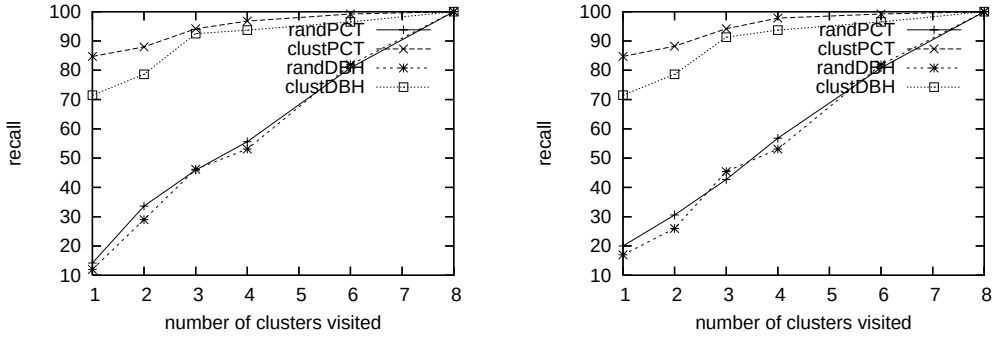


Figure 4.14: Pay as you go with dynamic programming based on (left) average distance of results and (right) query similarity.

as-you-go strategy in which clusters are visited gradually, starting from the "best" one for each query. We determine the visiting order of the clusters (index nodes) based on the similarity of the query to the clDBH. We also report our results when using an approach in which the clusters are visited in ascending order of the average distance of their results to the query. That is, an approach that assumes knowledge of the quality of the results and visits the clusters in optimal order, if we consider as optimal strategy visiting the cluster with the most relevant results first.

We measure the results accuracy (recall) each approach provides defined as:

$$accuracy = \frac{(K \text{ results returned}) \bigcap (\text{actual top } K)}{K}, \tag{4.4}$$

where with K results returned we refer to the results returned by each approach after visiting an index node. We measure this recall while gradually increasing the number of clusters visited.

Figure 4.14(left) shows that pay-as-you go works well with clustering. The clustered index achieves much higher recall than the random one in which recall increases proportionally to the clusters that processed the query.

As shown in Fig. 4.14(right), the ordering based on the similarity of the query to the clDBHs almost reaches the performance of the first approach, and both the greedy and random walks approach perform similarly to the case of threshold-based top-K evaluation,

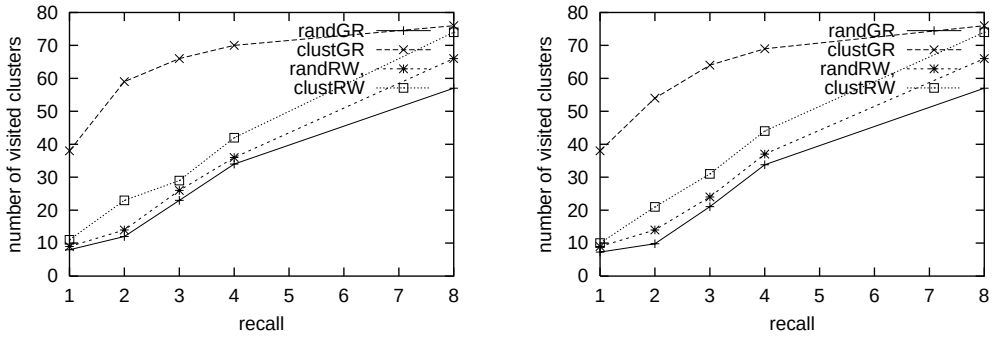


Figure 4.15: Pay as you go with greedy and random walks based on (left) average distance of results and (right) query similarity.

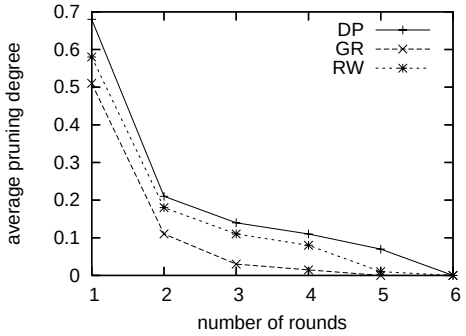


Figure 4.16: Pruning degree with multiple rounds.

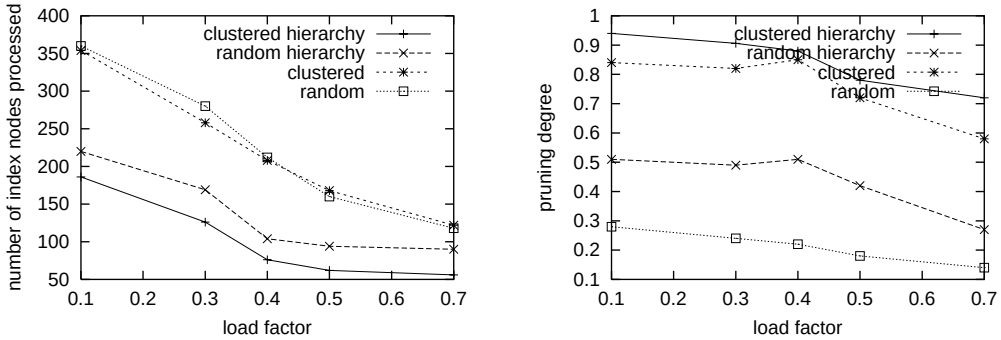


Figure 4.17: (left) Number of sites probed and (right) pruning degree for varying load factor.

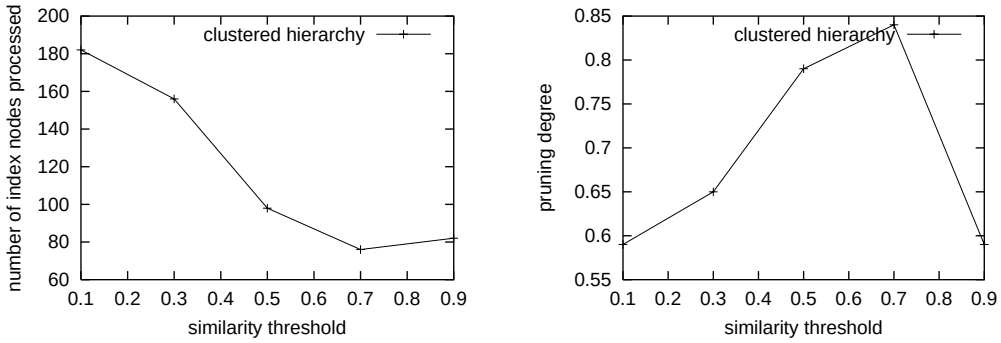


Figure 4.18: (left) Number of sites probed and (right) pruning degree for varying similarity threshold.

worse than the dynamic programming one (Fig. 4.15) but saving on processing cost at each visited node as we showed in the previous set of experiments. Since in this case we do not have the additional cost of a final result list, the gain in processing cost is much larger. Thus, this simple similarity measure can be used as an efficient criterion for cluster selection enabling us to avoid having to process a query against all clusters.

Summary of Results.

- Using a clustered index improves the pruning degree up to 250% compared to a random index.
- The largest pruning degree is achieved by the dynamic programming approach that also requires the lowest communication cost. The greedy and the random walks relaxation approaches result in worse performance in terms of pruning degree and communication cost, but both reduce the maximum processing cost (around 24% the greedy and 15% the random walks). Furthermore, despite not considering the whole search space, both approaches are still effective, achieving a recall up to 79% for the greedy and 98% for the random walks approach with $M = 4$.
- The clustered index can be effectively used as the base for a pay-as-you-go query evaluation, achieving high recall. Query similarity to the clustered index can be used effectively for determining the order of visiting the clusters in a pay-as-you-go evaluation.

4.5.4 Hierarchical Index Evaluation.

In this set of experiments, we examine how organizing the index hierarchically further improves the performance of the clustered index. We use four different variations of the distributed index; a random index, a clustered index, a hierarchical random index and a hierarchical clustered index. We set the size of the cluster unit (index node) the same in all cases. To attain a fair comparison between the hierarchical and non-hierarchical clustered index that is constructed based on *K-Means*, we first apply the hierarchical clustering procedure and then run *K-Means* with K equal to the number of leaf nodes

in the hierarchy. The random hierarchical index is constructed by using only the load factor to determine the split procedure and using a random way to determine which path to follow in the hierarchy when inserting a new document. The two new nodes have 0.5 probability to be attached as children of the split node, and 0.5 probability to be attached as siblings replacing the split node. The load factor is also used in the non-hierarchical indexes and causes the index node that is overloaded to split into two new nodes. In the clustered index, the documents are assigned to each node based on their similarity as in the clustered hierarchical index construction, while in the random index the nodes are assigned randomly to the two new nodes.

Threshold-based Top-K Evaluation. We first evaluate the performance of the top-K threshold based algorithm by measuring the number of index nodes that are accessed and evaluate a query as well as the pruning degree with varying load factor (Fig. 4.17) and varying similarity threshold (Fig. 4.18). For the similarity threshold we only evaluate the performance for the clustered hierarchical index since this parameter is not used in any of the other approaches. Also, when measuring the average pruning degree we consider that index nodes that were not accessed have the optimal pruning degree of 1.

Overall, the hierarchical indexes outperform the flat ones in terms of the number of nodes that evaluate a query. That is because the query is sent to all sites in the non-hierarchical indexes, while when using a hierarchical index some nodes may prune their subtrees during the elimination phase, thus pruning index nodes from the evaluation. With regards to the pruning degree, the clustered approaches exhibit the best performance, though the random hierarchical index also performs well compared to the clustered non-hierarchical one since its pruning degree is improved by pruning entire index nodes. The clustered hierarchical index has the best overall performance. We observe also that while increasing the value of the load factor improves performance, there is an upper bound in its value beyond which the performance deteriorates. Thus, very sparse index nodes do not favor performance since they lead to a very large number of nodes we need to access for each query. However, if the load factor allows for very dense index nodes errors due to false positives and estimation errors also damage performance. Similarly, the similarity threshold also needs to be tuned accordingly. High thresholds tend to create shallow wide hierarchies with less opportunities for pruning entire nodes, while small thresholds favor deep hierarchies in which we need to access many nodes before reaching any leaf nodes.

Pay-as-you-go Evaluation. We use the heuristic based on the similarity of the query to the clDBHs to determine the visiting order of the clusters and measure the overhead involved in the procedure in terms of index nodes probed, i.e., clDBHs compared to the query and also the number of index nodes that actually evaluate the query with respect to recall (Fig. 4.19).

Using the hierarchical index significantly decreases the number of index nodes probed as we probe and evaluate the query against clDBHs gradually in contrast to the non-hierarchical organization where all nodes need to be probed before we obtain any results.

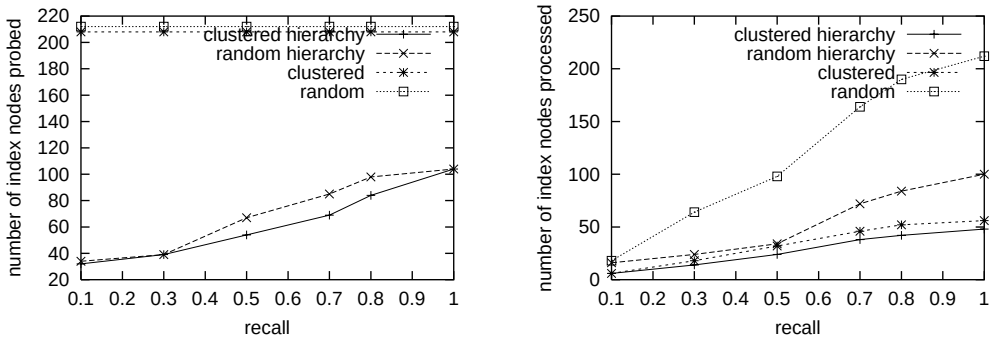


Figure 4.19: Recall with respect to number of index nodes (left) probed and (right) processed.

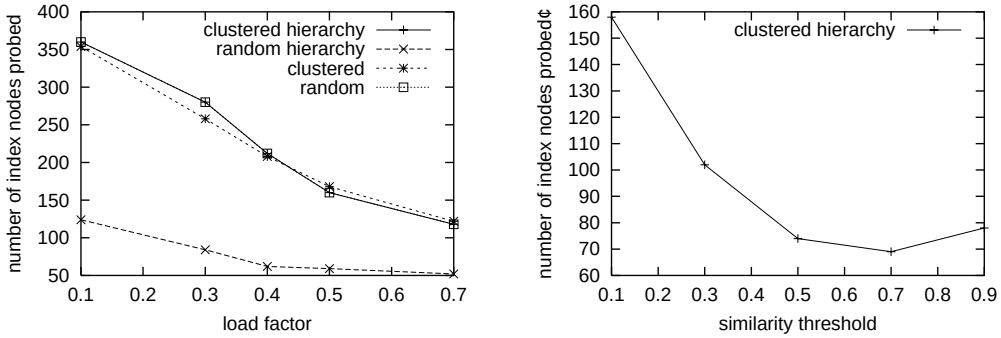


Figure 4.20: Number of sites probed with varying (left) load factor and (right) similarity threshold.

Again, the clustered hierarchical index is the one with the best overall performance. The number of index nodes that evaluate the query to obtain a certain number of results is also reduced when we use the hierarchical index as the clusters that are constructed through the hierarchical clustering process are of better quality than when using *K-Means* for the cluster construction. In particular, since *K-Means* is very sensitive to the initial sample, we have cases in which the performance is in that case significantly worse. Finally, if the same category of data is partitioned in many cluster units, the hierarchical index manages to locate all these partitions more effectively since they are located close in the hierarchy, while in the non-hierarchical index some of these units may be missed. This is because we compare the actual query to the clDBHs and not its relaxed forms. Thus, units that only maintain relevant results may be in some cases overlooked.

We also examine the influence of the load factor (Fig. 4.20(left)) and similarity threshold (Fig. 4.20(right)) by measuring the number of index nodes probed for attaining a recall above 70%. The results are similar to the ones obtained when evaluating the threshold-based approach.

Scaling. We increase the number of documents in the index and therefore the resulting index nodes keeping the load factor threshold set to 0.4 to evaluate the scaling properties of our index. Fig 4.21(left) and Fig 4.21(right) report the number of index nodes processed for the threshold-based evaluation and the number of index nodes probed for the pay-as-

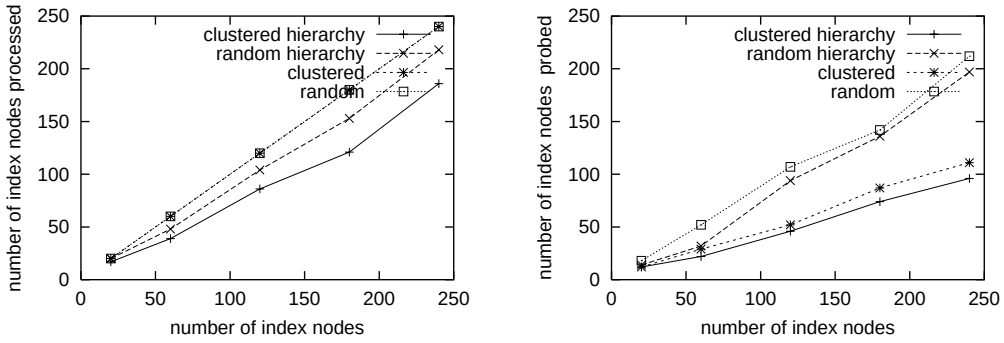


Figure 4.21: Scaling for (left) threshold-based and (right) pay-as-you-go query evaluation.

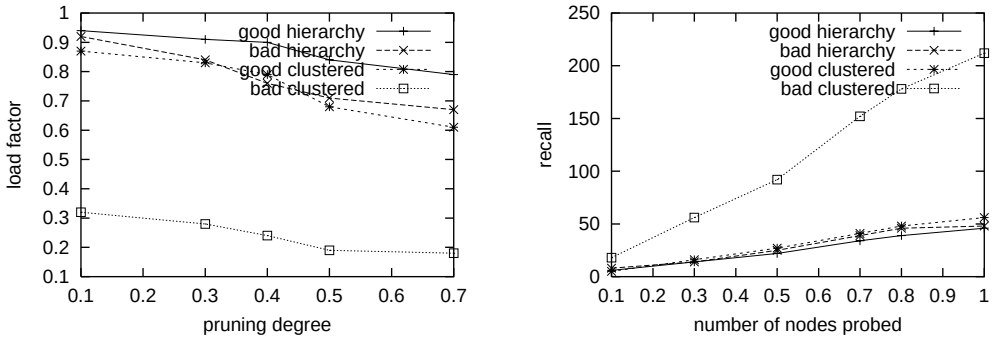


Figure 4.22: Bad initialization for *K-Means* for (left) threshold-based and (right) pay-as-you-go evaluation.

you-go evaluation. The hierarchical clustered index exhibits the best scaling properties. Though the number of processed nodes increases with the number of index nodes because more nodes need to be accessed to retrieve the required results, the difference with the non-hierarchical index organization also increases exhibiting the scalability of the hierarchies.

K-Means Initialization. The *K-Means* clustering algorithm used to initialize the non-hierarchical index is very sensitive to the initial sample of documents on which the clusters are initiated and the selection of K. Though in our experiments K has not the appropriate value, we chose it such to achieve a fair comparison with the hierarchical index and also on the premise of allocating to each index node a certain amount of space, thus leading to the documents of one semantic cluster to be partitioned among many index nodes. Otherwise the resulting index nodes would be overloaded. Thus, we evaluate the influence of the sample of documents and the order in which they are inserted. We compare the performance in both the threshold-based evaluation and the pay-as-you-go approach for an index in which the initial sample constituted of documents of a single category and the documents were inserted by category, and an index in which the initial sample constituted of documents of all the categories following the distribution of all the documents in the system and then the documents were inserted randomly into the index. We also compare these non-hierarchical indexes with the respective hierarchical ones. The hierarchical clustered index is much more resilient to the order of insertion than the non-hierarchical one, of which the performance when the initial sample is bad is as bad as the random

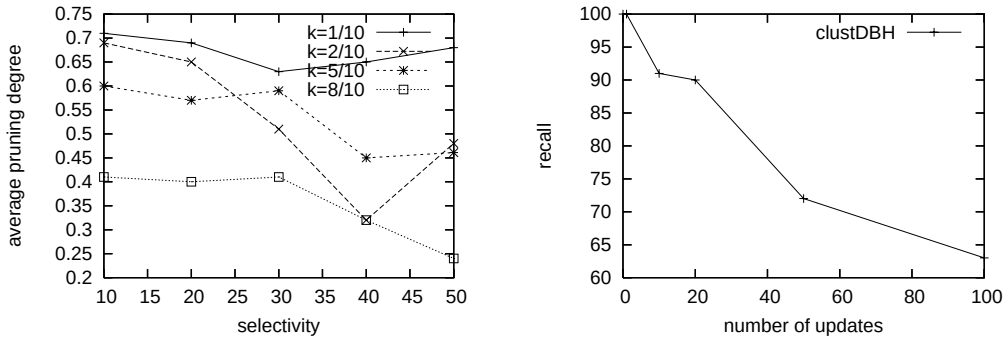


Figure 4.23: (left) Influence of K and (right) update propagation.

index (Fig 4.22).

Summary of Results.

- The hierarchical index organizations (clustered and random) outperform the respective non-hierarchical indexes in terms of accessing less nodes during query evaluation for both the threshold-based and the pay-as-you-go query evaluation.
- The best overall performance is achieved by the clustered hierarchical index that has the highest pruning degree in the threshold-based evaluation and the highest recall in the pay-as-you-go approach.
- The *K-Means* based construction of the non-hierarchical index is very sensitive to the initial sample of documents and a bad initialization leads to very low performance.
- Though the processing cost (in terms of probed or processed nodes) increases linearly with the number of index nodes for non-hierarchical indexes, it scales gracefully if a hierarchy is deployed.

4.5.5 Other Issues

Influence of *K*. It is clear that the value of K (i.e., the requested number of results) affects performance. The determinant factor is how similar the K-th result of the relaxed query is to the results of the original query. If sufficient results with a small distance from the original query do not exist, relaxation leads to queries with a very large distance from the original one, which match data at various clusters. To demonstrate this, for a distance x and a query q, we express K as a fraction of the available results with distance from q lower or equal to x. We fix x and measure the average pruning degree as the number of available results with at most this distance from the queries vary. For larger values of K the pruning degree decreases as low quality results, which are located at various clusters, are retrieved (Fig. 4.23(left)).

Dynamic behavior. The communication cost for propagating a local update to the clustered index depends solely on the size of the DBH. Thus, we focus on how well the

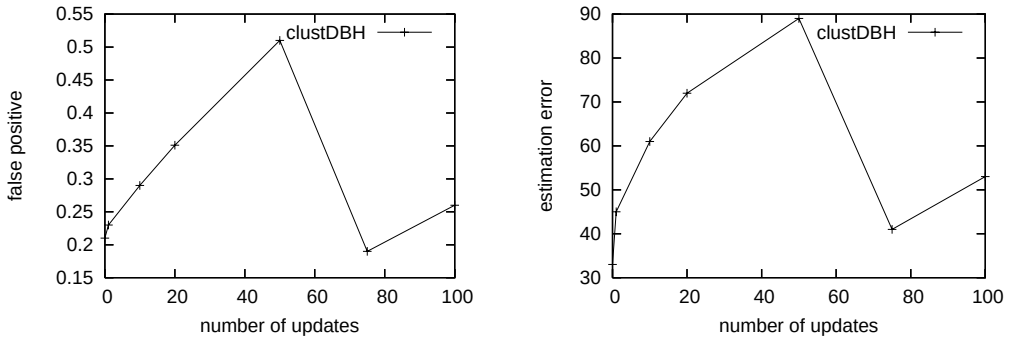


Figure 4.24: Cluster split influence on (left) false positives and (right) estimation error.

DBH's estimation accuracy is maintained after a series of updates. To this end, we consider two approaches. In the first, we apply updates on our DBH using the update propagation procedure, while in the second one, we reconstruct a new DBH with the updated data (let H and H' be the resulting DBHs, respectively). Next, we perform a batch of queries and measure the recall of H compared to that of H', i.e. we measure the percentage of results provided by H that are also provided by H'. Reconstruction may be required, when recall decreases. Our results (Fig. 4.23(right)) show that updates do not significantly affect performance and thus, reconstruction may not necessarily be frequent. Note that 20 updates correspond roughly to 1/3 of the cluster documents being updated.

Cluster Split. The communication cost for a cluster split depends on the number of documents that are indexed by the cluster before the split and the size of the DBH. We focus on the performance of the split process. In particular, we measure the false positive ratio and the estimation error in a clDBH as new documents are inserted into it, until a split is triggered (in this experiment after 50 insertions) and then after applying the split we show the average estimation error and false positive ratio for the two new clusters that were produced as new documents are still inserted and accordingly placed into one of the two clusters. From Fig. 4.24(left) and Fig. 4.24(right), we see that the split process can deal with the overloading of a single cluster and balance the document load between the two new clusters as the system continues to evolve.

Summary of Results.

- The pruning degree depends on the similarity of the available results to the original query. For large values of K, when sufficient similar results are not available, it decreases.
- Even after 1/3 of the documents have been updated, the update procedure maintains recall up to 90%. Cluster split deals effectively with the problem of cluster overloading.

4.6 Summary

In this chapter, we used the multi-level Bloom histogram for building a distributed clustered index and based on that, we addressed the problem of the efficient evaluation of XPath approximate queries over dynamic distributed collections of XML data. The use of the distributed clustered index reduced the communication cost required for evaluating top-K queries by enabling the sites responsible for query evaluation to prune the number of candidate results they need to consider. Organizing the clustered index into a hierarchy improved performance further by reducing the number of sites that need to be considered for each query significantly. Thus, we showed that the clustered index improves performance either by reducing the number of results exchanged in threshold-based distributed top-K processing or/and by considering only data in selected clusters.

The distributed clustered index for supporting XPath structural relaxation over collections of documents is presented in [73], [75], while the hierarchical organization for the index is first presented in [77].

Chapter 5

LCA-based Selection for Distributed XML Document Collections

In this chapter, we define a variation of the multi-level Bloom filter and use it so as to address the problem of database selection for distributed XML document collections. That is, given a user query and a set of XML document collections, we provide a ranking of the collections according to their usefulness to the user's query.

We deal with the problem by: supporting keyword queries over XML documents but ranking their results according to their structural properties. More specifically we adopt the lowest common ancestor semantics of the keywords ([140],[139]) to estimate how similar a document is to a query (Section 5.1). We define the usefulness of a collection by aggregating over the similarity of each document to a query and study two versions of the problem, a boolean and a weighted one (Section 5.2). Instead of evaluating the query over each document to estimate its similarity, we propose an approximate approach that estimates the structural properties of the lowest common ancestor of any keyword query based on information about the lowest common ancestors of pairs of keywords that appear in the documents (Section 5.3). We use the variation of the multi-level Bloom filters to

summarize this information and show how they can be used to evaluate the similarity of a document to a query and consequently to estimate the usefulness of a collection (Section 5.4). We present experimental results that illustrate the efficiency and accuracy of our approach (Section 5.5) and conclude with a brief summary (Section 5.6).

5.1 Preliminaries

Database selection is a very important problem gathering increasing attention from the research community [21, 53, 24, 116, 141, 133]. For many queries, we can identify databases that are more suitable than others, that is, they maintain the most relevant results and the queries would benefit if they were evaluated over these databases. Thus, the problem of database selection for a number of databases and a user query, is defined as providing a ranking of the databases according to their usefulness (goodness to the user's query).

5.1.1 Database Selection for Relational and Textual Data

The problem of database selection has been mainly addressed for text and relational databases. While the same basic ideas apply, XML introduces new challenges since determining the similarity of a document to a query cannot be addressed by the methods deployed in the other models.

In [53], the authors address the text-source discovery problem. The approach used the vector-space model for its documents and maintains statistical summaries of each document collection which is extracted automatically and maintains information such as the tf/idf frequencies of the terms in the collection. Queries are evaluated over the summaries to produce the goodness estimation according to the same model. A similar approach is followed in [21], where inference networks based on information retrieval principles are used to estimate the goodness of a document collections. Both approaches consider text documents and are unsuitable for XML data for which the vector space model also not appropriate and would result in a loss in the expressiveness of the queries.

In contrast, in Kite [116], keyword queries are deployed against relational data. This approach is closer to ours but in our case the underlying data is semi-structured and not fully structured. The main challenge in this case is combining tuples from heterogeneous tables to form the query answer, unlike our approach in which we consider that each results belongs only to a single document and try to estimate how specific and how many these results are. Thus, Kite focuses on combining schema matching and structure discovery techniques to find approximate foreign-key joins across heterogeneous databases. In [141], summaries are built on top of relational databases to evaluate the goodness of the database against query keywords. In this work, the focus is on identifying meaningful connections among the keywords of the query. The approach uses SQL statements to define the meaningful relationships between keywords and is based on the number of joins required to combine the tuples that hold the respective keywords. This approach is similar to ours

in which in lack of a relational schema and SQL, we try to define the relationship of the keywords through their lowest common ancestor and then use summarizing structures upon which to evaluate the goodness of the collection.

5.1.2 Keyword Queries for XML and Lowest Common Ancestor Semantics

Dealing with XML data differentiates the problem considerably from database selection for text document collections. The main difference is that when querying XML documents, we are not only interested in their content but their structure as well. Though keyword querying is also a very popular and intuitive way for querying XML documents, the results returned also need to consider the position of the keywords in the document, i.e., its structure. Thus, to evaluate the relevance of a document to a query and consequently the usefulness of a collection of documents we should take into account both content and structure. Furthermore, we want to estimate this usefulness without having to actually evaluate the query against the collection.

Consider a conjunctive keyword query q consisting of M keywords $w_1, \ldots, w_M$ and an XML Tree $Tree = (V, E)$. The result of such a query are the elements in $Tree$ that contain all the keywords in q. We say that an element (node) v_i in $Tree$ directly contains a keyword w_i ($contains(v_i, w_i)$), if the keyword appears as the label of the element, one of its attributes, the value for one of its attributes, or in the element's content. An element v_i indirectly contains a keyword w_i, if any of its descendants directly contains w_i. We define a predicate *contains*(v,w)* which is true if the node v directly or indirectly contains the keyword w. Let $R_0 = \{v | v \in V \wedge \forall w \in q$ *(contains*(v,w))*$\}$ be the set of elements that directly or indirectly contain all of the query keywords. The result of the query q is defined below. $Result(q) = \{v | \forall w \in q, \exists y \in V, ((v, y) \in E \wedge y \notin R_0 \wedge$ *contains*(y,w))*$\}$ ([56]). $Result(q)$ thus contains the set of elements that contain at least one occurrence of all query keywords, after excluding the occurrences of the keywords in sub-elements that already contain all query keywords. The intuition is that if a sub-element already contains all of the query keywords, it (or one of its descendants) will be a more specific result for the query, and thus should be returned in the place of the parent element. The above definition ensures that only the most specific results are returned for a keyword search query.

The function $lca(v_1, \ldots, v_M)$ computes the Lowest Common Ancestor (LCA) of nodes $v_1, \ldots, v_M$. The LCA of sets $LS_1, \ldots, LS_M$ is the set of LCAs for each combination of nodes in LS_1 through LS_M. $lca(LS_1, \ldots, LS_M) = \{lca(v_1, \ldots, v_M) | v_1 \in LS_1, \ldots, v_M \in LS_M\}$. A node v is called an LCA of sets $LS_1, \ldots, LS_M$ if $v \in lca(LS_1, \ldots, LS_M)$.

Definition 5.1 (Exclusive Lowest Common Ancestor)**.** A node u is called an Exclusive Lowest Common Ancestor (ELCA) of $LS_1, LS_2, \ldots, LS_M$, if and only if there is $v_i \in LS_i, \ldots, v_M \in LS_M$, such that: $u = lca(v_1, \ldots, v_M)$ and for each v_i, $1 \leq i \leq M$, the child of u in the path from u to v_i is not an LCA of $LS_1, \ldots, LS_M$ itself nor an ancestor of any

LCA of $LS_i, \ldots, LS_M$.

We define as $elca(v_1, \ldots, v_M)$ the function that computes the exclusive common ancestor of nodes $v_1, \ldots, v_M$ and similarly to lca we extend its definition to sets of nodes $LS_1, \ldots, LS_M$. If we consider that each set LS_1 to LS_M corresponds to the different nodes in an XML tree $Tree$, which directly contain the keywords w_1 to w_M of q respectively (i.e., $LS_i = \{u_j \in Tree | contains(u_j, w_i)\}$), then according to the definitions, any node $u \in Result(q)$ iff $u \in elca(LS_1, LS_2, \ldots, LS_k)$.

Similarly to previous work [56], we consider keywords that are contained into more specific elements as being more closely related. For example, two keywords that appear in the same chapter of a book are more related than two keywords that appear in the same book, but in different chapters. In XRank [56], the evaluation of the rank for a result took into account the distance of each keyword from their ELCA by including a decay factor whose value increased with that distance. In our approach, instead of taking the distances of all keywords from the ELCA, we take the maximum of all such distances which is the one that determines the height of the subtree that has the ELCA node as a root and the nodes directly containing the keywords as its leaves. In particular, we define the height, h, of each node $u \in Result(q)$ as:

$$h(u) = \max_i dist(u, v_i), \tag{5.1}$$

where v_i $(1 \leq i \leq M)$ is a node that directly contains keyword w_i, and $dist$ is the length of the path between the two nodes in the XML tree. The height of the result measures the height of the subtree that contains all the keywords in query q. Thus, the lower the height, the more closely connected the keywords in the XML tree, that is, they are contained into a more specific element (i.e., a shorter subtree). For example, in Fig. 5.1, for the keyword query (*b, o*), we have two subtrees, one with height 2 (located under the left most *a* element) and one with height 1 (located under the second from the left *a* element).

When using the height of the result to measure the quality of the returned result, one may want to take into account the maximum depth of the XML tree this result belongs to. In particular, if we are dealing with trees of large depth then a result with larger height might still be specific, since it represents only a small percentage of the whole tree. In contrast, in shallow trees, even a result with small height may represent a large portion of the XML tree. Depending on the semantics one wants to convey, when comparing results that are taken from trees with different maximum depths, one might need to normalize the height of the result with respect to the maximum depth.

5.2 Problem Definition

The problem of database selection over collections of XML documents is defined as:
Given N collections of XML documents $(D_1, D_2, \ldots, D_N)$ and a keyword query q, return a list of ranked collections according to their *usefulness* to q.

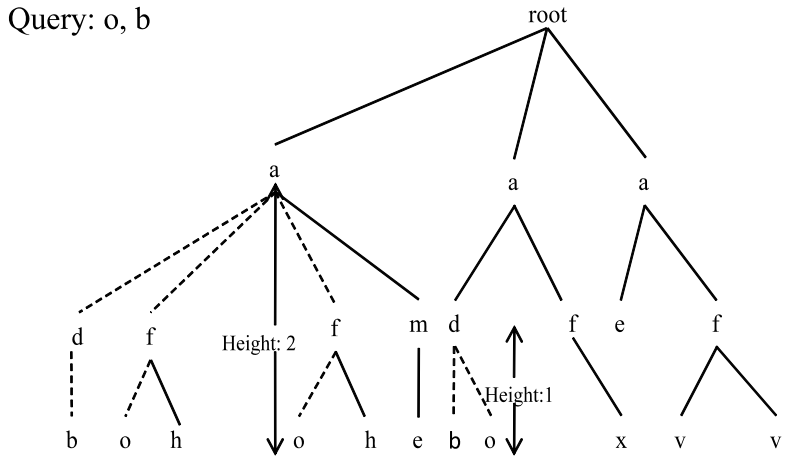


Figure 5.1: An example XML tree

We define the usefulness/goodness of an XML document collection to query q as:

$$Goodness(q, D, \lambda) = \sum_{d \in D} sim(q, d, \lambda), \tag{5.2}$$

where $sim(q, d, \lambda)$ determines the relevance of a single document to a query, and λ is a user-defined threshold that signifies that a user is interested in a document d as a result to her query q, only if its similarity with the query is greater than λ.

We present two versions of the selection problem. A boolean and a weighted one, which differ in the way the similarity of a document to a query is defined.

For the boolean version: A document is considered to *match* a query if there exists at least one subtree in its tree $Tree$ with height lower or equal to λ that contains all keywords in q.

$$sim(q, d, \lambda) = \begin{cases} 1 & \text{if } \min_{u \in Result(q)} h(u) \leq \lambda \\ 0 & \text{otherwise} \end{cases}$$

Thus, the goodness of a document collection is defined according to Eq. (5.2) as the number of matching documents in the collection.

For the weighted version: In the weighted version, besides determining whether a document contains all keywords under an ELCA node of height λ, we also take into account the height of that ELCA node, and determine the similarity of a document with a query depending on a function of the ELCA height, F. This function is such that the similarity is greater when the ELCA node containing all query keywords is of smaller height (i.e., the inverse of the node's height). Thus, giving greater similarity scores to documents that contain all query keywords within more specific elements.

$$sim(q, d, l) = \begin{cases} F(\min_{u \in Result(q)} h(u)), \text{if } \min_{u \in Result(q)} h(u) \leq \lambda \\ 0, \text{otherwise} \end{cases}$$

The goodness of a collection is evaluated this time by accumulating the similarity scores of every document.

5.3 Pairwise-based LCA Estimation

To evaluate a keyword query q against an XML tree, the straightforward approach is to apply an algorithm to find the ELCA of the M keywords that appear in q, and if its height is lower or equal to λ, then under the boolean model return a match. Since there may be multiple occurrences of each keyword in the XML tree, there may be more than one such ELCA node. Thus, this approach requires the computation of the ELCA nodes for each query. Following a brute force method this procedure takes $O(ML|LS_1||LS_2|\dots|LS_M|)$ for a tree with maximum depth L and M keywords in q. Even the state of the art more elaborate algorithms only reduce this complexity to $O(M * L|LS|)$ [139], where LS is the set with the maximum number of elements from LS_1 to LS_M. If we want to avoid this step at execution time, we can precompute the heights of the ELCAs for every possible combination of keywords that appear in an XML document. With this information available, during query processing we would be able to provide faster answers.

However, the number of all possible combination of keywords is very large and its precomputation imposes a large overhead both on processing cost and on storage, since this information needs to be maintained. In particular, the number of the ELCAs that need to be computed for an XML document with Z keywords (non-discrete keywords) is:

$$\sum_{i=2}^{Z} \binom{Z}{i} = O(2^Z) \tag{5.3}$$

We claim that one does not need to compute and maintain the ELCA nodes of all the possible combinations of keywords. Instead, we rely on pairwise ELCAs to estimate the height of the ELCA for any set of keywords.

Property 5.1. Let $V = \{v_1, v_2, \dots, v_M\}$ be a set of nodes in an acyclic directed graph. Then, $h(lca(v_1, \dots, v_M)) = \max_{i,j} h(lca(v_i, v_j))$.
Proof. Let $u = lca(v_1, \dots, v_M)$ and there is $u' = lca(v_l, v_m)$ such that: $h(u') > h(u)$. However, u is a common ancestor of v_l, v_m and since $h(u') > h(u)$, then, u cannot be the common lowest ancestor of all $v \in V$. Thus, $h(lca(v_1, \dots, v_m)) \geq \max_{i,j} h(lca(v_i, v_j))$.

Let us now assume there is a $u'' = lca(v_l, v_m)$ such that $h(u'') = \max_{i,j} h(lca(v_i, v_j))$, and $h(u) > h(u'')$. Then, there is at least one pair of nodes (v_s, v_t) such that $lca(v_s, v_t) = u''$. But since the LCA with the largest height is u, there cannot be such a node. Thus, $h(lca(v_1, \dots, v_M)$=$\max_{i,j} h(lca(v_i, v_j))$.□

Based on Property 5.1, we show how we can estimate the height for the result for any query q. If the keywords in an XML document are discrete (i.e., each keyword appears only once), then due to Property 5.1, we can provide an exact estimation of the height of any keyword query if we maintain all possible pairs of keywords along with the corresponding height of their ELCA node, by taking the maximum height of all the ELCAs that are recorded for any pair of keywords in our query.

However, in most cases there are multiple occurrences of each keyword in an XML tree. Thus, for each pair of keywords we may have multiple appearances under ELCAs

of different heights. That is, each pair of discrete keywords (w_i, w_j) is associated with a sorted list $Hlist^{ij}$ of height values in no decreasing order, corresponding to the different ELCA nodes under which the pair appears in the document. Let us consider all the pairs of keywords corresponding to a keyword query q. We then define as $Hlist_{min}$ the maximum value of the minimum values of height of all pairs of keywords associated with q. That is: $Hlist_{min} = \max_{ij} Hlist^{ij}[0]$. Similarly we define $Hlist_{max}$ as $Hlist_{max} = \max_{ij} Hlist^{ij}[t]$, where t denotes the last element in the list for each pair of keywords.

Theorem 5.1. *The height of any $u \in Result(q)$ is such that: $Hlist_{min} \leq h(u) \leq Hlist_{max}$.*

Proof. We first prove that any $u \in Result(q)$ has a height $h(u) \geq Hlist_{min}$. Let us assume that there exists $v \in Result(q)$, such that $h(v) < Hlist_{min}$. Then, the first element in the ordered list of all keyword pairs associated to q should also be lower or equal to $h(v)$, since due to property 1, the height of a result node is determined by the maximum height value of all the node pairs. But $Hlist_{min}$ is defined as the maximum value of all first elements and therefore we have at least one pair such that its minimum value is greater than $h(v)$. Thus, there cannot be a result node with height lower to $Hlist_{min}$.

Let us now prove that $u \in Result(q)$ has a height $h(u) \leq Hlist_{max}$. Let us assume that there exists $v \in Result(q)$ such that $h(v) > Hlist_{max}$. Then, according to property 1, there is at least one pair of keywords associated with an ELCA node with a height value greater than $Hlist_{max}$, which is impossible since $Hlist_{max}$ is defined as the maximum value of all height values associated with any pair of keywords corresponding to q. Thus, there cannot be a result node v with $h(v) > Hlist_{max}$.□

According to Theorem 5.1, we can bound the height of any result between $Hlist_{min}$ and $Hlist_{max}$. Thus, if we maintain all the appearances of any pair of keywords in a list $PList$ with pairs of the form $(w_i, w_j), h(u)$, $u \in elca(LS_i, LS_j)$, then, we can provide an estimation for the height of the result of any keyword query q between $Hlist_{min}$ and $Hlist_{max}$. That is, we determine that the height of any result cannot be lower than $Hlist_{min}$ and higher than $Hlist_{max}$. For example, in Fig. 5.1, the height of the result for query (o,b) is bound between 1 and 2.

If for a query q, $Hlist_{min} > \lambda$, then we can safely deduce that there are not any results in our document that satisfy the similarity condition the user has set. Theorem 5.1 guarantees, that there are no false negative presents. Otherwise, if $Hlist_{min} \leq \lambda$, we deduce that the document is a match, though there is a probability that we might be wrong (false positive). That is, pairs of nodes belonging to different subtrees may be combined to give the false $Hlist_{min}$ estimation, while no such ELCA node containing all keywords actually exists. We can determine whether the result is certainly not a false positive if $Hlist_{max}$ is also lower or equal to λ. Then the document surely matches q according to the specified relevance threshold. However, if $Hlist_{max} > \lambda$, then the document may contain the keywords under an ELCA node of height greater than λ and we may not reach a definite conclusion. In this case, we deduce a match though we cannot determine if it is a false positive or not.

5.3.1 Query Evaluation

Based on the above observations, we define appropriate algorithms for evaluating keyword queries under both the boolean and weighted models based on the pairwise ELCA nodes. Our basic idea is that if we maintain for any pair of discrete keywords in a document the value of the height of the ELCA with the minimum height and the value of the ELCA with the maximum height among all of its occurrences, we can determine whether any keyword query matches a document under a given similarity threshold λ. Thus, we reduce the complexity of the preprocessing phase from $O(2^Z)$ to $\binom{Z}{2} = O(Z^2)$, both processing and storage wise.

Thus, for each document $d \in D$ we maintain a table $PList(d)$ with all pairs of discrete keywords and two columns recording the corresponding minimum ($minH$) and maximum height ($maxH$) of the ELCA nodes under which each pair appears. Since we know that any pair of keywords with $minH$ greater than λ does not contribute in any result node, we can safely omit such pairs from table $PList(d)$. Furthermore, we do not need to maintain the value of $maxH$ for any keyword pair that $maxH > \lambda$ and simply set that value NULL in the $PList(d)$ table.

We first describe how a query q is processed under the boolean problem.

The processing of q against a document $d \in D$ proceeds by extracting all possible keyword pairs from q and checking them against the $PList$ table. If any pair is not found in the table, then we set the similarity of the document to the query to 0 (q does not match d). Otherwise, we set the similarity equal to 1. We also check the $maxH$ column of the table to determine whether there is a possibility for a false positive and set the corresponding flag ($fpFlag$) accordingly. The algorithm is detailed in Alg. 13.

Boolean Keyword-Query Evaluation
Input: q: keyword query, d: XML document, λ: similarity threshold
Output: sim: similarity estimation, $fpFlag$: false positive flag

1: Extract all possible keyword pairs (w_i, w_j) from q
2: $PList(d)$ is examined against all pairs of extracted keywords
3: **if** any pair does not appear in the table **then**
4: $\quad sim(q, d, \lambda) = 0$
5: **else**
6: $\quad sim(q, d, \lambda) = 1$
7: $\quad$ **if** For any (w_i, w_j), $maxH{=}NULL$ **then**
8: $\quad\quad fpFlag{=}1$
9: **RETURN**$(sim, fpFlag)$

Algorithm 13: Boolean Keyword-Query Evaluation

To compute the goodness of a collection D of XML documents, the above algorithm is applied for all documents d in the collection. The goodness of the collection based on Eq. (5.2) is estimated as the sum of the similarity values for each document in the collection,

and an upper bound on the error of the goodness estimation can be estimated as the sum of the false positive flags set.

For the weighted version of the problem, we do not only determine whether a document matches or not a query, but we also evaluate a measure for its similarity. The similarity is determined as a function of the height of the ELCA node. As in the boolean version, since there may be several occurrences of the query keywords in a document to determine the similarity value, we rely again on the height of the ELCA with the minimum height and the height of the ELCA with the maximum height among all the ELCA nodes in our result set.

Thus, the table $PList(d)$ as we described it for the boolean version of the problem suffices for determining the similarity value for the weighted version of the problem as well. However, we cannot determine a bound on the error for the similarity value if we maintain the table as in the boolean version since no value larger than λ is maintained in the table. The table can still provide an estimation on the number of false positives, but it is not enough to provide an estimation on the maximum height of the ELCA node that can contain our query keywords. To enable the query processing to provide such a bound on the similarity value, we alter the information we maintain in the table by inserting in the table for all keyword pairs with $Hlist_{min} < \lambda$ its $Hlist_{max}$ value even when $Hlist_{max} > \lambda$. Then, the if we consider the ELCA node with height $maxH$ as our result node, we can provide a worst case bound for the similarity ($wsim$) of the document to query q. The detailed procedure for the evaluation is presented in Alg. 14.

Weighted Keyword Query Evaluation
Input: q: keyword query, d: XML document, λ: similarity threshold
Output: sim: similarity estimation, $wsim$: worst case similarity

1: Extract all possible keyword pairs (w_i, w_j) from q
2: $PList(d)$ is examined against all pairs of extracted keywords
3: **if** any pair does not appear in the table **then**
4: $\quad sim(q, d, \lambda) = 0$
5: **else**
6: $\quad sim(q, d, \lambda) = F(minH)$
7: $\quad wsim(q, d, \lambda) = F(maxH)$
8: **RETURN**$(sim, wsim)$

Algorithm 14: Weighted Keyword Query Evaluation

To evaluate the goodness of the entire collection D, we apply the query processing algorithm to each document and sum the similarity values. Furthermore, by summing also the worst case similarities estimations we provide an upper and lower bound on the goodness of the collection. In particular, the upper bound on the goodness value is the sum of the similarity values whose estimation was based on the $minH$ and the lower bound is the sum of the similarity values whose estimation was based on $maxH$.

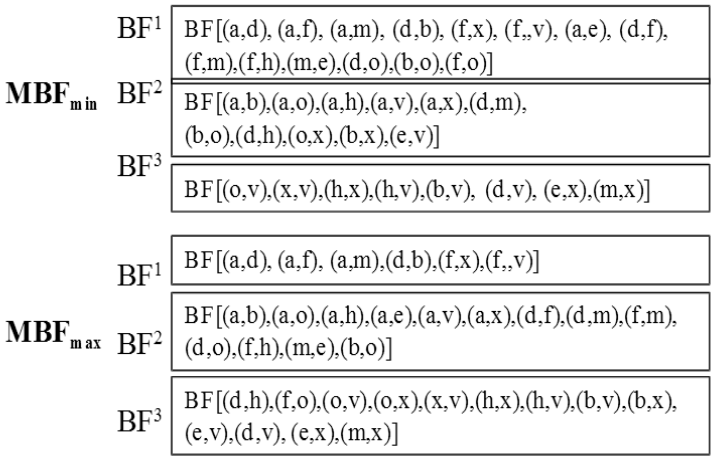


Figure 5.2: The MBFs corresponding to the XML tree in Fig. 5.1.

5.4 Index-based Evaluation

Instead of maintaining the information about the keyword pairs for every document in a table, for storage and processing efficiency, we summarize this information using Bloom filters and Multi-level Bloom filters. In particular, Bloom filters are deployed for the boolean problem while multi-level Bloom filters are required for the weighted problem.

In the boolean problem, we replace the $Plist$ table with Bloom filters. In particular, we maintain two Bloom filters for each document d in our collection, $BF_{min}(d)$ and $BF_{max}(d)$ corresponding two the two columns of the $PList(d)$ table that maintained the $minH$ and $maxH$ value for each keyword pair. Given a similarity threshold λ, we construct the two Bloom filters similarly to table $PList(d)$. Any keyword pair for which the minimum height value associated with an ELCA node is lower or equal to λ ($minH \leq \lambda$) is hashed as one key and inserted into $BF_{min}(d)$. Similarly, any keyword pair for which the maximum height value associated with its ELCA nodes is lower or equal to λ ($maxH \leq \lambda$) is also inserted into $BF_{max}(d)$.

To process a query, first every pair of keywords is checked against $BF_{min}(d)$ and if there are no misses, the similarity is set to 1 and then $BF_{max}(d)$ is also checked to identify any possible false positives (Alg. 15).

To compute the goodness for an XML collection the query processing procedure is applied for each document d in the collection on the corresponding Bloom filters $BF_{min}(d)$ and $BF_{max}(d)$. The sum of the similarity values of all documents is the goodness estimation for the collection. Note that this time the sum of false positive flags over all the collection only gives an estimation of the false positives present and not an upper bound. This is because the use of the Bloom filters introduces additional false positives due to the hash function collisions.

Bloom filters cannot be directly used as an indexing structure for the weighted version of the problem since they are unable to maintain the value of the height of the ELCA nodes which is required to determine the similarity value. Instead, we deploy the multi-level Bloom filters.

Bloom-Based Boolean Keyword-Query Evaluation Input: q: keyword query, d: XML document, λ: similarity threshold
Output: sim: similarity estimation, $fpFlag$: false positive flag

1: Extract all possible keyword pairs (w_i, w_j) from q
2: **for all** $(w_i, w_j) \in q$ **do**
3: Apply the Bloom filter's hash functions to (w_i, w_j)
4: Lookup (w_i, w_j) in $BF_{min}(d)$
5: **if** there is a miss **then**
6: $sim(q, d, \lambda) = 0$
7: **RETURN**(sim)
8: $sim(q, d, \lambda) = 1$
9: **for all** $(w_i, w_j) \in q$ **do**
10: Lookup (w_i, w_j) in $BF_{max}(d)$
11: **if** there is a miss **then**
12: $fpFlag = 1$
13: **RETURN**$(sim, fpFlag)$
14: $fpFlag = 0$
15: **RETURN**$(sim, fpFlag)$

Algorithm 15: Bloom-Based Boolean Keyword-Query Evaluation

In particular, instead of inserting all pairs of keywords with ELCA nodes with $Hlist_{min} \leq \lambda$ in a single Bloom filter as in the boolean version, we group the keyword pairs of each document according to their $Hlist_{min}$ value and we use a separate Bloom filter for each such group. Thus, we construct a multi level Bloom filter MBF_{min}, which is a set of simple Bloom filters $BF^1, BF^2, \ldots, BF^\lambda$ such that all pairs of keywords with $Hlist_{min} = 1$ are hashed into BF^1, with $H_{min} = 2$ into BF^2 and so on until $H_{min} = \lambda$. Again, we are not interested in any pair for which $H_{min} > \lambda$ since such pairs do not contribute to any query results according to our similarity definition.

Similarly to the MBF_{min} we can extend the BF_{max} to MBF_{max}. However, in this case if we want to provide an estimation for the worst case similarity as we described when we presented the weighted problem based on table $PList$, maintaining only λ levels in our filter is not enough. In this case, the number of levels in the filter is at most equal to the depth of the XML tree, since the highest value for the height of any ELCA node in the tree is at most equal to the XML tree. Figure 5.2 shows the MBF_{min} and MBF_{max} for the XML tree of Fig. 5.1(a) and $\lambda = 3$.

If we are not interested in providing a bound in the similarity measures, then a simple BF_{max} suffices for providing a false positive estimation.

Let us consider that we are interested in giving both a lower and an upper bound for our goodness estimation. Then, for each document $d \in D$ we maintain two multi-level Bloom filters, an MBF_{min} with λ levels and an MBF_{max} with *depth* levels, where *depth* is the maximum depth of the XML tree corresponding to d. For each query, we first examine

the MBF_{min} and maintain for each pair of keywords (w_i, w_j) the higher level at which we found a match (t'). If a pair cannot be found in any level, then we set similarity to 0. Otherwise, the highest level (t)in which we had a match for any of the keywords pairs is used to estimate the similarity. A similar procedure is followed then against the MBF_{max} to estimate the worst case similarity. The query evaluation procedure is presented in detail in Alg. 16.

After the evaluation algorithm is applied for all documents we sum the returned best and worst case similarities over all the documents in the collection and determine the corresponding best case and worst case goodness.

5.5 Experimental Evaluation

We evaluate the performance of our approach based both on a real dataset to show its effectiveness and on a synthetic dataset so as to study how the different parameters and characteristics of the data influence performance.

5.5.1 Real Datasets

In the first set of experiments, we evaluate our approach on real data. We use the DBLP bibliographic data collection. We split the DBLP data to a number of collections according to two criteria. First, we split the entries that correspond to publications according to the year they were published and create a different collection for each year. That is, each collection corresponds to a selection of the publication entries in the DBLP data according to their year. Then, we split the publications of each collection into different documents according to the conference they were published to, performing another selection operation on the conference field. For example, in the collection "2005" corresponding to year of publication 2005, we have an XML document with the publications in VLDB 2005, one for the ones in ICDE 2005, and so on. We maintained only the publications in conferences omitting the ones in journals. In our second experiment, the DBLP data is first split into collections according to the name of the conference they appeared at, and then into documents according to the year of their publication. For example, there is a "VLDB" collection with documents corresponding to VLDB 2008, VLDB 2007, etc.

The goal of this set of experiments is to demonstrate the usefulness of the LCA-based method compared to an approach based solely on the appearance of the query keywords in a document. To this end, we compare our approach both with using the full table of the LCA pairs and with using the Bloom filters to a simple approach that only records the appearance of the keywords in the documents and their respective frequencies (number of appearances of each keyword in the document). We pose queries using author names as our keywords and report our results. Note here that according to the format of the DBLP data, if author 'X' is a coauthor with author 'Y' in an article, then the minimum height of their ELCA nodes is 1, while if they are authors in different articles then the

Bloom-Based Weighted Keyword-Query Evaluation
Input: q: keyword query, d: XML document, λ: similarity threshold
Output: sim: similarity estimation, $wsim$: worst case similarity

```
1: t = 0
2: Extract all possible keyword pairs (w_i, w_j) from q
3: for all (w_i, w_j) ∈ q do
4:    Apply the Bloom filter's hash functions to (w_i, w_j)
5:    for all BF^k in MBF_min from k = λ to k = 1 do
6:       t' = 0
7:       Lookup (w_i, w_j) in BF^k
8:       if there is a match then
9:          t' = k
10:         BREAK
11:   if t' = 0 then
12:      sim(q, d, λ) = 0
13:      RETURN(sim)
14:   else
15:      if t < t' then
16:         t = t'
17: sim(q, d, λ) = F(t)
18: t = 0
19: for all (w_i, w_j) ∈ q do
20:   for all BF^k in MBF_max from k = depth to k = 1 do
21:      t' = 0
22:      Lookup (w_i, w_j) in BF^k
23:      if there is a match then
24:         t' = k
25:         BREAK
26:   if t < t' then
27:      t = t'
28: wsim(q, d, λ) = F(t)
29: RETURN(sim, wsim)
```

Algorithm 16: Bloom-Based Weighted Keyword-Query Evaluation

minimum height is 2. We evaluate our approach under the boolean model and setting λ equal to 1. Next, we report some of our results.

In our first experiment, we pose the query "Omar Benjelloun and Serge Abiteboul" against the document collections split by year. That is, we are interested in articles that were co-written by the two authors. According to the DBLP data, the two had the most publications together (excluding journals) in 2004 with 4, followed by 2002 with 3, 2003 with 3, and 2005 with 1. Thus, the correct order for the collection selection problem is 2004, 2002, 2003 and 2005, while all other collections contain no results of interest. By applying our approach first based on a full table we retrieved the collections in this exact order, i.e., with a 100% accuracy. By using the Bloom filters, the collections that were returned where: 2004, 2002, 2005, 2003 and 1998. The errors in this case were due to false positives. We note that one false positive is enough for including 1998 as the last element in this list although it does not actually contain any results of our interest. Finally, we applied the keyword-based approach. In this case since we do not take into account the structure of the documents and just count the appearances of the keywords the order we retrieved was very different. In particular, the first two collections were the ones corresponding to years 2006 and 2007 in which both authors had a lot of publications and many of them in common conferences, but not as co-authors. The collection corresponding to 2004 came just third, followed by 2002, 2003 and 2005. It was evident through this example that an approach solely counting on the frequencies of the keyword queries is not adequate to capture the semantics hidden in the XML document structure.

We performed a second experiment to the collections split by conference. We are interested in articles by "Alon Y. Halevy and Zachary G. Ives". The authors have the most articles together in SIGMOD (6 articles) followed by several venues such as WebDB, WWW, CIDR, ICDE where they have one article together. Note again that all journal articles are excluded from our results. Both our approaches and the keyword based approach are able to identify SIGMOD as the collection with the most articles of interest. The LCA based approach with the full list returns then all 4 other venues with the same similarity score. The Bloom-based approach returns ICDE as second, followed by WebDB, CIDR, WWW, and lastly VLDB due to false positives. In contrast, the keyword-based approach returns the collection corresponding to VLDB as second, since both authors have many publications in this conference but none in common. Then, the keyword-based approach returns ICDE, WebDB, CIDR and WWW in this order. Again, we can see the inability of the keyword approach to capture the XML structure semantics.

5.5.2 Synthetic Data Sets

To evaluate the quality of the goodness estimation our approach provides and examine how this estimation is influenced by the various input parameters in our problem, we perform a set of experiments with synthetic data with varying properties. In particular, we compare the estimation provided by four approaches. The first approach is the *tree-based* approach that evaluates the goodness of a collection by evaluating the query against

Table 5.1: Input parameters for the evaluation of XML document collections selection

Parameter	Range	Default Value
number of documents per collection ($\|D\|$)	20-200	100
number of elements per document (Z)	-	50000
maximum depth of XML tree ($depth$)	4-20	12
percentage of repeating element names (r)	0-0.6	0.3
query length (l_q)	1-6	4
relevance threshold (λ)	1-12	4
number of collections (N)		
number of Bloom filter hash functions		4
size of Bloom filter		996 bits

each document in the collection. That is, the tree-based approach evaluates the exact value of the goodness for a collection. The second approach is the *keyword-based* one. This approach ignores the structure of the XML document and evaluates the queries based only on the appearance of the keywords if we consider the document as flat text. The third approach, the *pair-based* one, is our approach that estimates the similarity of a document to a query based on the pairwise ELCAs of the keywords in the query. Finally, the fourth approach is the one that uses Bloom filters to summarize the pairwise ELCA information (*bloom-based*).

We use the Niagara generator [101] to generate our data. The number of elements of the XML tree remains fixed at 50000 and we use different tree-depths and percentages of discrete element names to variate the produced trees structure. The queries are generated with 90% of the keywords belonging to the documents and 10% random ones. To examine the influence of each parameter, we vary its value and set the rest of the parameters to their default value.

Goodness Estimation

In general, our approach provides a very accurate estimation for both the boolean (Fig. 5.3-Fig. 5.4) and weighted (Fig. 5.5-Fig. 5.6) approaches. The keyword-based approach in most cases overestimates the goodness of the collection, since most keywords appear in the collection. While, our approach also overestimates the goodness since it is based on the minimum height values that are maintained for all keyword pairs, the estimation error is significantly smaller than in the keyword-based case. The Bloom-based approach increases the estimation error slightly due to the addition of false positives besides the height estimation.

The weighted-problem has lower goodness values than the boolean version because of the weight of the height. The keyword-based approach is not considered in this case since it ignores structure. Considering tf/idf as the appropriate weight does not provide comparable results with respect to our approach that weights the documents based on

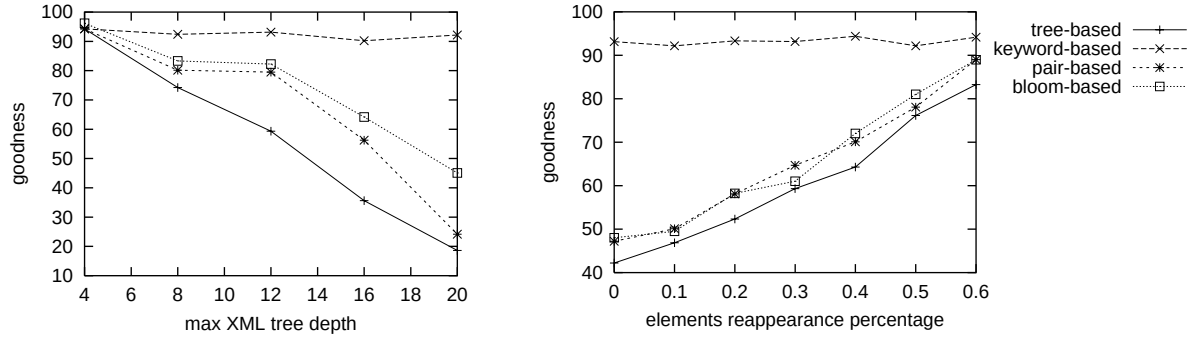


Figure 5.3: Goodness estimation for the boolean problem with varying (left) XML tree depth and (right) percentage of repeating elements

the inverse of the ELCA height.

Maximum XML Tree Depth. For tree-depths smaller or only slightly larger to the relevance threshold, all approaches provide the best estimations compared to larger trees (Fig. 5.3(left)). Still, our approach is still satisfying even for larger depths, where in the worst case we have for depth=16, an estimation error of about 25% for the pairwise approach and 29% for the Bloom-based one. The keywords-based approach is not affected by the depth of the tree. If the tree depth increases even further, since we keep the number of elements fixed, due to the small fan-out of the tree, our approach again behaves better.

Percentage of Repeating Elements. Our approach is 100% accurate when we have no repeating element names and its performance decreases as the percentage of repeating elements increases (Fig. 5.3(right). Again, after a point the performance again starts to improve because the keywords of the query have greater probability of at least once appearing closer to the document. Again, the keyword-based approach is not significantly affected.

Query Length. As the query length increases, our approach behaves better because it considers more pairs to evaluate the ELCA (Fig. 5.4(left)). For a single-keyword query, all approaches except the Bloom-based one provide accurate estimations.

Similarity Threshold. The similarity threshold affects mostly our approach with respect to the XML tree depth. When the value is small, then our approach provides better estimations (Fig. 5.4(right)). For values closer to the tree-depth our estimation again improves since most documents are considered matches.

Lower Bound Estimation. The keyword based approach has no way to estimate a lower bound to the estimation error it provides so the lower bound is the same as the estimation. The tree-based approach is included as it produces the exact value so we can see how much the lower bound differs from it. We see that our lower-bound is usually tight (Fig. 5.7, Fig. 5.8, Fig. 5.9, Fig. 5.10). The Bloom-based approach usually presents a more optimistic lower bound estimation due to false positives introduces by the structure, which is around 10%. This error decreases for larger percentages of repeating elements. Thus, though the approach introduces more errors, it usually provided a better estimation than the pair-wise based approach.

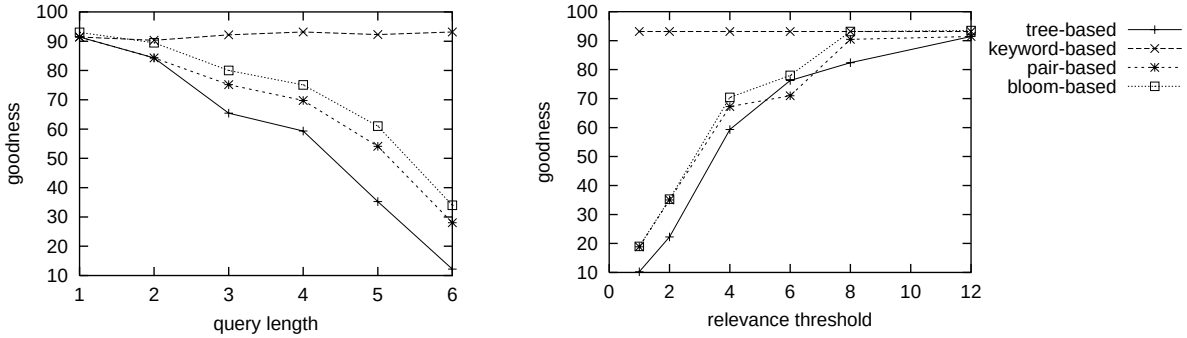


Figure 5.4: Goodness estimation for the boolean problem with varying (left) query length and (right) relevance threshold

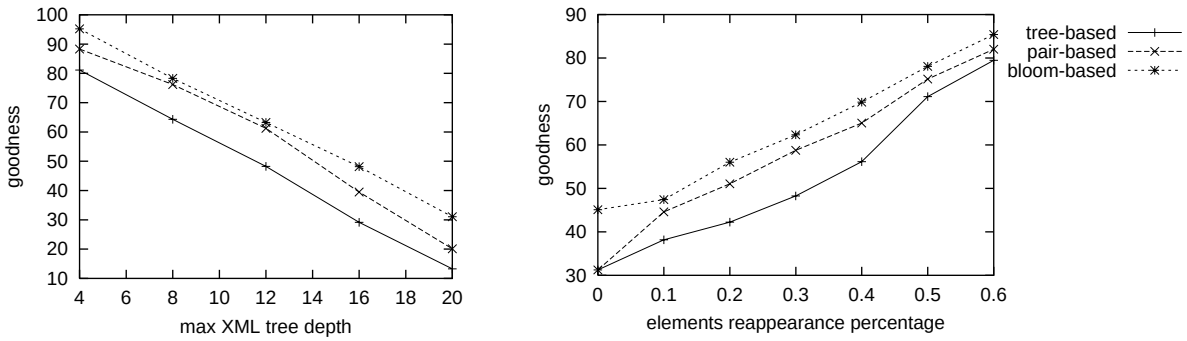


Figure 5.5: Goodness estimation for the weighted problem with varying (left) XML tree depth and (right) elements reappearance percentage

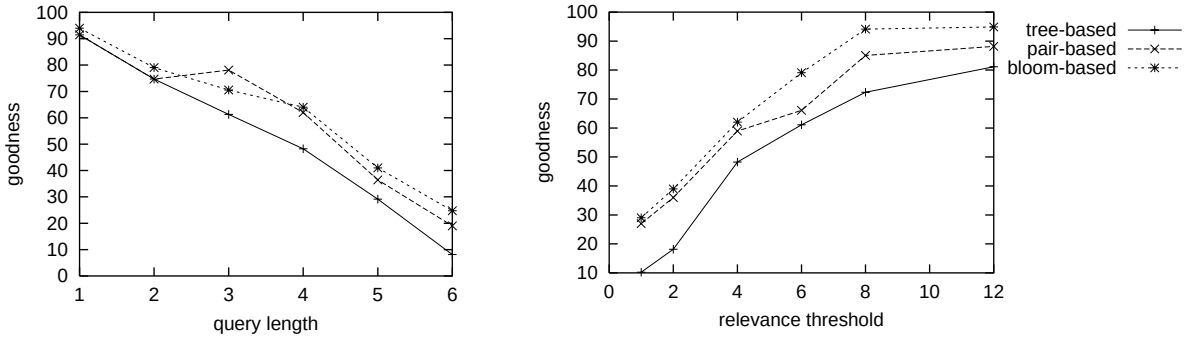


Figure 5.6: Goodness estimation for the weighted problem with varying (left) query length and (right) relevance threshold

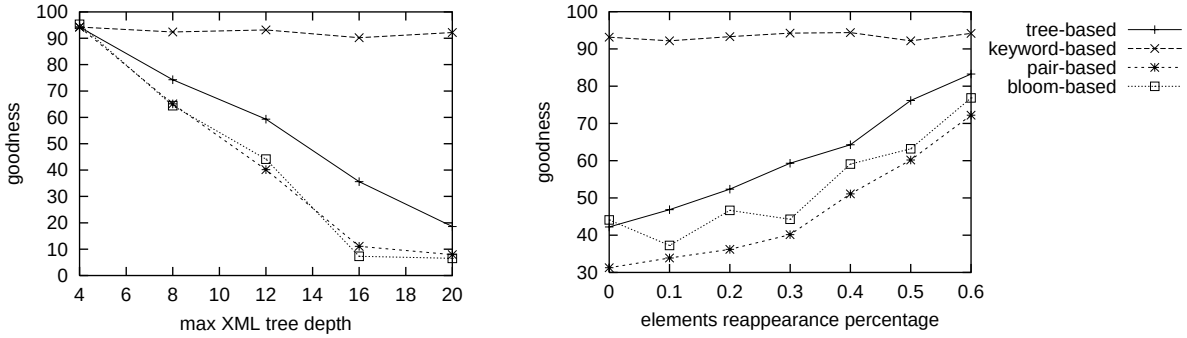


Figure 5.7: Lower bound for goodness for the boolean problem with varying (left) XML tree depth and (right) elements reappearance percentage

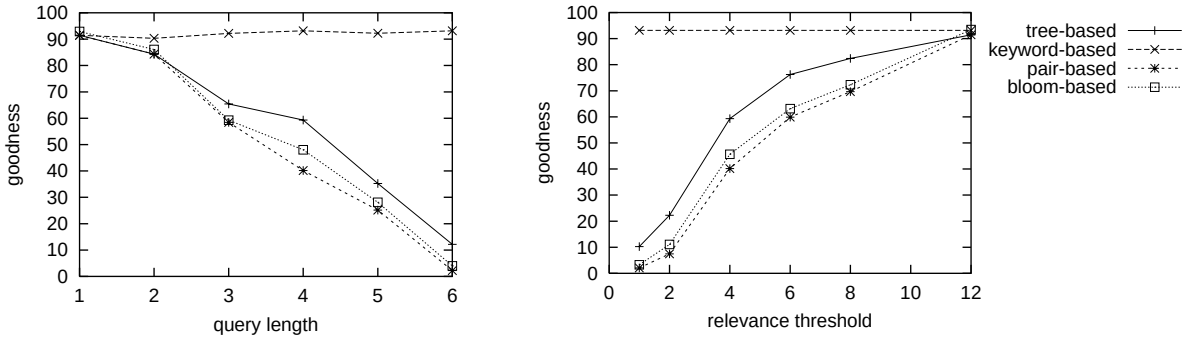


Figure 5.8: Lower bound for goodness for the boolean problem with varying (left) query length and (right) relevance threshold

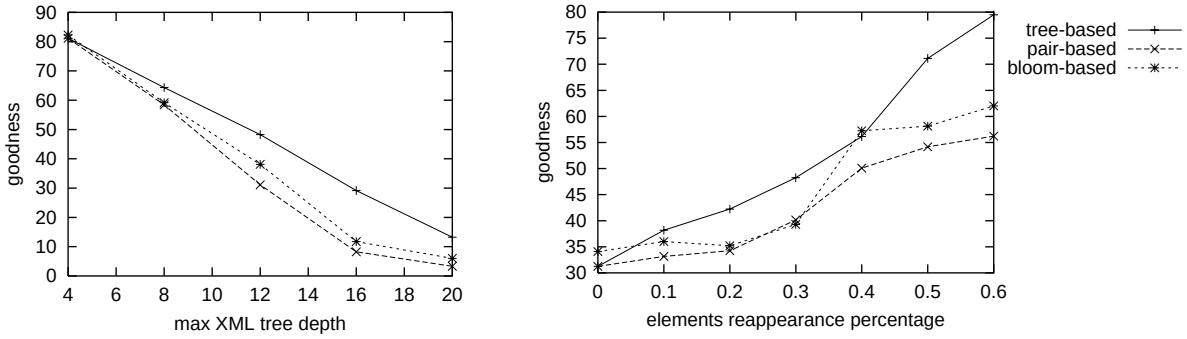


Figure 5.9: Lower bound for goodness for the weighted problem with varying (left) XML tree depth and (right) elements reappearance percentage

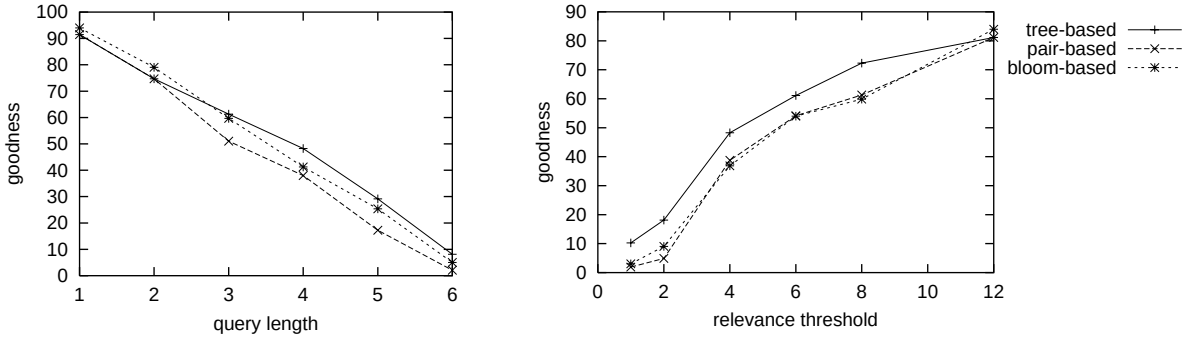


Figure 5.10: Lower bound for goodness for the weighted problem with varying (left) query length and (right) relevance threshold.

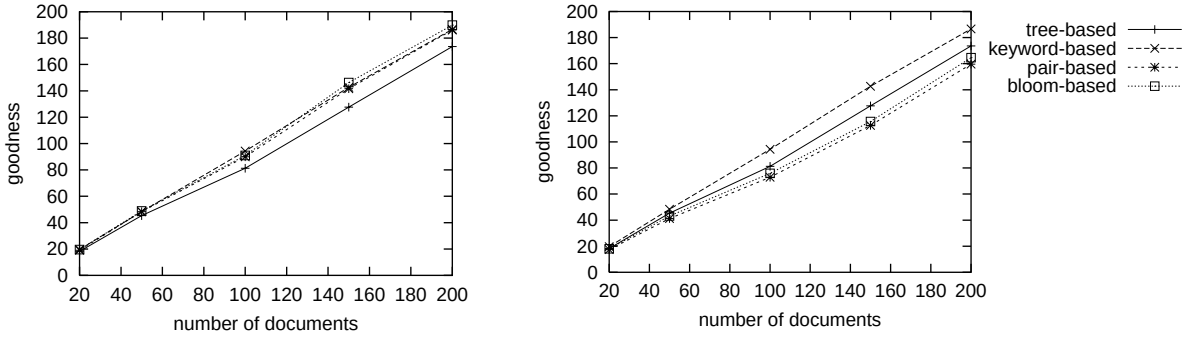


Figure 5.11: Scalability for (left) goodness estimation and (center-left) bound estimation for the boolean problem.

Scalability

We vary the number of documents per collection from 20 to 200 and measure the goodness estimation and the lower-bound estimation for all our approaches. Our approaches scale gracefully for larger number of documents with the estimation error being about 20% at the most (Fig. 5.11-Fig. 5.12). The Bloom based approaches behave a bit worse for larger collections also due to an increase of the false positives. Allocating larger sizes for the Bloom filters would alleviate this problem.

Database Selection

In this experiment, we evaluate how our goodness estimation approach is applied in a database selection problem. In particular, we consider 12 collections of documents and compare the keyword-based, the pairwise based and the Bloom-based approach with respect to the ranking they derive for the 12 collections. To measure the quality of their ranking we evaluate for each of them the Spearman Footrule distance to the ranking provided by the tree traversal based approach, which is also the correct ranking for the collections. The Spearman Footrule distance between two ranked lists is defined as the absolute difference of their pairwise elements and is normalized by dividing with $1/2(S)$, where S is the number of elements in each list.

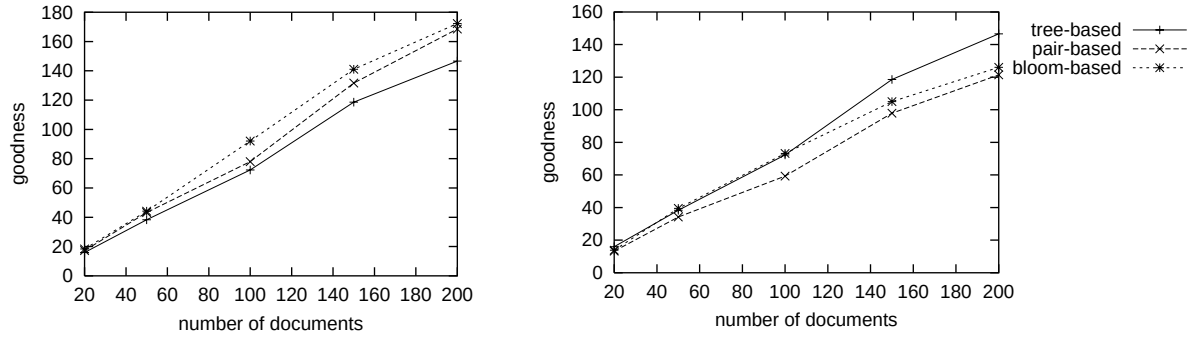


Figure 5.12: Scalability for (left) goodness estimation and (center-left) bound estimation for the weighted problem.

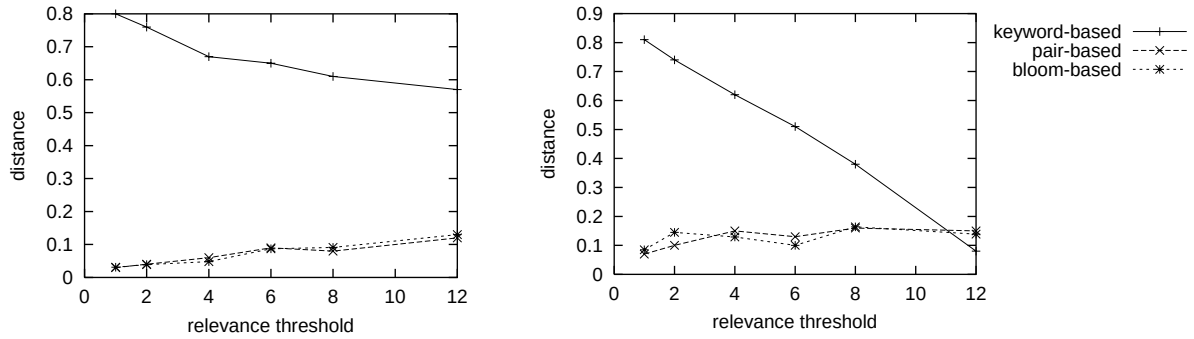


Figure 5.13: Boolean database selection with respect to λ, for collections of (left) equal and (right) different size.

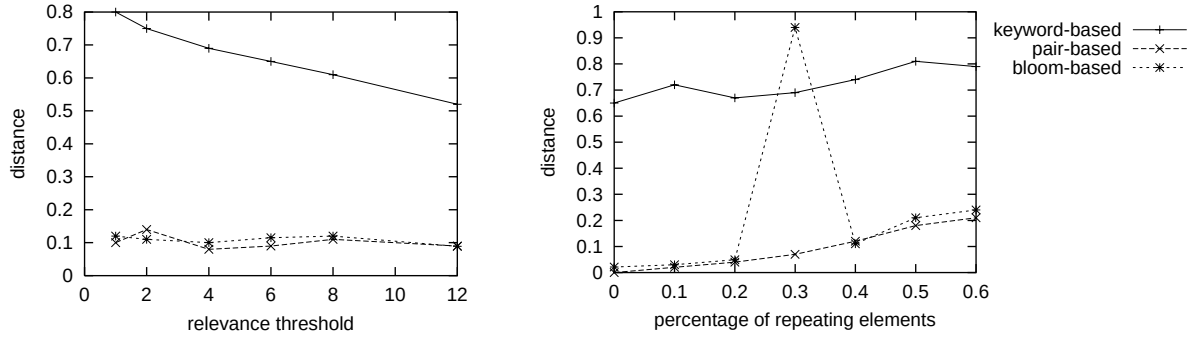


Figure 5.14: Boolean database selection for random collections with respect to (left) λ and (right) percentage of repeated elements.

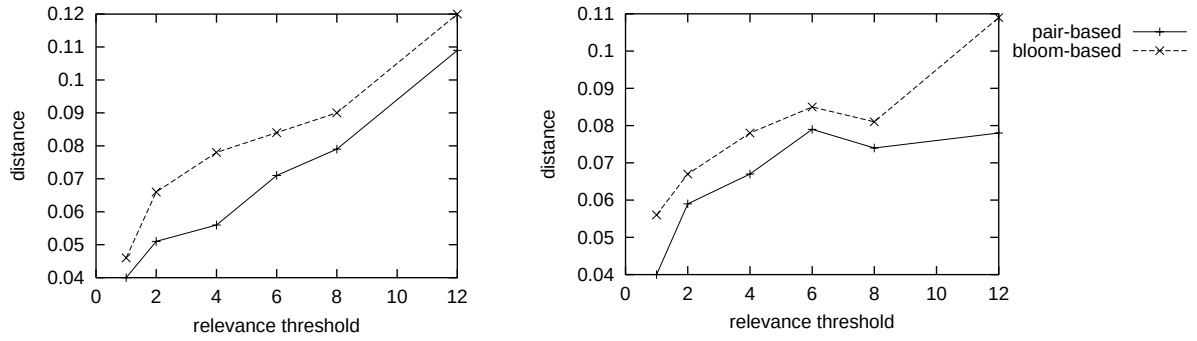


Figure 5.15: Weighted database selection with respect to λ, for collections of (left) equal and (right) different size.

The collections where constructed as follows: for a given query keyword, we produced a collection in which each document has at least one result with height 1, another collection in which each document has at least one result with height 2, and so on. Each document is constructed with the default parameters values.

We first consider collections with an equal number of documents. We evaluate the performance for varying λ (Fig. 5.13(left)). We also consider collections of different size (Fig. 5.13(right)). In particular, we set the collection with the largest ELCA height result we constructed (the 12th collection) as the one with the most documents, and decrease the size of the collection as the ELCA height decreases. That is, the collection with the results with ELCA equal to 1, thus, the most relevant collection, is the one with the smallest size. Finally, we include a set of random collections as constructed in the first set of experiments and evaluate for varying λ and varying percentage of repeated elements (Fig. 5.14).

The keyword based approach has the worst overall performance since its goodness estimation ignores structure, thus all collections are regarded as equally relevant despite their different structural properties. We observe that it approximates the real ranking best when collections of different sizes are regarded. The real ranking in the boolean model favors collections with more documents among collections that satisfy the relevance threshold despite the actual value of the ELCA. The keyword based approach also favors collections of larger size (however ignoring λ), thus approximating the real ranking better in this case than in the case of equal sized collections.

The pairwise approach and the Bloom-based approximation provide a ranking very close to the real one. The pairwise-based approach has the best performance but the Bloom-based approach sometimes outperforms it (Fig. 5.13). As far as the size of the collections is concerned, the effects are the same as with the real ranking achieved by tree traversal if we use the given measure of goodness and similarity. We expect the weighted version that takes into account the ELCA heights to provide more interesting results. In this case, the real ranking orders all the collections with ELCA larger than λ according to their size, and give a 0 goodness estimations to the others. Our approaches except the keyword based one approximate this ranking really well. The Spearman distance is in

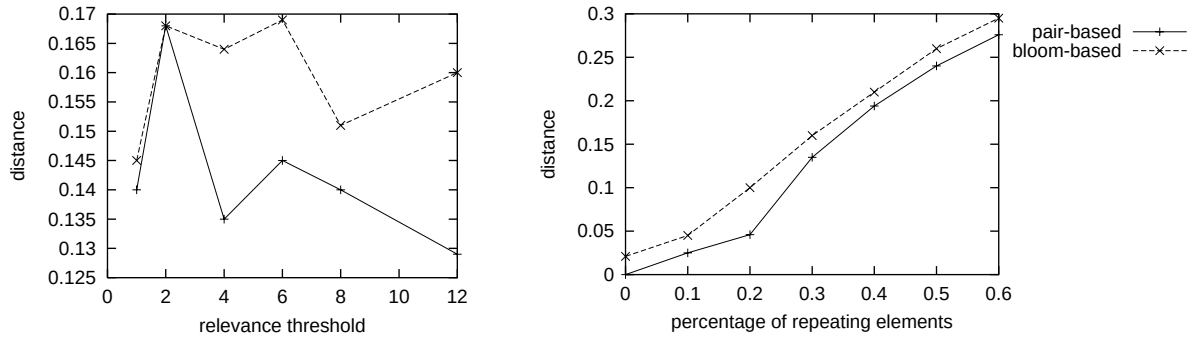


Figure 5.16: Weighted database selection for random collections with respect to (left) λ and (right) percentage of repeated elements.

most cases lower than 0.15.

The results regarding the weighted version of the problem are similar to the ones of the boolean version (Fig. 5.15, Fig. 5.16). All our approaches rank the databases very close to the real ranking for all cases (maximum observed distance 0.3). When the collections are of different size the performance is slightly worse than for equal size collections. This is because the goodness estimation is not as accurate. In the weighted version, large sized collections are not favored against smaller size ones with more specific ELCAs as results for the queries. The height of the ELCA acts as a normalizing factor and a compromise between the size of the collections and the accuracy of their results is accomplished. That is, for $\lambda = 12$ for example, the 5-6th collections that have ELCA with height 5,6 and are about average size among all are the ones ranked higher. also, we observe that the rankings in the case are closer to each other. That is, the goodness for the first collection is 89.5, while for the 12th 88.4. Finally, for the weighted scenario the random collections induce the largest estimation errors, since there are more errors with regards to the ELCA height estimation. The weighted version is more sensitive to such errors than the boolean one, that only needs to compare the heights to the relevance threshold.

5.6 Summary

In this chapter, we deal with the problem of database selection over distributed collections of XML documents. We consider keyword queries but evaluate the similarity of an XML document to a query based on the height of its ELCA node, thus considering for the similarity evaluation both the document structure and its content. We introduce a novel and very efficient way to approximate the height of the ELCA node of any keyword query by maintaining information about the ELCAs of pairs of keywords that appear in a document. Furthermore, we used a variation of the multi-level Bloom filter for maintaining this information reducing significantly the storage and processing cost required. Through our experiments on both real and synthetic data, we showed that our approximation methods are accurate and able to deal with the problem of database selection efficiently.

The approach presented in this chapter also appears in [78].

Chapter 6

A Recall-Based Cluster Formation Game

In this chapter, we focus on the second axis of techniques we studied for improving the efficiency of query evaluation in peer-to-peer systems, that is, the self-organization of the peers in the logical overlay network in clusters that are formed based on the similarity of the content and the query workload.

We studied the formation and evolution of such clustered overlay networks in which the clusters are formed based on the recall of the query workload of the peers (Section 6.1). We adopted a game theoretic approach and modelled the problem of cluster formation as a strategic game and discerned between selfish and altruistic behavior of peers (Section 6.2). We studied the game with respect to stability and optimality (Section 6.3) and further explored its properties, by studying specific interesting scenarios (Section 6.4). In addition, we considered besides the problem of cluster formation, the problem of cluster evolution and maintenance and provided a fully distributed uncoordinated protocol based on local decisions made by the peers to cope with updates (Section 6.5). Through an extensive experimental evaluation, we showed that our protocol successfully competes

with a corresponding coordinated one, and manages to maintain the overall recall of the queries in the system under changing conditions (Section 6.6). We conclude the study of the self-organization of the peers into clusters with a summary of our results (Section 6.7).

6.1 Problem Definition

Measurements from the deployment of large-scale systems such as social networks and file sharing peer-to-peer systems have shown that the interactions among their participants (peers) indicate the existence of implicit groups (*clusters*) of peers having similar content or interests. For example, the formation of implicit groups centered around topics described by common keywords has been observed in the blogosphere [12] and in measurements of popular on-line social networks [94], it was also observed that the network structure is such that users form clusters based on common interests, social affiliations or the wish to exploit their shared content.

We consider a distributed system consisting of highly dynamic nodes (peers) that share content. Usually, such distributed systems need to scale up to a large number of peers (Internet-scale). Thus, a peer is unable to know and directly communicate with all other peers in the system. Instead, it establishes logical links with only a few other peers, creating logical overlay networks on top of the physical one. Queries are routed through this overlay to locate peers that hold content of interest.

We use N to denote the current set of peers. We do not assume any specific model for the data items shared by the peers, but adopt a rather generic approach where each data item is described by a set of attributes (e.g. keywords for text documents). We denote the number of results for a query q (e.g. keyword query) against the documents of peer n_i as *result(q, n_i)*.

Let Q be the list of all queries in the system. Note that a query q may appear more than once in Q. Let $num(Q)$ be the number of all queries in Q and $num(q, Q)$ be the number of appearances of query q in Q. We characterize the importance of a peer n_i in the evaluation of a query q in Q based on the results that n_i offers for q with regards to the total number of available results (i.e. the recall achieved when q is evaluated solely on n_i). Specifically:

$$r(q, n_i) = \frac{result(q, n_i)}{\sum_{n_k \in N} result(q, n_k)}.$$

We also define as *local workload* of peer n_i, $Q(n_i)$, the list of queries issued by peer n_i. Again, $num(Q(n_i))$ stands for the number of all queries in $Q(n_i)$ and $num(q, Q(n_i))$ for the number of appearances of query q in $Q(n_i)$.

The efficiency of query evaluation depends heavily on the structure of the overlay network formed by the peers since that is used to forward queries to interesting content. Thus, efficient overlays improving system performance are required. Typical examples of such overlays are structured overlays such as Chord and CAN with strict topologies in which locating any peer (i.e. any data item) takes $O(log|N|)$ and clustered overlays

in which peers are more loosely structured. In clustered overlay networks, peers form sets, called *clusters*. The main motivation for clustering is that inside each cluster, the evaluation of a query is cost efficient. In such overlays, the peers within each cluster are usually highly connected and it is very efficient for each of them to communicate with any other member of the same cluster. For example, Fig. 6.1 shows 8 peers forming a cluster following a fully connected topology (Fig. 6.1(a)), where each peer can reach any other peer in the cluster with 1 hop, while (Fig. 6.1(b)) a structured Chord-like ([96]) topology in which finding any peer takes at most log(8) hops.

Once the appropriate cluster for a query is identified, the peers in the cluster possess relevant content that can be exploited to evaluate and refine the query efficiently. In particular, traces of popular p2p systems have indicated that peers exhibit the property of interest-based locality, that is, if a peer holds content satisfying some query of another peer, then it is most likely that it also maintains additional content of interest to this other peer [120, 58]. Thus, placing the two peers in the same cluster would increase the recall of their queries.

While, there is a large body of research on the discovery and construction of clustered overlays [13, 31, 85, 131, 63, 64, 41, 27, 36], their maintenance, which is imperative for coping with the dynamic nature of peers, has been mostly ignored. Peers, which join or leave the system constantly and change their content and query workload frequently, may render the original clustered overlay inappropriate under the current system conditions. One solution to the problem is to re-apply the clustering procedure that was used to form the original overlay from scratch taking into account the updated state. However, this incurs large communication costs. It also requires global knowledge about the system state that compromises peer autonomy.

We study the dynamics of clustered overlay networks by adopting a game-theoretic perspective. We model the problem of cluster formation as a strategic game with peers as the players. Each peer plays by selecting which clusters to join. This selection or strategy is determined individually by each player, so as to minimize a utility function that depends on the membership cost entailed in belonging to a cluster and the cost of evaluating its query workload at remote clusters. We model both selfish and altruistic behavior of peers as demonstrated in real content-sharing systems by proposing appropriate utility functions. To cope with dynamics, our game is a repeated one: peers re-evaluate their strategy and potentially relocate to other clusters. We define appropriate relocation policies for both selfish and altruistic peers and propose an uncoordinated cluster reformulation protocol based on local decisions made independently by each peer.

Game theoretic approaches have been applied to model the behavior of peers in p2p systems. The originality of our approach lies on the fact that we consider clustered overlays, focus on queries and aim at increasing their recall. In [37], the creation of an Internet-like network is modelled as a game with peers acting as selfish agents without central coordination. The aim of the game is for each peer to choose the peers with which to establish links. The peers pay for the creation of a link, but gain by reducing the

shortest distance to any other peer in the system. In our approach, instead of establishing links randomly, we consider content and query workload for creating clusters of peers with similar properties. [80] considers a more sophisticated model, in which strict bounds are enforced on the out-degree of the peers, links are directed and peers are allowed to express preferences regarding the choice of their neighbors. Our approach can be viewed as setting these preferences based on recall benefits. In [97], the authors show that allowing peers to act completely freely performs much worse than collaboration, and prove that even a static p2p system of selfish peers may never reach convergence. This result is in accordance to our findings that show that in only specific scenarios, we reach a Nash equilibrium. In [132], altruistic peers determine the level of their contribution to the system based on a utility function that depends on a variety of parameters such as the amount of data they upload and download, whereas in our altruistic policy, the choice of the cluster a peer joins depends on its contribution to this cluster.

6.2 Recall Based Clustering

We adopt a game theoretic perspective and model the cluster formation problem as a strategic game by focusing on queries and aiming at improving their recall.

Clustering as a Game: We model the problem of cluster formulation as a strategic game. Each peer n_i represents a player in the game and its strategy s_i is defined by the set of clusters it joins. In particular, each peer n_i chooses which clusters to join from the set of C_{max} clusters in the system, $C = \{c_1, c_2, \ldots, c_{C_{max}}\}$, thus, defining its strategy $s_i \subseteq C$. For example, consider $N = \{n_1, n_2, n_3, n_4\}$ and $C = \{c_1, c_2, c_3\}$, and let us assume that n_1 belongs to clusters c_1 and c_2, n_2 belongs to c_1, n_3 to c_3 and n_4 to c_2 and c_3. Then, the corresponding strategies are $s_1 = \{c_1, c_2\}$, $s_2 = \{c_1\}$, $s_3 = \{c_3\}$ and $s_4 = \{c_2, c_3\}$.

We can describe any cluster configuration by the set of strategies $S = \{s_1, s_2, \ldots, s_{|N|}\}$ that the peers in N deploy, since from this set, we can derive the set of peers belonging to each cluster in C. We constrain C_{max} to be equal to $|N|$, i.e. it cannot exceed the number of peers, and assume that some clusters may be empty if needed. To cope with peer dynamics, each peer plays more than once, thus, the cluster configuration is not static.

The goal of the game is for each player (peer) to minimize or maximize a *utility* function. We discern between two types of peers, *selfish* and *altruistic* ones, and define a corresponding utility function for each type.

6.2.1 Individual Peer Measures

A selfish peer is interested in increasing the recall of its local query workload by joining those clusters whose peers would increase the recall of its local workload the most. Specifically, let $N(s_i)$ be the set of all peers belonging to any cluster $c \in s_i$. The gain for a peer n_i for choosing strategy s_i is the recall of its local workload achieved by evaluating its queries in the peers $N(s_i)$. Stated differently, the cost for n_i associated with s_i is the

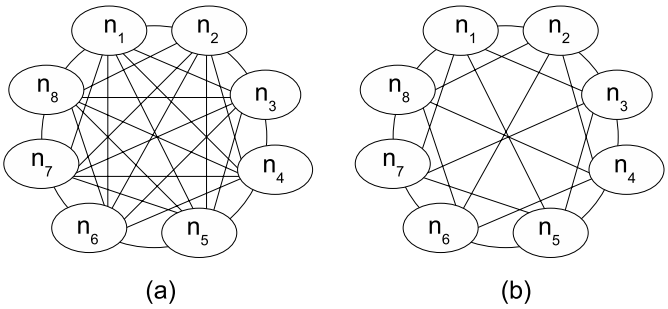


Figure 6.1: Examples of intra cluster topologies

cost (recall) for obtaining query results from peers located in clusters that do not belong to s_i, that is, for peers not in $N(s_i)$.

Clearly, this recall-based cost is minimized, if a peer joins all C_{max} clusters in the system. However, participation in a cluster imposes communication and processing costs. Such costs depend on the size and the topology of the cluster. The larger the size of the cluster, the higher the cost of joining, leaving and maintaining the cluster. Furthermore, a highly connected topology, where each peer maintains links to a large number of other peers, increases the cluster membership cost. To capture this, the cluster membership cost is defined as a monotonically increasing function θ of the number of peers belonging to the cluster, i.e. as a function of the cluster size $|c|$. This function depends on the cluster topology, for instance, when all peers are connected to each other (Fig. 6.1(a)), θ may be linear, whereas in the case of structured overlays (Fig. 6.1(b)), θ may be logarithmic.

Definition 6.1 (Individual Peer Cost)**.** In a cluster configuration S, the individual cost for a selfish peer n_i for choosing strategy s_i is:

$$pcost(n_i, S) = \alpha \sum_{c_k \in s_i} \frac{\theta(|c_k|)}{|N|} + \sum_{q \text{ in } Q(n_i)} \frac{num(q, Q(n_i))}{num(Q(n_i))} \sum_{n_j \notin N(s_i)} r(q, n_j)$$

The first term expresses the cost for cluster membership and the second one the cost (in terms of recall) for obtaining results from peers outside the selected clusters, that is, the average result loss from not participating in all clusters. The recall loss of each query is weighted by its frequency in the local workload of n_i. Parameter α ($\alpha \geq 0$) determines the extent of influence of the cluster membership cost in cluster formation. From a system perspective, parameter α characterizes the ratio between updates and queries in the system. For a given θ, a large value of α means that updates in the system are rather frequent and therefore the cost for cluster maintenance is high, while a small value indicates that query evaluation efficiency is more important for determining the overall system performance. Finally, the factor $1/|N|$ is used for normalizing the cluster membership cost.

Observe that the two terms of the cost function tend to guide the peer towards selecting opposing strategies. For example, assume that a peer can join only one cluster and that

$\alpha \geq 1$. In a cluster configuration in which all peers form a single cluster, the membership cost, that is θ, is maximized ($\theta(|N|)$), while the recall loss is minimized (0) since for any peer all results for its queries are located within its cluster. In contrast, the recall loss is maximized when n_i forms a cluster by its own, while the membership cost is in this case minimized.

Note that in the proposed model, we focus on recall and abstract the cost of processing queries both within each cluster and across clusters. There is a large body of research on routing queries among interconnected peers. By concentrating on recall, we aim at capturing the basic mechanisms underlying clustering independently of any query routing specifics.

While minimizing the individual cost function is appropriate for modelling the behavior of selfish peers that try to maximize the recall of their own queries, we also want to model the behavior of altruistic peers. Altruistic peers are not concerned about their own queries, instead they are interested in offering to other peers. Therefore, we define the corresponding utility function, called *individual peer contribution* (*pcontr*) that an altruistic peer n_i aims at maximizing based on how much n_i improves the recall of the other peers that belong to the clusters of its strategy. Thus, analogously to Def. 6.1, the individual contribution is defined as follows:

Definition 6.2 (Individual Peer Contribution)**.** In a cluster configuration S, the individual contribution of an altruistic peer n_i by choosing strategy s_i is:

$$pcontr(n_i, S) = \frac{1}{|N|} \sum_{n_j \in N(s_i)} \sum_{q \in Q(n_j)} \frac{num(q, Q(n_j))}{num(Q(n_j))} r(q, n_i) - \frac{\alpha}{|N|^2} \sum_{c_k \in s_i} |c_k| \theta(|c_k|)$$

While *pcost* measures the cost n_i pays for its query workload and membership to clusters in s_i, *pcontr* is a positive measure showing what other peers gain when n_i chooses strategy s_i. The first term of the sum measures the contribution of peer n_i to the peers in the clusters of its strategy, while the second term measures the membership cost these peers pay if n_i joins their clusters. Similarly to the individual cost, the membership cost also takes its lowest value when the peer forms a cluster by its own and its largest when all peers form a single cluster (if we consider that each peer joins only a single cluster), whereas the recall it contributes to other peers takes its largest value in the single cluster and its lowest when it forms a cluster by its own. Individual contribution is defined from the perspective of each beneficiary peer, that is, the queries weights are defined based on their relative frequencies per such peer.

Besides the pure selfish and the pure altruistic behavior, hybrid behavior can be captured by trying to minimize the following cost function:

$$hpcost(n_i, S) = d\ pcost(n_i, S) - (1 - d)\, pcontr(n_i, S),$$

where $d \in [0,\ 1]$ captures the degree of selfishness of peer n_i. A hybrid peer considers both its own cost (with degree d) and its contribution to the others (with degree $1 - d$).

Finally, note that our game is a non-cooperative asymmetric game. A game is *asymmetric*, if the value of the utility function or *payoff* differ if different players select the same strategy.

6.2.2 Global Cost Measures

We measure the overall quality of a cluster configuration by the achieved *social cost* ($SCost$) defined as:

Definition 6.3 (Social Cost)**.** The social cost of a cluster configuration S is defined as the sum of the individual costs of all peers in N:

$$SCost(S) = \sum_{n_i \in N} pcost(n_i, S)$$

We can also evaluate the overall quality of the configuration from a query workload perspective, by considering the average cost for attaining results for all queries in Q.

Definition 6.4 (Workload Cost)**.** The workload cost of a cluster configuration S is:

$$WCost(S) = \alpha \sum_{c_k \in C} \frac{|c_k|\theta(|c_k|)}{|N|} + \sum_{q \text{ in } Q} \frac{num(q,Q)}{num(Q)} \sum_{n_i \, s.t. \, q \text{ in } Q(n_i)} \frac{num(q, Q(n_i))}{num(q,Q)} \sum_{n_j \notin N(s_i)} r(q, n_j)$$

The first term expresses the cost for maintaining the clusters. The second term expresses the cost for all queries, i.e., the cost for evaluating them outside the clusters of their initiator.

The main difference between the social and the workload cost lies on how they assign weights to the queries. In the social cost, each peer assigns weights to its queries based on their frequency in its local workload, whereas in the workload cost, the weight assigned to each query is based on the frequency of the query in the overall query workload. Intuitively, while the social cost regards all peers as equals, the workload cost considers more demanding peers, i.e. peers that pose more queries, as more important than low demanding ones.

The two cost measures are not equal in the general case, but for equally demanding peers the following proposition holds.

Proposition 6.1. If for all peers n_i, $n_j \in N$, $num(Q(n_i)) = num(Q(n_j)) = \frac{num(Q)}{|N|}$, the social and the workload cost measures are proportional to each other.
Proof. Using the definition of individual cost (Def. 6.1), the social cost can be written as:

$$SCost(S) = \alpha \sum_{n_i \in N} \sum_{c_k \in s_i} \frac{\theta(|c_k|)}{|N|} + \sum_{n_i \in N} \sum_{q \text{ in } Q(n_i)} \frac{num(q, Q(n_i))}{num(Q(n_i))} \sum_{n_j \notin N(s_i)} r(q, n_j)$$

The membership cost of $SCost$ is equal to the first term of $WCost$. Just consider that each cluster c_k appears in the sum of $SCost$ as many times as the peers that belong to it, i.e., its size $|c_k|$. The second term differs from the second term of $SCost$ only on how much the workload of each peer is taken into account. It is easy to see, that if peers get an equal part of the query workload, i.e., $num(Q(n_i)) = num(Q(n_j))$, for all peers n_i, $n_j \in N$, the recall parts of the two costs are proportional.□

Proposition 6.1 implies that improving the social cost improves the workload cost and vice versa.

In accordance to the social and workload cost, we define the corresponding social and workload contribution as:

Definition 6.5 (Social Contribution)**.** The social contribution of a cluster configuration S is defined as the sum of the individual contributions of all peers in N:

$$SContr(S) = \sum_{n_i \in N} pcontr(n_i, S)$$

Definition 6.6 (Workload Contribution)**.** The workload contribution for a cluster configuration S is:

$$WContr(S) = \sum_{q \text{ in } Q} \frac{num(q,Q)}{num(Q)} \sum_{n_i \, s.t. \, q \text{ in } Q(n_i)} \frac{num(q,Q(n_i))}{num(q,Q)} \sum_{n_j \in N(s_i)} r(q,n_j) - \frac{\alpha}{|N|^2} \sum_{n_i \in N} \sum_{c_k \in s_i} |c_k| \theta(|c_k|)$$

Similarly to $SCost$ and $WCost$, the $SContr$ and $WContr$ are also proportional for specific workload distributions, in particular, when the query workload is uniformly distributed among the peers.

Proposition 6.2. If for all n_i, $n_j \in N$ and all q in Q, $num(q, Q(n_i))/num(Q(n_i)) = num(q, Q(n_j))/num(Q(n_j)) = num(q, Q)/num(Q)$, the social and the workload contribution measures are proportional to each other.

Intuitively, social contribution favors queries that are popular to specific peers, whereas its workload counterpart favors overall popular queries.

Let us now examine the relationship between the workload cost and the workload contribution.

Proposition 6.3. For $\alpha = 0$, that is, if ignore the cluster membership cost, it holds: $WCost(S) = 1 - WContr(S)$, which means that the two measures are complementary.
Proof. It holds that:

$$\sum_{n_j \notin N(s_i)} r(q,n_j) + \sum_{n_j \in N(s_i)} r(q,n_j) = 1, \forall q \text{ in } Q, s_i \in S. \tag{6.1}$$

For $\alpha = 0$, we have: $WCost(S) = 1 - WContr(S)$.□

Let us consider now, the social cost and the social contribution.

Corollary 6.1. *For uniform query workload among peers the social cost and social contribution are complementary:* $SContr(S) = 1 - SCost(S)$.
Proof. Again, for $\alpha = 0$, we can rewrite $SCost(S)$ using (6.1) as:

$$SCost(S) = \sum_{n_i \in N} \sum_{q in Q(n_i)} \frac{num(q, Q(n_i))}{num(Q(n_i))} (1 - \sum_{n_j \in N(s_i)} r(q, n_j)) =$$
$$|N| - \sum_{n_i \in N} \sum_{q in Q(n_i)} \frac{num(q, Q(n_i))}{num(Q(n_i))} \sum_{n_j \in N(s_i)} r(q, n_j)$$

and if we assume that $\frac{num(q,Q(n_i))}{num(Q(n_i))}$ is the same for all peers n_i, then we have that: $SContr(S) = 1 - SCost(S)$.□

6.3 Stability and Optimality

The goal of each player (peer) is to minimize/maximize its individual cost/contribution. We will refer in the following to selfish peers, but the same results are applicable for altruistic behavior.

6.3.1 Stability

The question that arises is: if we leave the players free to play the game to achieve their goal, will the system ever reach a stable state in which no players desire to change their strategy (the set of clusters they belong to)? That is, will the system reach a Nash equilibrium?
Nash Equilibrium: Formally, a (pure) Nash equilibrium is a set of strategies S such that, for each peer n_i with strategy $s_i \in S$, and for all alternative set of strategies S' which differ only in the i-th component (different cluster sets s'_i for n_i):

$$pcost(n_i, S) \leq pcost(n_i, S') \tag{6.2}$$

This means that in a Nash equilibrium, no peer has an incentive to change the set of clusters it currently belongs to, that is, Nash equilibria are stable.

We shall first prove an interesting property of the cluster formation game. Due to the form of our cost function, the stable states in our system have the following property that constraints the number of possible configurations:

Lemma 6.1. *In any stable state, there are no clusters* c_i, c_j *such that* $c_i \subseteq c_j$, $i \neq j$.
Proof. Let S be a cluster configuration, c_i, c_j be two clusters in C such that $c_i \subseteq c_j$. Consider a peer n_k, $n_k \in c_i$. Clearly, $n_k \in c_j$. Let the individual cost of n_k be: $pcost(n_k, S) = \alpha\gamma + \delta$, where γ is the membership cost for n_k when following strategy

$s_k \in S$ and δ the respective recall it loses from the peers that do not belong to $N(s_k)$. Assume for the purposes of contradiction that S describes a stable configuration, then n_k can not select a strategy that would reduce its cost. Let us examine the strategy $s'_k = s_k - \{c_i\}$. Let S' be the configuration resulting by replacing s_k with s'_k in S. Then, $pcost(n_k, S') = \alpha(\gamma - \frac{\theta(|c_i|)}{|N|} + \delta) < pcost(n_k, S)$. The recall part of the cost function remains the same, because $N(s_k) = N(s'_k)$. Thus, n_k can reduce its cost by selecting the strategy s'_k, and therefore S is not a stable state, which contradicts our assumption.□

Because of Lemma 6.1, it holds:

Corollary 6.2. *When a peer forms a cluster by itself, it cannot belong to any other cluster.*

It is rather simple to show that for the cluster formation game, a pure Nash equilibrium does not always exist.

Proposition 6.4. A pure Nash equilibrium does not always exist for the cluster formation game.

Proof. Let us consider a simple scenario of two peers n_1 and n_2. Consider also that $Q(n_1)$ consists of a single query q_1 satisfied by n_2 (i.e. $r(q_1, n_2) = 1$) and $Q(n_2)$ consists of q_2 also satisfied by n_2. Let $C = \{c_1, c_2\}$ be the clusters in the system. Using Lemma 6.1, the following cluster configurations are possible: $n_1 \in c_1$ and $n_2 \in c_2$, described by $S_1 = \{\{c_1\}, \{c_2\}\}$, $n_1 \in c_2$ and $n_2 \in c_1$, described by $S_2 = \{\{c_2\}, \{c_1\}\}$ and both $n_1, n_2 \in c_1$ or c_2 described by $S_3 = \{\{c_1\}, \{c_1\}\}$ and $S_4 = \{\{c_2\}, \{c_2\}\}$, respectively. Let us assume a linear θ function, $\theta(n) = n$. Then, for any value of $\alpha > 0$, we can show that none of the possible configurations is a Nash equilibrium. In particular, since the first two configurations are symmetric, let us examine the first one. The individual costs of the two peers are: $pcost(n_1, S_1) = \alpha\frac{1}{2} + 1$ and $pcost(n_2, S_1) = \alpha\frac{1}{2}$. If n_1 moves to cluster c_2, then the system configuration becomes $\{\{c_2\}, \{c_2\}\}$, that is, configuration S_4, and the cost for n_1 becomes $pcost(n_1, S_4) = \alpha \leq pcost(n_1, S_1)$. Thus, configuration S_1 is not a Nash equilibrium, since n_1 can reduce its cost by moving to c_2. Let us consider now the configuration S_3 (S_4 is symmetric) in which both peers belong to the same cluster. Their individual costs are now: $pcost(n_1, S_3) = \alpha$ and $pcost(n_2, S_3) = \alpha$. Peer n_2 can reduce its cost by moving to the (empty) cluster c_2 (resulting in configuration S_1) and therefore S_3 is not a Nash equilibrium. Table 6.1 summarizes the payoff (cost) table for this two-player game.□

6.3.2 Social Optimum

Even if the system does eventually reach a stable state (Nash equilibrium), it is not always the case that this stable state has a satisfying cost. A measure widely used for evaluating how far from the best possible outcome a stable state is, is the *price of anarchy* defined as the ratio between the social cost of the worst Nash equilibrium and the "social optimum".

Table 6.1: Payoff table

	n_2 joins c_1	n_2 joins c_2
n_1 joins c_1	α,α	$\frac{\alpha}{2}+1$,$\frac{\alpha}{2}$
n_1 joins c_2	$\frac{\alpha}{2}+1$,$\frac{\alpha}{2}$	α,α

The social optimum is obtained by minimizing the social cost measure over all possible configurations, even for those configurations that do not correspond to a stable state.

We can acquire a rough bound of the social optimum by considering each peer separately and evaluating its individual cost over all possible configurations. Then, by selecting for each peer the configuration that yields the minimum individual cost and adding these values, we obtain a bound for the minimum value of the social cost in the system, i.e., for the social optimum. Note that we are adding together individual costs that may correspond to different configurations, thus, the estimated social cost may refer to a configuration that cannot exist and may be very far from the actual value of the social optimum that we can achieve.

6.4 Case Studies

Although, in the general case, a Nash equilibrium does not always exist, there are cases in which, for specific configurations and data and query workload distributions, stable clusters may be formed. Next, we present two scenarios:

Case I: No Underlying Clustering: In this case, all peers in N are considered similar in the following sense:

$$num(Q(n_i)) = num(Q(n_j)) = num(Q)/|N|, \forall n_i, n_j \in N$$
$$r(q, n_i) = r(q, n_j) = 1/|N|, \forall q \text{ in } Q, \forall n_i, n_j \in N$$

This corresponds to a data and query distribution for which no physical grouping among the peers exist. Note that our game becomes a symmetric one, since all players yield the same payoffs when applying the same strategy.

Case II: Symmetric Clusters: In this case, the data and query distribution are such that a perfect underlying clustering/grouping exists among the peers. In particular, the peers in N belong to m ($m > 1$) different groups of the same size $|c| = |N|/m$. The members in each group offer and demand data only within their group. Formally, for all pairs of peers n_i, n_j in the same group, it holds $num(Q(n_i)) = num(Q(n_j))$ and $\forall q$ in $Q(n_j)$, $r(q, n_i) = 1/|c|$, whereas for all pairs of peers n_i, n_j not in the same group, the lists $Q(n_i)$ and $Q(n_j)$ have no queries in common and $\forall q$ in $Q(n_j)$, $r(q, n_i) = 0$.

For each of these two scenarios, we consider a number of cluster configurations and study each of them in terms of stability and optimality.

6.4.1 Stability

To determine whether a cluster configuration constitutes a Nash equilibrium, we need to ensure that the individual cost of any peer is not smaller in any possible configuration that can result from the current one by changing only the strategy of this peer, by evaluating Inequality (6.2).

For the first scenario (Case I), we study the following cluster configurations:
CASE(I.A): A SINGLE CLUSTER. In this case, all peers form a single cluster. Let us call this configuration S_1. Then for each peer n_i we have:

$$pcost(n_i, S_1) = \alpha \frac{\theta(|N|)}{|N|}. \tag{6.3}$$

From Corollary 6.2, the only way a peer n_i can change its strategy is by forming a cluster by its own (configuration S_2). Its new *pcost* is:

$$pcost(n_i, S_2) = \frac{\alpha\theta(1) + |N| - 1}{|N|}. \tag{6.4}$$

For configuration S_1 to correspond to a Nash equilibrium, it must hold $pcost(n_i, S_1) \leq pcost(n_i, S_2)$, $\forall n_i \in N$ and by evaluating these costs, we get that this is true for:

$$\alpha \leq \frac{|N| - 1}{\theta(|N|) - \theta(1)}. \tag{6.5}$$

CASE(I.B): EACH PEER FORMS A CLUSTER BY ITS OWN. In this case, each peer forms a cluster by its own. The only way for a peer n_i to change its strategy is to leave its own cluster and join k other clusters, where $1 \leq k \leq |N| - 1$. After following an evaluation similar to Case (I.a), we conclude that this configuration is a Nash equilibrium for:

$$\alpha \geq \frac{k}{k\theta(2) - \theta(1)}. \tag{6.6}$$

CASE(I.C): m NON-OVERLAPPING CLUSTERS. The peers form m *non-overlapping clusters* of the same size $|c|$. Consider a peer $n_i \in c_j$. The available options for n_i for changing its strategy are to: (1) form a cluster by its own; (2) additionally to c_j, join k other clusters, where $1 \leq k < m$; or (3) leave c_j and join k other clusters.

The conditions that need to hold for this case to be a Nash equilibrium are with respect to each of the options n_i has to change its strategy:
for option (1):

$$\alpha \leq \frac{|c_j| - 1}{\theta(|c_j|) - \theta(1)}, \tag{6.7}$$

for option (2):

$$\alpha \geq \frac{|c|}{\theta(|c| + 1)}, \tag{6.8}$$

Table 6.2: Conditions for stability

	Case (I.a)	Case (I.b)	Case (I.c)
Cost-based	$\alpha \leq \frac{\lvert N\rvert-1}{\theta(\lvert N\rvert)-\theta(1)}$	$\alpha \geq \frac{k}{k\theta(2)-\theta(1)}$	$\alpha \leq \frac{\lvert c\rvert-1}{\theta(\lvert c\rvert)-\theta(1)}$, $\alpha \geq \frac{\lvert c\rvert}{\theta(\lvert c\rvert+1)}$, $\alpha \geq \frac{k(\lvert c\rvert-1)+1}{k\theta(\lvert c\rvert+1)-\theta(\lvert c\rvert)}$
Linear θ	$\alpha \leq \frac{1}{\phi}$	$\alpha \geq \frac{k}{(2k-1)\phi}$	$\alpha \leq \frac{1}{\phi}$, $\alpha \geq \frac{\lvert c\rvert}{\phi(\lvert c\rvert+1)}$, $\alpha \geq \frac{k(\lvert c\rvert-1)+1}{k\phi(\lvert c\rvert+1)-\phi\lvert c\rvert}$
	Case (II.a)	Case (II.b)	Case (II.c)
Cost-based	$\alpha \leq \frac{\lvert N\rvert(\lvert N\rvert-m)}{m\theta(\lvert N\rvert)-\lvert N\rvert\theta(1)}$	$\alpha \geq \frac{m}{\theta(2)-\theta(1)}$	$\frac{\lvert N\rvert^2-m}{\lvert N\rvert(\theta(\lvert N\rvert/m+1)-\theta(\lvert N\rvert/m))} \leq \alpha \leq \frac{(\lvert N\rvert-m)}{(\theta(\lvert N\rvert/m)-\theta(1))}$
Linear θ	$\alpha \leq \frac{(\lvert N\rvert-m)}{\phi(m-1)}$	$\alpha \geq \frac{m}{\phi}$	$\frac{\lvert N\rvert^2-m}{\phi\lvert N\rvert} \leq \alpha$, $\frac{(\lvert N\rvert-m)}{\phi(\lvert N\rvert/m-1)} \leq \alpha$

and for option (3):

$$\alpha \geq \frac{k(|c|-1)+1}{k\theta(|c|+1)-\theta(|c|)} \tag{6.9}$$

Table 6.2(line 1) presents the results of our evaluation for selfish peers that aim at minimizing their individual cost. By considering altruistic peers that aim at maximizing their individual contribution, Inequality (6.2) becomes:

$$pcontr(n_i, S) \geq pcontr(n_i, S') \tag{6.10}$$

We summarize in Table 6.2(line 3) the results of the respective analysis for altruistic peers.

This shows that, even if there is no underlying clustering according to the data and query workload distribution, a system can still reach a stable state. This state depends on the cluster maintenance costs and the portions of data and query workload each peer offers or demands.

To make this more concrete, let us assume a θ function corresponding to a linear function of the form: $\theta(n) = \phi n$, $0 < \phi \leq 1$, and rewrite the conditions regarding α (Table 6.2(line 2)). Then, a configuration in which all peers belong to a single cluster is stable for $\alpha \leq 1/\phi$. Recall that large values of α mean that maintenance costs are more important than query recall. Thus, for the same θ, for values of α larger than this threshold, the maintenance cost would surpass those gained by recall and would lead to splitting the cluster. Note also, that whether a single cluster is stable or not depends also on the topology as captured through function θ. For instance, when ϕ is small (less connected topology), a single cluster remains stable for larger values of α.

Table 6.2(line 4) also includes the corresponding conditions for altruistic peers when θ is linear.

For the symmetric clusters scenario, we limit our analysis to the case in which each peer can belong only to one cluster and study the same configurations as in Case I.

Case(II.a): A single cluster. Same as Case (I.a).

Case(II.b): Each peer forms a cluster by its own. The only option for n_i is to join another peer n_j, which either is in the same group with n_i, or belongs to a different group. This configuration is the same, whatever group out of the $m-1$ we consider, since all such peers are symmetric to n_i, i.e., they do not satisfy any of its local query workload.

CASE(II.C): m NON-OVERLAPPING CLUSTERS. In this case, we consider that each of the m clusters contains peers of a single group. Then, the individual peer cost for each peer $n_i \in N$ is equal to its cluster membership cost, since the cost for computing queries outside its cluster is zero (there are no results for $Q(n_i)$ in peers not in $N(s_i)$). If n_i wants to change its strategy s_i, then it can move either to a cluster on its own or to a different existing cluster.

The results of the same analysis as in the first case are presented in Table 6.2(line 5) for selfish peers and for selfish peers when θ is linear in Table 6.2(line 6). The results for altruistic ones can be easily computed and are omitted.

By comparing Case I (no underlying clustering) with $k = 1$ and Case II (perfect underlying clustering), we see that in Case II, configuration (B) in which each peer forms its own cluster (no clustering) is stable for larger values of α, whereas the other two configurations (A) and (C) (with some form of clustering) are stable for smaller values of α.

6.4.2 Optimality

We examine whether any of the Nash equilibria that we have previously computed achieve a social cost equal to the social optimum. To this end, we need to compare their social cost against that of any other possible configuration. We assume selfish peers and a linear θ function. Our results can be easily adapted for altruistic peers.

In Case I, where all peers are symmetric, minimizing the individual cost of any peer suffices to minimize the social cost. CASE(I.A): A SINGLE CLUSTER. We already know that a configuration in which each peer forms its own cluster has a larger cost than Case (I.A), since we assume that $\alpha \leq 1/\phi$ and Case (I.A) is an equilibrium. The only other possible configuration is when a peer n_i joins k clusters with different sizes. The best case (the case with the lowest cost) is the one where the k clusters have no overlapping members. It also holds that $|N(s_i)| < |N|$, otherwise we would have a single cluster. By comparing the social cost of this configuration to the cost of Case (I.A), we see that for $\alpha \leq 1/\phi$, Case (I.A) has the lowest cost. Thus, the value of the cost of Case (I.A) corresponds to the social optimum.

CASE(I.B): EACH PEER FORMS A CLUSTER BY ITS OWN

In this case, we have a equilibrium when $a \geq k/(2k-1)\phi$. k can take values from 1 to $|N|$ and α takes its largest value for $k = 1$. Thus, the cost corresponds to the social optimum as in the previous scenario for $a \geq 1/\phi$.

We observe that for $\alpha = 1/\phi$ both case (I.a) and (I.b) correspond to equilibria with the same social cost, equal to the social optimum.

CASE(I.C): m NON-OVERLAPPING CLUSTERS

If we consider $m > 1$ then the third case does not correspond to a social optimum since we already showed that for any value of α as we already showed above. If $m = 1$ then this case is the same as forming a single cluster or each peer its own and we fall back to the previous analysis.

In Case II, the peers belonging to each group are symmetric to each other and each group is symmetric to the others. Thus, to determine the social optimum, it again suffices to find the configuration that minimizes the individual cost for any of the peers.

Case(II.a): A single cluster

For the first configuration we can easily discern that for any value of $\alpha > 0$ a configuration with a lower social cost is one where the m groups form separate clusters. When all the peers form a single cluster each peer connects to $|N| - |c|$ peers which do not offer any recall to its queries thus the cost can easily be reduced by not connecting to them.

Case(II.b): Each peer forms its own cluster

In this case we need to consider the case where a peer joins a cluster with k other peers where $1 < k < |c|$. We only consider peers of the same group (i.e. at the most —c—-1) since peers from other groups only increase the membership cost without reducing the recall factor. If we consider $a \geq m\phi$, then we conclude that the cost for a peer of joining a cluster with more peers is higher than forming a cluster by its own. So again, this configuration corresponds to a social optimum.

Case(II.c): x non-overlapping clusters

For the linear θ function the configuration corresponds to the social optimum. We have already considered the case of a peer moving to its own cluster or a different existing cluster. If a peer joins its cluster and any other cluster it only increases its membership cost without reducing its recall loss as all other peers hold no data of interest. Thus, the only case we need to consider is when n_i forms a cluster with $|c'| < |c|$ peers of its own category (configuration S"'). If the cost in this configuration is again larger, then S corresponds to a social optimum. This holds for $\alpha \leq 1/\phi(c - c')$.

6.5 Cluster Evolution

Assume some initial cluster configuration. As the system evolves, the recall achieved by this cluster configuration may deteriorate. Changes that affect the quality of clustering include topology updates as peers enter and leave the system, as well as changes of peer content and query workload. We propose a suite of protocols to keep the clustered overlay up-to-date with respect to these changes. Our protocols are based on local relocation policies that each peer follows so as to move to the most appropriate cluster under the given system conditions. Such protocols can also be used to bootstrap the system, for example, by applying them on an initial configuration in which all peers belong to a single cluster or each peer forms a cluster by its own. We describe first the relocation policies followed by each peer, and then how they are applied to form a new cluster configuration.

6.5.1 Relocation Policies

Unlike most network creation games, our game is not a one-shot game but a repeated one, where the peers re-examine their strategy selection through time to cope with the

system dynamics.

Let S_{cur} be the current cluster configuration. When it is its turn to play, each peer n_i considers all possible configurations S_j that differ from S_{cur} only at their i-th component, i.e., the strategy s_i that peer n_i follows. For simplicity, we focus on the case where each peer belongs to a single cluster. Let C_{cur} be the current set of (nonempty) clusters in the system. Then, for each peer n_i, it holds: $s_i = \{c_l\}$, for some $c_l \in C_{cur}$. In this case, the possible strategies for n_i besides its current one are: either moving to a cluster c_v, $c_v \neq c_l$ for $c_v \in C_{cur}$ or if $c_l \neq \{n_i\}$, forming a cluster by its own.

Based on the behavior of each peer, we consider two types of relocation policies: *selfish* and *altruistic*. A peer with a selfish policy chooses the strategy s_{new} for which the corresponding cluster configuration S_{new} is:

$$S_{new} = \arg\min_{S_j} pcost(n_i, S_j)$$

Analogously, a peer with an altruistic policy chooses the strategy s_{new} for which the corresponding S_{new} is:

$$S_{new} = \arg\max_{S_j} pcontr(n_i, S_j)$$

A hybrid relocation policy that uses the hybrid peer cost, *hpcost*, is also feasible.

To measure how much a peer benefits from moving to a new cluster, we define a new measure, the *gain*. The gain is defined for a selfish peer as:

$$gain_{n_i} = pcost(n_i, S_{cur}) - pcost(n_i, S_{new})$$

If $gain_{n_i} > 0$, then n_i benefits from selecting the new strategy. Analogously, gain is defined for altruistic peers.

To implement the policies, we assume that each cluster has a unique identifier, *cid*, known by all its peers, which is assigned based on peer IPs and timestamps. For example, when the first peer joins a cluster, its *cid* is formed by the IP of the peer concatenated with a timestamp. When other peers join the cluster, they are informed of its *cid*. Query results are annotated with the corresponding *cid*s of the clusters that provide them. Thus, peers do not need to know all system *cid*s, but they gradually learn them, as their queries acquire results annotated with new *cid*s. Therefore, when all peers leave a cluster, its *cid* just becomes unused. Recycling *cids* is beyond the scope of the work in this thesis.

In the selfish relocation policy, since, all query results received by a peer are annotated with the *cid* of the cluster they came from, each peer can monitor its recall with respect to all clusters in the system and use it to evaluate its individual cost for the different configurations it needs to consider when it plays. In the altruistic relocation policy, instead of its recall, each peer records the number of results it sends to each cluster so as to evaluate its individual contribution for all different s_i components.

Since no global view of the system is available, a peer can not be aware of all available results for a query and we instead use as recall, in the cost evaluation, the fraction of results returned to peer n_i for query q by a cluster c_j to the total number of results returned for the query.

6.5.2 Cluster Reformulation Protocol

The relocation policies along with the gain are used to form the reformulation protocols.

Coordinated Protocol

We first consider a *coordinated reformulation protocol*. Cluster representatives are used to achieve this coordination by gathering and exchanging information about their clusters. Each peer applies its relocation policy and determines the cluster it needs to move to. Then, all relocation requests are gathered by the representatives and ordered according to non-increasing value of gain. Each representative grants a percentage x of them (Alg. 17).

The cluster representative does not need to remain the same. Representative selection is local within each cluster and may be random or based on specific properties of the peers. When a peer stops acting as a representative, it suffices to redirect all requests to the new representative. Furthermore, for a peer to join a cluster it just needs to know one of its members. It then sends a relocation request to that member, which forwards the request to the current cluster representative.

Uncoordinated Protocol

We argue that the use of coordination is not necessary and instead propose an *uncoordinated reformulation protocol* in which each peer determines when to play locally and independently from the other peers. When a peer determines that it is its turn to play, it applies its relocation policy locally and moves to the cluster the policy indicates. Cluster representatives may be used to facilitate moves between clusters and reduce the overhead.

Protocol Variations

Based on *when* the peers determine that there is their turn to play, we define two variations of the reformulation protocol: an *event* and a *trigger-based* one.

The coordinated event-based protocol is initiated after each system event, i.e., a query or an update. In the uncoordinated event-based protocol, a peer determines that it is its turn to play after it becomes aware of a relevant event. Selfish peers consider as relevant the evaluation of queries of their local query workload, while altruistic peers the provision of results to another peer's query. A hybrid peer may choose either one or both types of events as relevant. A variation of the event-based protocol, the *batch-based* protocol, is initiated after a number (batch) of events instead of just after a single one.

In the coordinated trigger-based protocol, the social or workload cost (or corresponding contributions) are continuously updated and the protocol is initiated when the respective global gain becomes greater than zero. For the uncoordinated protocol, each peer continuously updates its individual gain and plays whenever this value becomes greater than zero. For updating the measures, trigger-based protocols require to monitor the system (workload and content) continuously and thus, introduce additional overheads.

6.5.3 Controlling Parameters

The gains that individual peers attain from relocation may not always worth the reorganization cost. To this end, we present three mechanisms for overhead control, which can be applied to all variations of both the uncoordinated and coordinated protocols, either individually or in combinations.

Stopping Condition. After applying its relocation policy, each peer compares its gain against a system-defined threshold ϵ. The peer determines that it needs to move only if its gain is larger than ϵ. Consequently, reformulation stops without the system reaching an equilibrium, but rather an ϵ-stable state. In the coordinated protocol, the stopping condition may also be applied on a global level, if we measure the gain with respect to the social cost or contribution.

Playing Probability. Instead of allowing a peer to play (i.e. re-evaluate its strategy) every time it is its turn, we introduce the use of a *playing probability* Pr, which determines how aggressive a player is, i.e., how high is the player's chance to play. The playing probability can either be the same for all peers, so as to treat all peers as equals or it may differ for each peer. For example, giving higher probability to peers that change their content or workload often allows them to adapt faster to these changes. Alternatively, a higher probability may be given to peers with more content or heavier query workload, since they are the ones that influence the workload cost and contribution the most.

Quota. Overhead can finally be controlled by enforcing a movement quota. Each peer is assigned a quota of ν possible moves, which is the maximum number of moves it is allowed for a specified period TP_q. After the end of TP_q, the quota is replenished and the peer has again ν available moves for the next TP_q. TP_q can either be a time interval or a number of events. Note, that using a time interval corresponds to treating all peers as equals, while using events to measure TP_q may allow more demanding peers to play more often, as they are affected by more events.

The value of ν expresses a trade-off between consuming system resources for reclustering and tolerating low recall values from a poor clustering.

6.6 Experimental Evaluation

We model a system of peers sharing data belonging to different semantic categories. Each peer in the system is associated with a data category j and maintains documents belonging to it. The local query workload of each peer is generated by first selecting a document category with probability $P(j)$ following a zipf distribution, and then a document d from that category with probability $P(d, j)$ following another zipf distribution within each category. We define $P_{x \in l}(d, j)$ as the probability of peer x associated with category l posing a query about document d of category j as:

$$P_{x \in l}(d, j') = \begin{cases} (1-\mu)P(d,j), l \neq j \\ ((1-\mu) + \mu/P(j))P(d,j), l = j \end{cases}$$

Coordinated Event-Based Protocol
Input: $|N|$: number of peers, $C = \{c_1, \ldots, c_n\}$: cluster set,
$R = \{r_1, \ldots, r_n\}$: cluster representatives

1: **for all** global events **do**
2: **for all** $r_i \in R$ **do**
3: send a game initialise request to all $n_j \in c_i$
4: **for all** $n_j \in c_i$ **do**
5: evaluate $gain_{n_j}$
6: send to r_i a relocation request with $gain_{n_j}$ for c_{new}
7: send all relocation requests to all other $r_i \in R$
8: sort relocation requests in non-increasing order of gain
9: **for all** $gain_{n_j}$ within x% of the list **do**
10: n_j moves from c_i to c_{new}

Algorithm 17: Coordinated Event-Based Protocol

Parameter μ is a measure of the interest-based locality ([120]) the users exhibit. The presence of interest-based locality, i.e., of the peers property to maintain data similar to their local query workload is derived from measurements in real traces of p2p system found in ([120]). For our evaluation we consider three specific scenarios. For the first, $\mu = 1$. This is the *symmetric scenario* in which both queries and data of each peer belong to the same category. In the second scenario, we consider again $\mu = 1$, but for a $j \neq l$ which is selected randomly from the remaining categories. This is the *asymmetric scenario*, in which each peer has data from one category but poses queries for a single different category. Thus, the symmetric scenario exhibits maximum interest-locality, while the asymmetric none. Finally, in the third scenario, the *random scenario*, $\mu = 0$. Again, there is no interest-based locality and each peer has both data and queries uniformly distributed from all categories.

We used as data Newsgroup articles belonging to 10 different categories. The articles were pre-processed, stop words were removed from the text, lemmatization was applied and the resulting words were sorted by frequency of appearance. The texts are distributed among 10000 peers. Queries are generated by choosing a random word from the texts such that each query is satisfied by documents only in a single category. Table 6.3 summarizes our parameters.

We present four sets of experiments. In the first set, we evaluate the event-based and the trigger-based games and the influence of the tuning parameters on them. Furthermore, we compare the uncoordinated versions we propose to corresponding coordinated protocols. In the second set, we examine how our protocols perform when we start from various initial configurations. In the third set, we focus on how well our protocols enable the system adapt to changes. Finally, in the forth set we compare our protocols adjusting capability to a caching scheme.

Table 6.3: Recall-based reformulation tuning parameters

Parameter	Range	Default Value
Topology and Strategy		
number of peers ($\|N\|$)	-	10000
parameter α	1-100	1
membership cost function (θ)	-	log,linear
strategy	-	self-alt-hybrid-mixed
Data-Query Distribution		
number of categories	-	10
interest locality degree (μ)	-	0-1
Tuning Parameters		
stopping condition (ϵ)	0-10^{-8}	$10^{-4}, 10^{-3}, 10^{-6}$
playing probability (Pr)	0-1	0.5
movement quota (ν)	1-15	∞
quota period (TP_q)	-	20, (5*batch size)
% of allowed moves (x)	0.1-1	1

6.6.1 Comparison with Coordinated Protocols

We compare our two types of game, the event-based and trigger-based, as well as the batch-based variation of the event-based game with corresponding coordinated protocols. Our goal is to demonstrate that our protocols work efficiently and competitively to the coordinated ones, despite the lack of global control.

Controlling Parameters Influence

We compare uncoordinated and coordinated versions of the event-based, trigger-based and batch-based protocols with batches of 20, 50 and 100 events, which correspond to the 1/10th, 1/4th and 1/2 of the queries in the average local workload of a peer respectively, for different tuning parameters. We consider asymmetric peers since this type of peers pose the greatest challenge when applying clustering. The peers are all selfish in this experiment. We explore the influence of strategy selection later. We assume that clusters are organized in a chord-like topology (logarithmic θ). We start with an initial configuration in which each peer forms its own cluster, which is similar to a system in which no clustering algorithm has been applied yet. We measure the social cost, the number of movements and the average number of turns per peer in the system until we reach a stable state. With the term turn, we refer to the turn of each player in the game when it needs to decide whether to play or not. The more playing turns the protocol needs to complete, the slower it reaches stability.

Stopping Condition. We first examine how the various protocols behave for different values of the stopping condition ϵ. We compared all approaches for different values of ϵ.

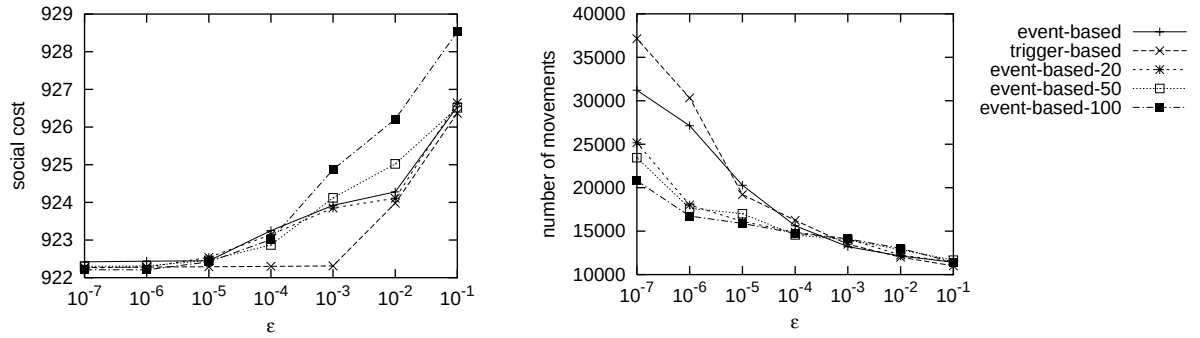


Figure 6.2: (left) Social cost and (right) movements with no coordination

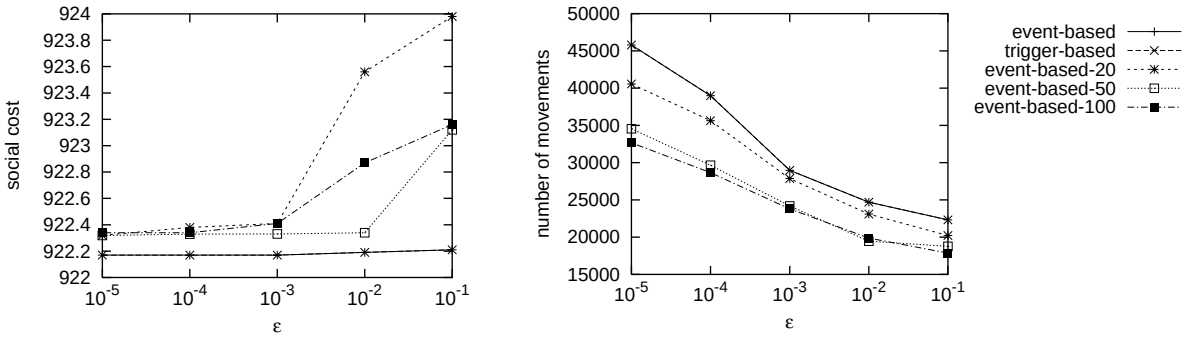


Figure 6.3: (left) Social cost and (right) movements with coordination

For the uncoordinated protocols we used different probabilities for playing (1, 0.5, 0.25). Figure 6.3(left) and Figure 6.3(left-center) report our results for probability $Pr = 0.5$. For the coordinated protocols, we set the playing probability of all peers to 1 and regulate how aggressively the peers play by changing the percentage of peers in the ordered list that are allowed to play at each round. We experimented with setting this percentage to 25% 50% and 100% of all the peers and report our results in Figure 6.3(center-right) and Figure 6.3(right) when 50% of the peers in the list are allowed to play.

We observed that for the same value of ϵ each variation presents an approximate fixed value for the social cost regardless of the value of the playing probability or the percentage of peers allowed to play. Among the uncoordinated protocols the trigger-based is the one that displays the largest overhead. Since our experiments so far concern only updates in the local query workload of the peers (queries are being issued gradually), we observe that the coordinated versions of the trigger-based and the event-based protocols behave identically. That is, the event-based protocol is initiated after each query, and since after each query the social cost also changes, the trigger-based protocol is initiated as well. Both protocols require the largest overhead we encountered yet, especially when 50% and more peers are allowed to play. However, the final value of the social cost is similar to that achieved by the corresponding uncoordinated protocols with respect to the selected ϵ.

For each of the protocols we can detect a range of values ϵ for which the protocols exhibit the best trade off between social cost and number of movements and turns. For

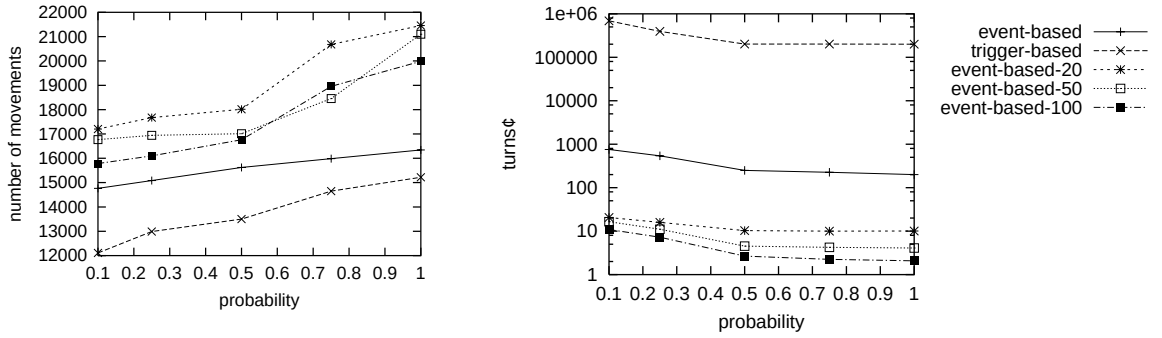


Figure 6.4: Varying probability and (left) movements and (right) turns with uncoordinated protocol

values greater than this value, the protocols have increased social cost, while for lower values the additional number of movements required is not justified by the reduction in the social cost it achieves. For the event-based protocol this critical range is around 10^{-4}. The social cost decreases significantly as ϵ decreases up to 10^{-4}, but when $\epsilon < 10^{-4}$, more than $1/2|N|$ of movements are required for the social cost to decrease by a factor of about 10^{-2}. For the uncoordinated trigger-based approach the critical value range is larger and around 10^{-3}. Due to the large overhead the trigger-based approach entails, lower values render it useless (almost all players play after each event that occurs in the system). Finally, when we use batches of events, we observe that the system works better for lower values of epsilon around 10^{-5} to 10^{-6}, where with a small number of movements we still achieve a significant reduction in the social cost. For the larger values of the parameter this variation performs much worse than the other two, especially for larger batches of events like 50 and 100. For the coordinated event-based and trigger-based variations, the value of ϵ is even greater than the uncoordinated trigger-based one and around 10^{-2} or larger. Even then, the overhead is larger with respect to the uncoordinated trigger-based game. The corresponding value for the individual cost in this case is at most around 10^{-3}. For the batch-based version we observe that we have much better behavior for lower values of ϵ and the number of moves is only marginally larger than that in the uncoordinated version. When using large batches, the moves become even the same as we have almost no unnecessary moves and unlike the event and trigger-based version, the batch-based ones work better with larger percentages of playing peers.

Playing Probability. We now select the value of ϵ for each protocol according to the previous results, i.e., 10^{-4} for the event-based, 10^{-3} for the trigger-based, 10^{-6} for all the batch-based variations and 10^2 and 10^{-4} for the coordinated ones.

The playing probability does not influence the value of the social cost we achieve considerably, as for each protocol it depends mostly on the selected value of ϵ and is around 922 for the values we selected for this experiment. Thus, we measure the number of required moves and turns.

We first discuss the uncoordinated protocols. The trigger-based approach has the largest overhead both in terms of moves and turns for all values of probability. For all the

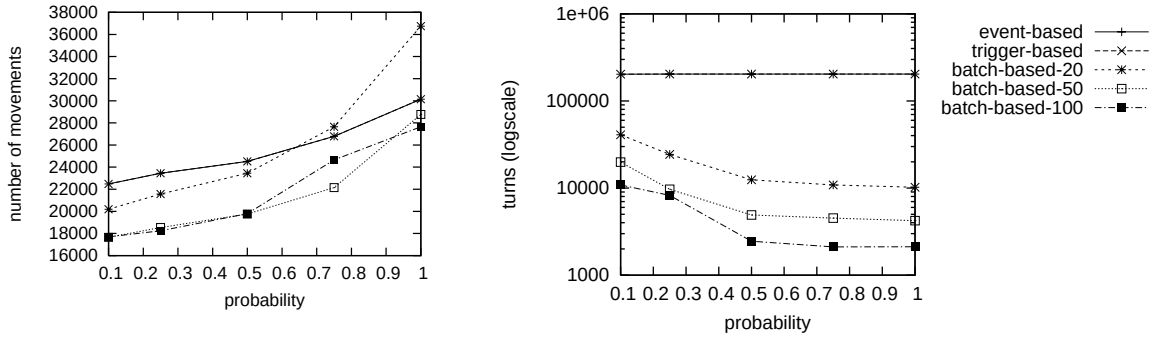


Figure 6.5: Varying probability and (left) movements and (center-left) turns with coordinated protocol

protocols, while a lower playing probability decreases the number of required movements by avoiding unnecessary ones (Fig. 6.5(center-left)), it increases the number of turns required for stability. The peers do not play every time they could, since the probability check fails frequently, thus resulting in more required turns (Fig. 6.5(center-right)). For the batch-based approaches, we observe that for smaller batches of events (such as 20) the approach behaves similarly to the event-based one. When the batch sizes are larger, then low playing probabilities cause even larger increases in the number of required turns without reducing the number of movements more significantly. Thus, the batch-based approaches are more sensitive to the probability value when it falls under some value (around 0.5).

To compare with the coordinated protocols, we consider the relationship between the percentage of peers we allow to play at each round and the playing probability. This percentage of peers can be viewed as another way to control the overhead by reducing the number of peers that play at each turn in a way similar to the probability. Furthermore, by ordering the peer requests according to their gain value, we implicitly associate their playing probability to their gain value. Thus, we set the playing probability to 1 for all peers and instead experiment with different percentages of playing peers. The results (Fig. 6.5)(center-right) and Fig. 6.5)(right)) are similar to the corresponding results for the uncoordinated version when we varied the playing probability. We notice that the coordinated version of the protocols perform better for lower values of playing probabilities than the corresponding uncoordinated versions.

Quota. To evaluate the influence of the quota protocol we consider the same values of ϵ we used in the previous experiment and a playing probability of $0.25, 0.5 and 1$. In Figure 6.8 we report our results for probability equal to 0.5 and the uncoordinated protocols. We consider that each peer is given a quota of ν moves which is replenished after TP_q events. We set $TP_q = 20$ for the event-based and trigger-based approaches. For the batch-based approaches, TP_q is defined as a product of the size of the batch. We use 5 times the size of the batch as our replenishing period. We vary ν from 1 to 15.

The social cost is again the same regardless the quota used (Fig. 6.8(left)). When ν is relatively small $\nu < 6$ then for all values of probability we observed a large number

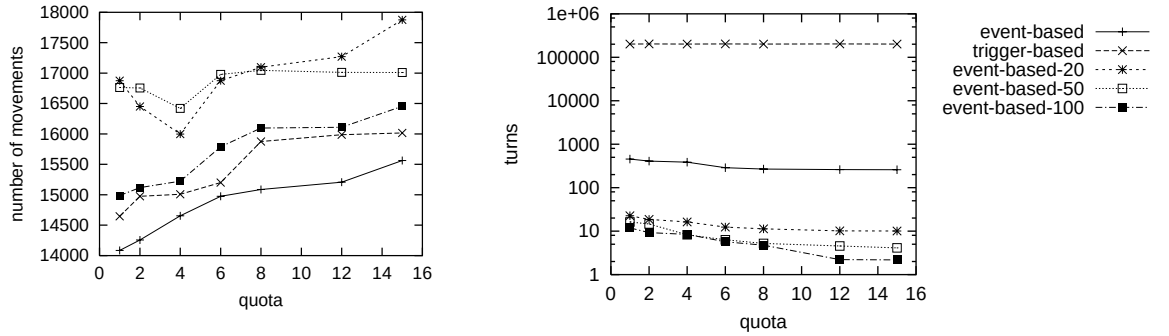


Figure 6.6: (left) Movements and (left) turns with varying quota

Table 6.4: Social cost progress

	Uncoordinated					Coordinated				
	Event	Trigger	Batch-20	Batch-50	Batch 100	Event	Trigger	Batch-20	Batch-50	Batch 100
SCost	989.32	945.56	1207.05	1876.15	2876.45	926.41	926.41	1100.25	1543.55	2454.67

of checks with the worst behaviour for the smallest probability value (Fig. 6.8(center)). There is a small gain in the number of movements that is slightly decreased especially in the case of the trigger-based approach where the communication overhead is considerably larger. We observed that there is an area of quota for each approach and each probability value in which our protocols behave the best. For example, for the event-based approach for $\nu > 6$ and $\nu < 12$ and probability of 0.5 we observe the best system performance (a balance between the number of movements and the number of checks). In general, for larger probabilities and protocols that make more movements, the use of quota combined with the playing probability can significantly improve the performance. For example, for the trigger-based approach with probability 0.5, while with quota $\nu = 10$ we have a decrease of about 10% in the number of movements compared to not using any quota, while the number of checks is not increased. This proves that we mostly avoided unnecessary movements.

Similarly to the observation regarding the playing probability, the use of a low movement quota is also beneficial for the coordinated protocols. While the trigger-based protocol in the uncoordinated version required a quota of about 10 moves, in the coordinated version the protocol works better when the quota is around 12. Similar conclusions apply for the batch-based approaches.

An advantage of the controlling parameters is that they can be dynamically adjusted locally by each peer. By observing its processing and communication cost caused by its moves and the fluctuations in its utility function, a peer can achieve appropriate tuning. For example, a low individual cost may cause the peer to decrease its ϵ value, while a large number of moves that do not considerably improve its cost may cause the peer to increase ϵ. Similarly, if a peer observes too many moves, it may decrease its playing probability, or in the presence of frequent updates increase it to react faster to changes. Similar observations may be applied to tuning quota.

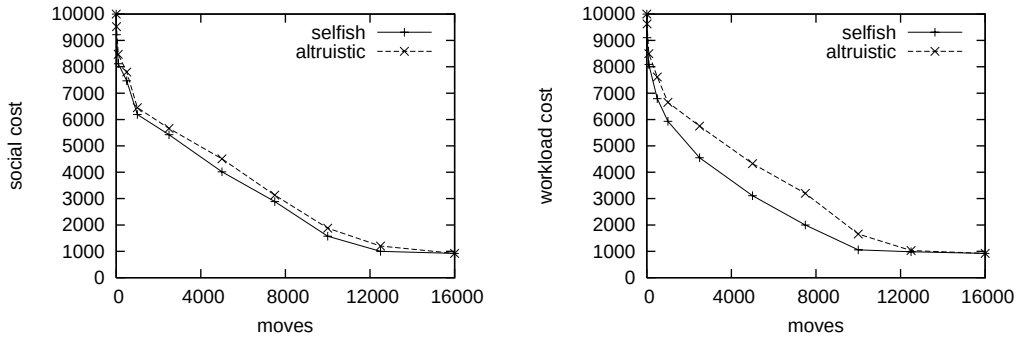


Figure 6.7: (left) Social and (right) workload cost through progressing rounds

Progress Per Round

It appears from our experiments so far that the batch-based approaches perform the best. They reach the same social cost as the other two while requiring less movements and less turns, thus they are the ones with the lowest overhead. However, we need to consider that the reported social cost value is reached at the end of the protocol when we reach stability. While the protocol is still active, since the batch-based approach is the one that acts the last to correct the increase in the cost caused by updates, we expect that it will have the worst behaviour.

To demonstrate this we measure the average social cost per turn (Table 6.4). Among the uncoordinated protocols the trigger-based has the lowest average social cost (945.56), with the event-based approach coming a close second (989.32). All batch-based approaches perform considerably worse, and for 100 batch size, we have the worst performance. The main advantage of the coordinated version seems to be the very fast reaction to changes. Both the trigger-based and the event-based version work more efficiently than the uncoordinated trigger-based approach with an average cost of 926.41. The advantage is explained because of the policy the coordinated protocol follows by applying ordering in the relocation requests. By granting the requests with the larger gain, the protocol seems to avoid some unnecessary moves that would have to be retracted in the future. This is also the only difference between the two versions of the trigger-based protocol as they are both triggered by the same events that affect the social cost. The batch-based approaches also perform much better, since they take into account global rather than just local events.

Protocol Variations

Varying Probabilities. So far in our experiments, we use the same value of probability for all the peers. This does achieve a sense of fairness since we give all players an equal chance of playing. However, one may need to consider that all players are not equal, they have different sizes and different levels of demand. Also, some may update their content or workload more frequently than others. Thus, we may want to tune their playing probability accordingly. Besides tuning the playing probability, another way to

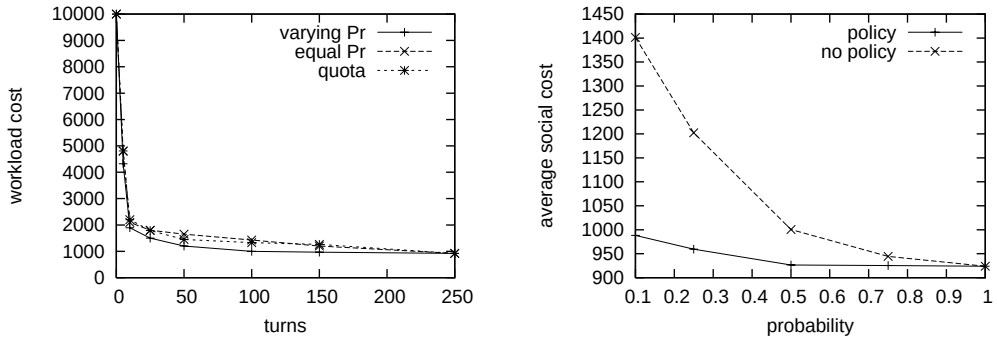


Figure 6.8: (left) Workload cost with different probabilities for each peer and (right) social cost with policy and no policy

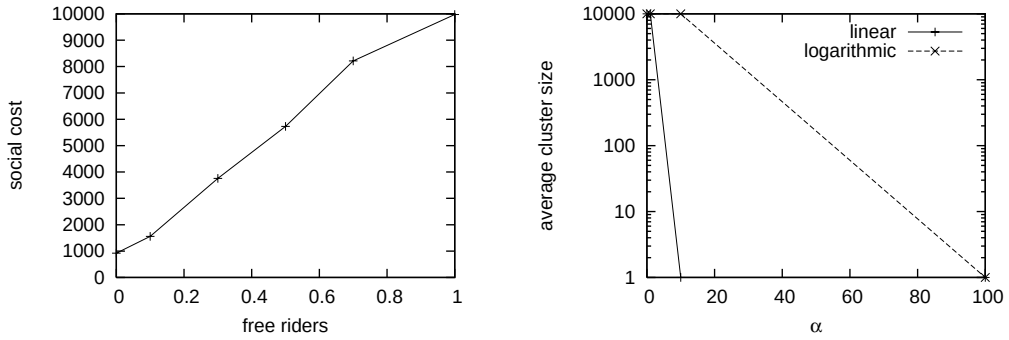


Figure 6.9: (left) Influence of free riders and (right) influence of α

determine the number of times each player plays is with the moving quota. The quota in contrast to tuning the probability according to the levels of demand ensures fairness. In this experiment, we increase the level of demand of 20% of the peers to double the average level of demand of all and we set their playing probability to 1 while the probability for the rest of the peers is set to 0.5. We compare it to an approach where all peers have the same probability 0.5 regardless their demand level and another approach where all players have playing probability 1, but movement quota is utilised. We used the setting from the previous experiment and set $\nu = 10$ and $TP_q = 20$. We measure the average workload cost each peer observes at its turn.

Giving the more demanding peers larger playing probability improves the workload cost much faster than when all players have the same probability. The more impressive result is that this configuration also decreases the number of movements significantly. While with equal probability we require about 15000 movements, the varying probability scheme reduces them to about 11500, more than 1/4 of the peers save one move. The movement quota approach performs similarly to tuning the probability. Since the quota is replenished after a number of events and the more demanding peers are influenced by more events, their quota is replenished faster than that of the other peers thus allowing them to play more often. Thus, quota can be used when no information about the level of the demand of the peers is known and we cannot tune the probability accordingly. The additional overhead is that we need to tune the quota policy instead.

Ordering of Relocation Requests. The coordinated protocols besides reacting faster to changes also have the advantage of being able to order the relocation requests and granting first the ones that result in the largest reduction in the social cost. To evaluate how much they benefit from this ordering we compare the event-based coordinated protocol for different percentages of playing peers with a corresponding protocol in which no ordering among the requests is utilised. That is, $x\%$ of the requests are selected randomly to be granted. We measure the average social cost per round (global event) achieved by both approaches. While the ordering does impose an additional cost, it also improves social cost significantly (up to 15%). The improvement is better observed when a small number of requests is only granted (small x), while their performance becomes identical when almost all requests are satisfied.

Events Not Affecting the Cost. Thus far, our experiments involved events in the type of issuing queries that always yielded to a change (improvement or not) of the cost of some peers. To differentiate between the trigger-based and the event-based protocols, we performed an experiment which we added events that did not affect any cost functions. Such events include updates of content that is not queried by anyone so far, or insertion of new peers that have no content of interest to any of the other peers so far. In this case, the trigger-based protocol outperforms the event-based one in term of turns, while they still require the same number of movements and yield the same social cost. The difference is the number of times the event-based protocol is initiated without any reason causing the excessive evaluation of cost functions at each peer. The batch-based events are also influenced by these types of events, but since they consider them in batches, the problem is not so significant, especially if the rate of such events is not that large.

Summary:

- The value of ϵ is the main factor controlling the value of the achieved social cost. For each approach we can define an appropriate value lower than which the improvement in the social cost does not justify the number of required moves. The playing probability and quota reduce the number of movements while increasing the number of turns.

- The trigger-based approach is the most expensive among the uncoordinated approaches. Its main advantage is that it reacts much faster to any change. The batch-based approaches present the lowest overhead but also react much slower to changes. The event-based approach seems the most reasonable compromize between overhead and fast reaction to changes.

- Variations that adjust the probability according to the peer's level of demand manages to reduce the movements and increase the workload cost faster (in less turns).

- The coordinated protocols achieve approximately the same social cost value while imposing a much larger overhead, especially with regards to required turns. Their main advantage is that they react to changes even faster than the uncoordinated

Table 6.5: Cluster formation

	Moves			Clusters			Cluster Size		
	Self	Alt	Mix	Self	Alt	Mix	Self	Alt	Mix
Symmetric Scenario									
i	15358	15436	14900	10	10	10	1087.5	1100	1100.5
ii	14786	14365	14657	10	10.5	10	1079	1109.5	1125.5
iii(a)	9654	9812	9867	10	10	10.5	1112	1085.5	1119.5
iii(b)	10211	10210	11270	10	10.5	10.5	1010.5	1056.25	1011.75
iii(c)	9841	9554	9456	10	10.5	10	1012.25	1009.5	1016.5
iv	0	0	0	10	10	10	1100	1100	1100
v	0	0	0	10	10	10	1100	1100	1100
Asymmetric Scenario									
i	16875	16758	17008	89.25	89.5	90.25	110.75	110.9	111.15
ii	16257	16109	16431	89.25	89.15	90.01	111.9	111.75	111.45
iii(a)	9405	9115	9650	90.1	90.2	90	111.05	110.5	110.33
iii(b)	9180	9320	9125	89.9	90.05	90.2	111.15	110.9	110.67
iii(c)	8502	8745	8790	89.75	89.5	90.1	110.5	110.24	111
iv	5865	6120	6275	90	90.15	90.15	111.4	111.75	111.05
v	6015	6104	6436	90	89.9	90	110.9	111.09	110.89
Random Scenario									
i	19450	19710	19340	1	1	1	10000	10000	10000
ii	0	0	0	1	1	1	10000	10000	10000

trigger-based approach and use requests ordering yielding the best average social cost. The event-based approach is initiated by events that do not affect the cost, thus making it even more expensive in terms of turns with respect to the trigger-based version.

6.6.2 Cluster Formation

In this set of experiments, we start from a random peer configuration and examine whether the peer reformulation protocol leads to the desired cluster configuration. In this and the rest of the experiments, we use the uncoordinated event-based protocol with Pr equal to 0.5 for all the peers and $\epsilon = 10^{-4}$ as it offers the best tradeoff between overhead and performance. Let M be the number of peer categories in the system for each scenario, i.e., for the symmetric scenario $M = 10$. We consider five different cases for the initial system configuration: (i) each peer forms its own cluster; (ii) all peers form a single cluster; (iii) peers are randomly distributed to m groups and we discern for different values of m the subcases: (a) $m = M$, (b) $m < M$ and (c) $m > M$; (iv) peers are clustered according to their content and (v) peers are clustered according to their workload.

The peer that issues a query each time is selected uniformly at random from all peers in the system. We allow the system to run multiple queries and check whether the system

Table 6.6: Global cost measures for cluster formation

	SCost			WCost			SCont			WCont		
	Self	Alt	Mix	Self	Alt	Mix	Self	Alt	Mix	Self	Alt	Mix
Symmetric Scenario												
i	10.07	10.23	10.15	9.97	10	10.15	9.67	9.85	9.84	9.32	9.45	9.65
ii	10.42	10.67	10.93	10.2	10.28	10.65	9.45	9.63	9.58	9.52	9.35	9.61
iii(a)	10.4	10.97	11.35	10.55	10.86	10.93	9.98	9.64	9.69	9.22	9.28	9.53
iii(b)	11.45	10.81	11.59	10.75	10.65	10.9	9.82	9.36	9.42	9.29	9.22	9.22
iii(c)	10.65	10.83	11.72	10.35	10.65	11.93	9.32	9.22	9.29	9.13	9.16	9.14
iv	10.09	10.09	10.09	10.09	10.09	10.09	9.94	9.94	9.94	9.94	9.94	9.94
v	10.09	10.09	10.09	10.09	10.09	10.09	9.94	9.94	9.94	9.94	9.94	9.94
Asymmetric Scenario												
i	919.9	922.35	924.85	914.77	920.01	925.86	987.45	974.15	992.09	982	989.76	985.05
ii	924.75	929.25	926.05	919.87	925.5	931.43	974.01	959.25	964.26	971.17	948.89	963.25
iii(a)	922.86	926.41	941.02	917.9	921.12	937.09	947.05	961.67	961.08	957.26	954.25	958.25
iii(b)	922.65	921.15	924.66	918.07	917.78	924.67	967.33	965.09	964.23	962.25	954.57	962.44
iii(c)	923.08	924.44	926.12	920.17	922.99	924	978.02	967.15	976.45	971.67	962.32	969.35
iv	929.05	926.10	924.57	926.48	922.33	923.08	966.14	967.86	968.1	964.56	963.47	963.18
v	917.95	919.05	920.69	915.05	919.7	918.6	952.21	954.33	956	952.39	951.05	952.75
Random Scenario												
i	11.33	11.33	11.33	11.33	11.33	11.33	11.33	11.04	11.04	11.04	11.04	11.04
ii	11.33	11.33	11.33	11.33	11.33	11.33	11.04	11.04	11.04	11.04	11.04	11.04

reaches an equilibrium and if so, what is the total number of moves the peers made, the number of clusters they formed and the average size of those clusters (Table 6.5). We also provide the achieved social and workload cost as well as the respective social and workload contribution (Table 6.6).

In all scenarios, all strategies reach an ϵ-Nash equilibrium and form the desired number of clusters regardless of the number of clusters in the initial configuration. Thus, our protocol does not require a predefined number of clusters, but dynamically determines the appropriate number and may also change it over time to cope with updates as the desired number may also change over time. Selfish and altruistic peers do not differ significantly and a mixed strategy usually requires more moves, but has the same social cost.

Furthermore, for the symmetric peers both social and workload cost only depend on the cluster membership cost; the cost for the recall is zero, since all relevant data are located within the cluster (Table 6.6 lines 1-7). Similarly the recall term in the social and workload contribution has its maximum value and the peers only pay for their membership cost. Moreover, according to our case studies for the given $\alpha = 10$ the social cost achieved is very close to the social optimum. If we consider that all our peers are perfectly symmetric and all clusters have the same number of peers (1000) as in Case Study II, then the lowest individual cost for a peer is equal to: $10^{-3}\alpha$ and the social cost is: $10\alpha = 10$ which is about the same with the cost our approach achieves. The value is not exact because the

peers and the clusters are not perfectly symmetrical as in the study case.

For the asymmetric scenario, the recall factor in this case is not zero, and we observe higher social cost and lower social contribution. Also, since the queries are not uniformly distributed among the peers, the social cost differs from the workload cost (similarly for the respective contribution measures). When the symmetric peers are clustered according to their content or workload, there is no need for change since the appropriate clusters are already formed (Table 6.5 lines 6-7). For the asymmetric peers, both configurations are not stable, i.e., the peers can improve their cost, though they require less moves to reach stability than the other initial configurations. Thus, we deduce that relying only on content or workload information is not enough to provide the appropriate clustering and thus, our policies take into account both.

Finally, for the third scenario since no clear number of desired clusters can be deduced, we consider all peers as single-membered clusters (Table 6.5 line 16) and all peers in one cluster (Table 6.5 line 17). This scenario is similar to our first case study where no underlying clustering exists. Due to the small value of the membership cost (logarithmic) and the α parameter that is set to 10, the peers converge towards a single cluster in both cases as we showed in our case studies and achieve a social cost around the optimum. In particular, the second case requires no moves since all peers are already in one cluster. For the same α, if we use the linear function for the θ, then the single cluster is no longer the best configuration and the peers tend to split into smaller clusters. Similarly, a larger value of α would force the peers to form single membered clusters. For example, we repeated the experiment setting $\alpha = 100$. In this case, the first configuration consisting of single member clusters was the one both initial configurations converged to. Consequently, according to the importance we want to give to the membership cost, we can accordingly tune the α parameter to suit our needs. Figure 6.9(right) reports the average size of the clusters that are created for different values of α when we start with an initial configuration of all peers in a single cluster for the linear and the logarithmic θ function.

In all three scenarios, we did not observe significant differences between selfish and altruistic peers. When a mixed strategy is used, we see that, in general, more moves are required so as to reach a stable state. However, the social cost in the resulting state is not significantly different.

Social vs Workload Cost. We also measured the progress of the social and workload cost as the peers realised more moves, i.e., during the game and until reaching a stable state. To show the difference in the two measures we used varying probabilities for each peer for the playing probability Pr. Furthermore, we selected a percentage of peers (20%) as demanding peers and doubled their query workload, and did the same with respect to content to a different 20%.

The more demanding peers are the ones that move first to accommodate their query workload. As Fig. 6.9(center-left) shows, the workload cost decreases faster as the demanding peers are the ones that dominate the moves. After the demanding peers find the appropriate cluster, even if they do have a higher probability to move they do no

longer have anything to gain from the move. The social cost that treats all peers equally decreases linearly through all rounds (Fig. 6.9(left)). When we consider altruistic peers in this experiment we observe that the workload cost is not reduced much faster. The peers that have large size (i.e., more data) do not have an accordingly higher probability to move. The demanding peers that do move more move towards clusters that they can offer to, not caring if their own workload will be satisfied. To have similar results for the altruistic peers as the ones for the selfish we would have to increase the probability Pr for those peers with large size.

Hybrid Peers. Besides the selfish and altruistic strategy we also have hybrid peers that take into account both individual cost and contribution when moving to another cluster. From repeating the same experiments for hybrid peers, we observe behaviors similar to the selfish and altruistic ones. The number of moves required to reach stability is slightly greater ($\approx 10\%$ greater) than when using the purely selfish or altruistic strategies, but the number of clusters formed and their size is not influenced. Another difference is that hybrid peers reduce the social cost of the system faster at first, compared to selfish ones, but since they have more options than purely selfish or altruistic peers, they require more fine-tuning moves.

Free Riders. Free-riders are peers that use data items offered by others, but never contribute any content. We model a free-rider n as a selfish peer with $r(q, n) = 0$, for all q in Q. We want to examine how the appearance of free riders that is a phenomenon often met in p2p [7] influences the behaviour of the system. As the percentage of free riders increases, the social cost also increases (Fig. 6.9(center-right)), and finally the system degenerates to a system where each peer forms a cluster by its own since it has nothing to gain from connecting to other peers.

Summary:

- Applying the relocation policies enables the system to reach a stable state and the peers to form the desired clusters for a variety of different starting system configurations and both symmetric and asymmetric peers.
- For symmetric peers the social cost achieved depends only on the membership cost and is close to the social optimum (≈ 10) for the given setting. Similarly, when no underlying clusters exist, the social cost of the system again is very close to the social optimum.
- Clustering based solely on the content or workload distribution may be enough for dealing with symmetric peers, but both need to be considered for more general cases (i.e., asymmetric peers).
- The value of α determines the significance we put in the membership cost and is the main factor determining the number and size of the produced clusters.
- When the playing probability of the most demanding peers is increased, the workload cost is reduced faster than the social cost that treats all peers equally. (Similarly

for peers with more content when dealing with altruistic peers).

- Hybrid peers require about 10% more moves to reach stability but reduce the social cost faster during the first turns compared to purely selfish or altruistic strategies.
- The presence of free riders affects the social cost of the system negatively.

6.6.3 Cluster Adaptation

In this set, we start from a "good" cluster configuration for given content and workload, and examine how well the reformulation protocol adapts to changes in the system conditions. We consider content, workload and topology updates (peers join/leave the system). The initial system configuration consists of clusters of peers complying to the symmetric scenario, i.e., maintaining data and posing queries belonging to a single category. We use mixed populations of both selfish and altruistic peers with different ratios, i.e., from all selfish peers ($100\%selfish$) to all altruistic ($0\%selfish$). We consider four different update scenarios:

- Popular existing category ($Sc1$): $x\%$ of peers randomly distributed across all clusters change their query workload to the specific category.
- New popular category ($Sc2$): $x\%$ of the peers become interested in a data category that does not have a corresponding cluster. We assume that content regarding this data category was already distributed among peers but no queries were issued concerning it.
- k Popular existing categories ($Sc3$):k data categories become more popular and $x\%$ of the peers are assigned one of those categories at random.
- k categories deletion ($Sc4$): k categories seize to be popular, i.e., different percentages of peers from k clusters change their query workload to one of the other existing categories.

We compare the social cost our policies achieve for all scenarios to not applying any changes at all. We assume the logarithmic as the theta function, $Pr = 0.5$ and $\epsilon = 10^{-4}$.

In all scenarios, our policies significantly reduce the social cost compared to the social cost the system would exhibit if no measures were taken to cope with the changes, i.e., if we applied no changes in the cluster memberships and maintained a static overlay. For example, the cost is reduced up to 1/3 of the cost of the static overlay for $Sc1$ (Fig. 6.11(left)). Also, the more the selfish peers in the system the better our policies perform. This is because the changes occur in the workload and not the content of the peers. For the altruistic peers to issue a relocation request, a large percentage of peers needs to change its workload. This percentage is even larger than expected, since the changes affect peers across all clusters, and for an altruistic peer to have a gain to move to another cluster, a large number of peers in both its current and the new cluster should change

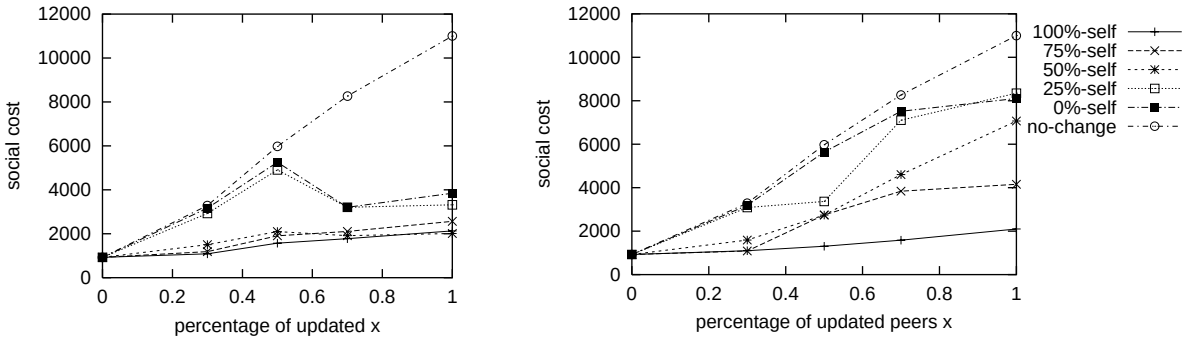


Figure 6.10: Social cost for different percentages of updated peers (query workload) for (left) scenario 1, (right) scenario 2

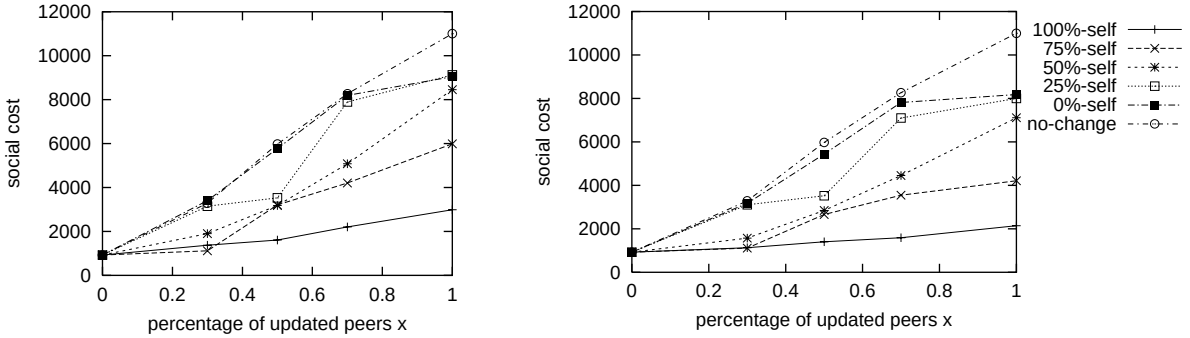


Figure 6.11: Social cost for different percentages of updated peers (query workload) for (left) scenario 3, and (right) scenario 4.

their workload. Also, we have to note that the social cost never reaches its initial value since a cluster of bigger size is created by this update.

The second scenario performs better than the first (Fig. 6.11(center-left)), as the relocation requests are more evenly distributed among the clusters, since the peers maintaining data of the new category are also distributed across the clusters unlike the first scenario in which they were all concentrated in one. Figure 6.11(center-right) shows the third scenario for $k = 4$, which is similar to the second. The slightly worst social cost is due to the existence of more asymmetric peers. Figure 6.11(right) shows the corresponding results for the last scenario when $k = 4$.

Content Changes. A conclusion from all four scenarios is that when only the workload of the peers change, selfish peers contribute to better performance since altruistic peers take more time to react to such changes. We repeated the same experiment for all four update scenarios, when the peers change their content instead of their query workload. As Figure 6.13 shows, when the content changes, then the more altruistic peers exist in the system, the better the system performance. In particular, the behaviour of the social cost when the altruistic peers increase is completely analogous to that of the social cost when selfish peers increased in the case of changes in the query workload.

Hybrid Peers. While selfish peers react fast to workload changes, altruistic ones react faster to content changes. Hybrid peers consider both selfish and altruistic criteria to

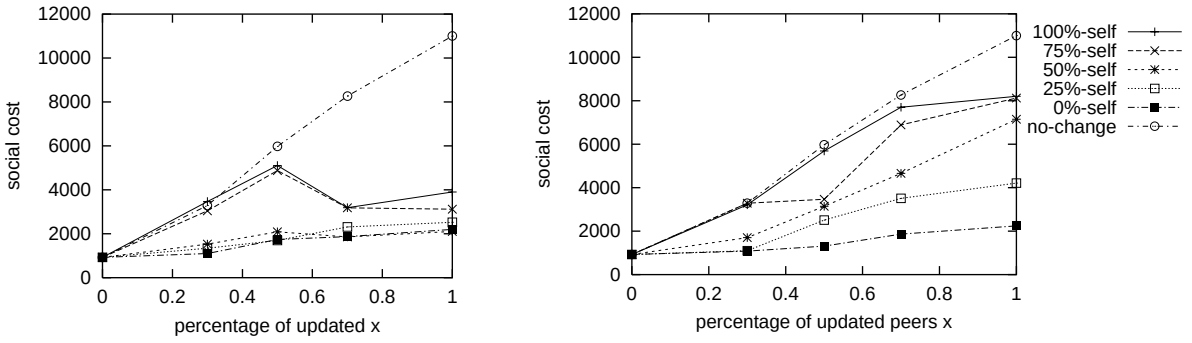


Figure 6.12: Social cost for different percentages of updated peers (content) for (left) scenario 1 and (right) scenario 2.

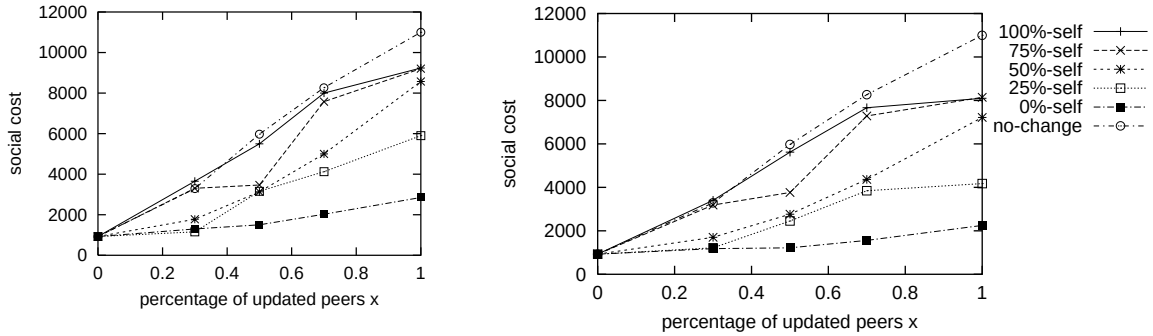


Figure 6.13: Social cost for different percentages of updated peers (content) for (left) scenario 3 and (right) scenario 4.

determine their cluster membership. We evaluate system performance when all peers are hybrid with different d for update scenario 2, when the changes refer to workload (Fig. 6.15(left)) and content (Fig. 6.15(left-center)). When d is small, the peers act more like altruistic peers being affected from content changes more than from workload changes. For d closer to 1 their behavior is similar to selfish peers. When both selfish and altruistic criteria are taken into account equally ($\beta = 0$), hybrid peers react faster than selfish peers to content changes but slower than altruistic ones and vice versa for workload changes. That is, if changes in both content and workload happen with the same frequency then adapting a hybrid strategy.

Workload and Content Change. We consider changes in the workload that also cause changes in the content of the peers. If we assume that a peer becomes interested in a topic, i.e., changes its query workload to a specific data category, then this peer will also acquire data of this category. We consider that 50% of the peers change their query workload to a category that has not any data in the system so far. Then, we measure the social cost as the number of turns increases, while considering that these same peers change incrementally 10% of their content at each turn to that particular category. Figure 6.15(right-center) reports our results. In the first rounds, the social cost increases as there are not enough data to satisfy the query workload of the updated peers. As the number of turns increases and the peers change more of their data, the updated peers tend to

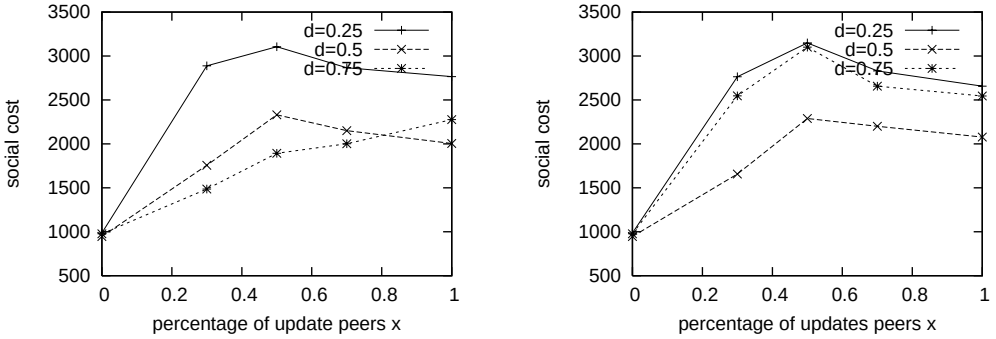


Figure 6.14: Social cost for (left) workload and (right) content changes for hybrid peers.

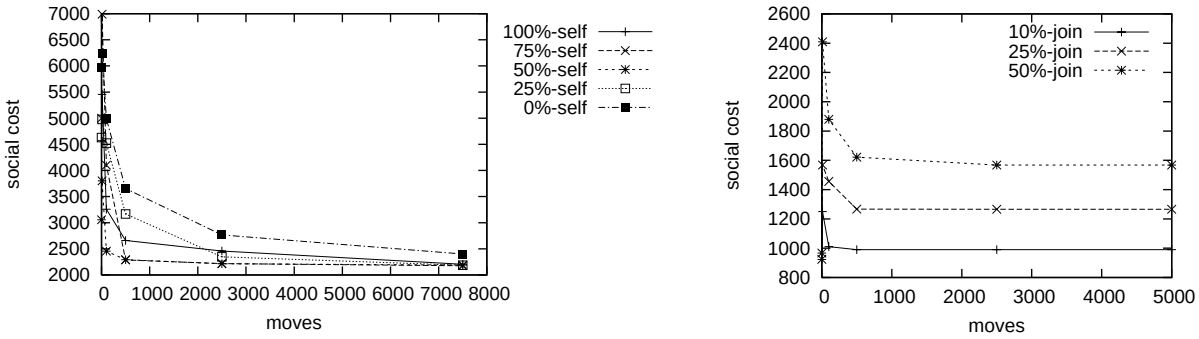


Figure 6.15: (left) Different percentages of updating both workload and content, and (right) peers joining the system.

form a new cluster, effectively reducing the social cost.

Peers Joining and Leaving the System. Besides content and query workload updates that peers perform locally, another type of changes concerns peers joining and leaving the system. We examine how the system adapts when a varying percentage of peers joins the system. The peers that are selected to join are selected uniformly from all the categories and are all selfish. When peers first join, they form a cluster by their own. As they pose queries and are gradually informed about the other clusters, they move to the one that matches their workload. Thus, there is initially a considerable increase in the social cost because of the burst of new peers, but as the rounds progress and the peers join their cluster this is corrected (Fig. 6.15(right)). However, the social cost remains slightly larger than its original value since the system now has more peers. When peers leave the system, the social cost is actually reduced as the number of peers is reduced. In this case, no immediate adaptation is required from the reformulation protocols. If a large percentage of the members of a cluster leave it, then the other member may require to move to other clusters to find more results to their queries, or peers from other clusters might be motivated to move to this cluster to exploit the low membership cost.

Cluster Reformulation vs Clustering from Scratch. Another alternative to cope with changes, rather than applying the reformulation policies, would be to apply the clustering procedure from scratch so as to take into account the new data and query workload distributions. Such an approach is expected to result in a better clustering

scheme with a lower social cost than the reformulation protocols. However, we argue that this would entail a much larger communication cost with respect to the gain in the social cost it would offer. To demonstrate this we consider the second update scenario where a new category of data becomes popular. To apply clustering from scratch, we first "decluster" the peers, considering each as forming its own cluster and then apply our policies.

We measure the social cost in this case after the system has reached stability for various percentages of updates peers (Fig. 6.19(right)). Compared to the case where we apply the reformulation policies for adaptation only (Fig. 6.11(center)), the social cost in this case is lower up to 10% for all selfish peers. Furthermore, the re-clustering is able to perform well even in the case of all altruistic peers in which adaptation does not work when only the workload changes. However, the re-clustering technique is much more expensive. We measure the number of turns for both cases to achieve a stable state and also the number of movements to different clusters that take place. The re-clustering technique requires for both all selfish and all altruistic peers about 250 turns, while the reformulation policies reach stability in only 10 turns. As far as movements, the reclustering realises $O(N)$ movements, where N is the number of peers. Even in the case where only 10% of the peers make the change in their workload, it still requires over 10000 movements, whereas the number of movements in reformulation, depends on the number of the updated peers and is around 1100 when 10% of the peers update their workload, while it remains below 10000 even for 100% of updated peers.

Summary:

- The relocation policies are able to cope with changes efficiently for a wide variety of update scenarios affecting the content, workload and population of peers. Their application reduces the social cost of the system up to 1/3 compared to a static overlay in which the changes are ignored.

- Selfish peers react faster to workload changes, while altruistic ones are more sensitive to content changes. Hybrid peers perform according to their b value and provide a compromise between sensitivity to workload and content changes.

- When peers join the system the social cost is initially increased but the policies are able to fast accommodate the new peers to their appropriate cluster.

- Re-clustering from scratch taking into account the changed conditions yields a system with a social cost up to 10% lower than applying the relocation policies and is not affected by the existence of selfish or altruistic peers that do not react to changes in the content or workload respectively. However, re-clustering imposes a considerable overhead in both turns (250 turns instead of 10 in cluster adaptation) and moves (always $\geq |N|$, while in cluster adaptation it depends on the number of affected by the changes peers).

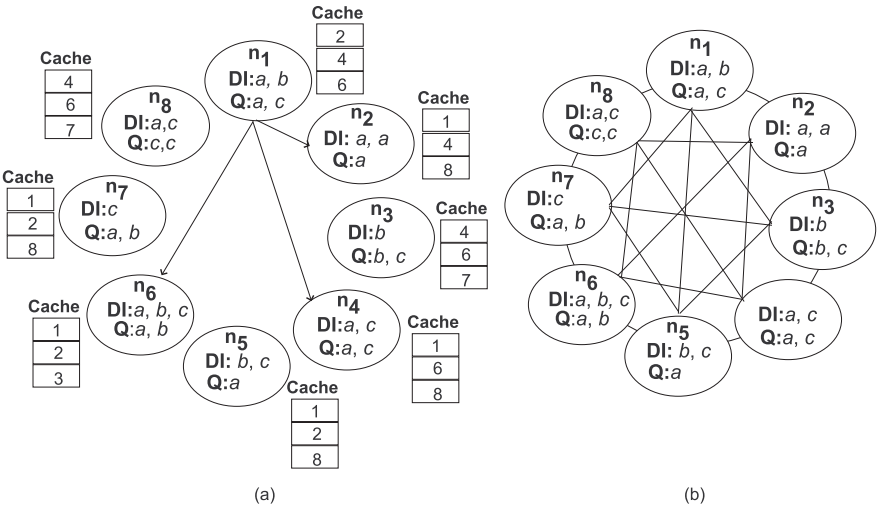


Figure 6.16: (left) A caching scheme and (b) the corresponding clustered scheme

6.6.4 Comparison with a Caching Scheme

Caching is widely used to improve performance in distributed systems. It either refers to caching the results of previous queries to facilitate future similar queries or caching the peers that provided answers to previous queries and forwarding future queries to them. We consider a system based on the second idea and similar to [120]. The cache entries form an implicit overlay network in which peers that are considered to share similar interests are connected. Figure 6.16 shows a simple example with six peers and their content and query workloads, and how the peers would be connected (a) if a caching scheme was used or (b) if clustering was applied. Queries are first forwarded to the peers in the query's origin cache and if the results are not satisfying, then the peer resorts to other more inefficient methods of search, i.e., flooding.

In an interesting variation of this strategy (*transitive*), a peer sends its queries to the peers in its cache as a message with a $TTL = 2$ prompting the peers that receive them to also forward them to the peers in their own caches. This is based on the idea that a peer has common interests with the peers that have common interests with the peers in its own cache, and enables the peers to discover more peers with results to their queries without flooding.

After each query, the peer that issued it updates its cache. For the peers already in the cache, their recall value (or any measure indicating their usefulness) is updated according to the results for the latest query. For the peers which have returned results and are not in the cache an update cache policy is deployed. We discern between two variations. In the first variation (*update(1)*), the peer with the highest recall from the ones not in the cache is selected. If there is enough space in the cache then it is just added. Otherwise, it replaces the peer with the lowest recall value that was found in the cache. The alternative $update(x)$, selects the x new peers with the highest recall for either adding them or replacing them in the cache. This alternative is able to adapt faster to

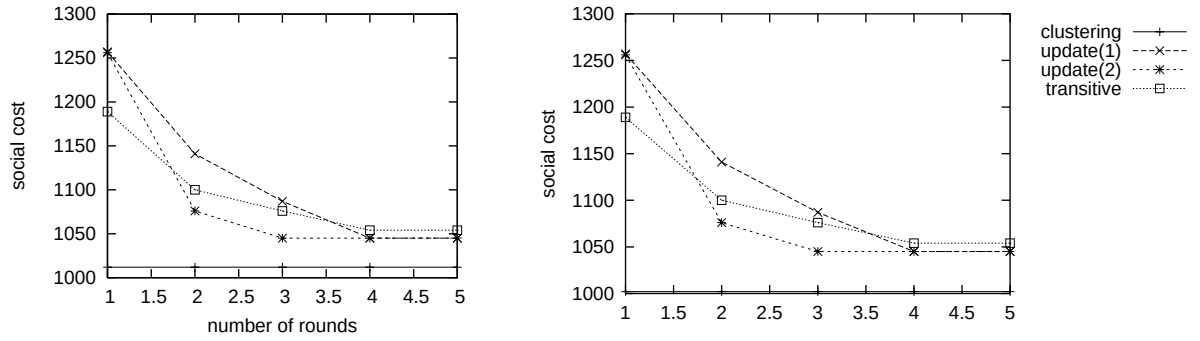


Figure 6.17: Caching for symmetric peers vs clustering (left) with linear and (center-left) logarithmic θ function,

changes in the interests of a peer as it maintains a fresher cache.

We want to compare our clustering scheme to a cache-based scheme like the one we described above. We model the cache-based system by considering a random graph where each node representing a peer has a maximum out-degree dgr of which a constant number of 3 links is devoted for the formation of the graph, while the rest $dgr-3$ links are allocated as spaces in the peer's cache. To accomplish a fair comparison with our scheme, we set dgr equal to the number of links a peer establishes in its cluster. We first consider a linear θ function for the topology within each cluster, i.e., a fully connected graph and then, a chord-like topology in which the number of links is equal to $log(|c_i|)$.

Symmetry. The cache-based scheme is expected to work better than clustering when the interests among peers are not symmetric. That is, when the peers offer content different from their query workload. Each peer has a cache with the peers that maintain its content of interest, while it belongs to the caches of peers that are interested in its own content. In contrast, for symmetric peers, while the caching scheme needs for both peers to discover each other, in clustering, it suffices for one of them to discover the other to establish the bidirectional link between them (i.e., for them to become members of the same cluster).

To demonstrate this we repeat the first experiment of our first set of experiments, for both symmetric and asymmetric peers. We measure the social cost for all caching scheme variations, setting x equal to 2, after intervals in which about 1/5 of the local query workload is issued by each peer and compare it to that of the clustering scheme. As the membership cost for the caching scheme we consider the cost of the dgr links and as the loss in the recall, the percentage of results a peer obtains through flooding rather than its cache. We assume same size clusters and compare the caching scheme for $dgr = 1000$ with fully connected clusters, and for $dgr = 10$ with chord-like ones.

Our results confirm our expectation but also indicate that the topology within the clusters that determines their membership cost also plays an important factor. In particular, when θ is logarithmic yielding a low membership cost, clustering outperforms caching for both symmetric (Fig. 6.18(center-left)) and asymmetric peers (Fig. 6.18(right)). For symmetric peers, with $dgr = 10$ links, the efficiently structured clustering scheme reaches

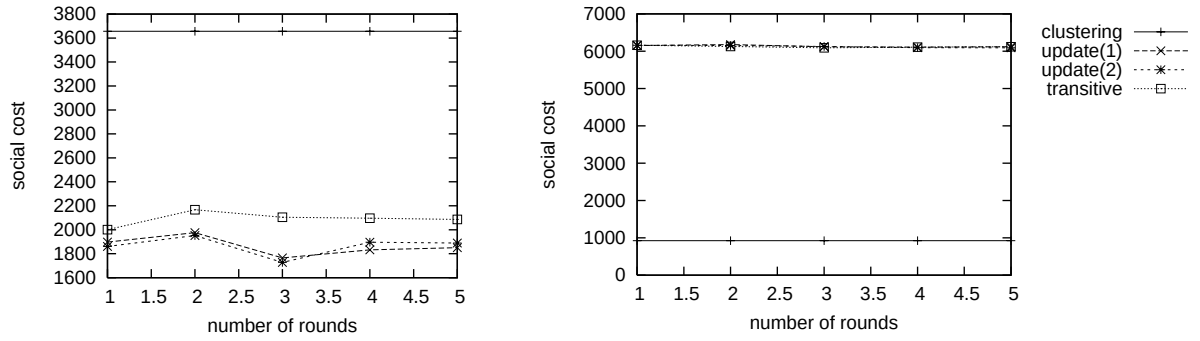


Figure 6.18: Caching for asymmetric peers vs clustering (left) with linear and (center-left) logarithmic θ function.

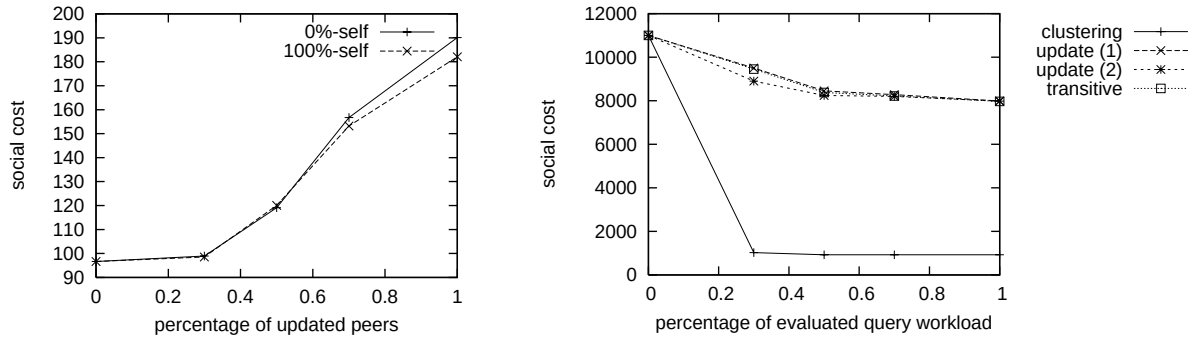


Figure 6.19: (left) Social cost with re-clustering and (right) clustering vs caching for changes in the workload.

all 1000 peers of the same category, while in caching only 10 such peers are reached. In contrast, when θ is linear, the difference between the performance of clustering and caching becomes much smaller for symmetric peers (Fig. 6.18(left)), while caching outperforms clustering for the asymmetric ones (Fig. 6.18(center-right)). Both replace policies behave similarly since the peers do not change the category of their workload or content. The transitivity property is only useful when the peers are symmetric, and again it does not suffice to achieve the performance we have with clustering.

Coping with Changes. We consider the asymmetric scenario and the peers organised in a chord-like topology. We repeat scenario 3 of the third set of experiments and measure the social cost after different percentages of the local queries workload have been processed (Fig 6.19(right)). Clustering adapts to changes more efficiently. While caching changes one to two neighbours, clustering changes all neighbours at once, thus, achieving lower social cost faster.

Summary:

- Caching is more appropriate for asymmetric peers, while clustering is more suitable for symmetric ones. However, deploying efficient topological structures that significantly reduce the membership cost within the clusters, such as chord-like topologies, enable clustering to outperform caching even with asymmetric peers.

- Clustering reacts faster and more effectively to changes. While caching changes a subset of neighbours (one to two) after each query, when the cluster membership changes the entire neighbour set is changed.

6.7 Summary

We have modelled peers in a clustered overlay as players that dynamically change the set of clusters they belong to according to an individual utility function, which is based on a cluster membership cost and query recall. We modelled both selfish peers that aim at minimizing their individual cost, i.e., maximizing their recall, and altruistic peers that try to maximize their contribution to other peers. In addition, we have defined measures for evaluating global system quality. We have proposed corresponding relocation policies for both selfish and altruistic peers. Our experimental results showed how by following these policies, the peers can change the clustered overlay to reflect the current system conditions thus, gradually correct system performance. Furthermore, our results indicated that the proposed policies can also be used for the initial construction of clusters, when the underlying data distribution permits it.

The results presented in this chapter appear also in [74] and [76].

Chapter 7

Conclusions

In this thesis, we dealt with the problem of efficient query evaluation in large-scale distributed systems such as peer-to-peer systems that are characterized by the complete lack of central control and administration and the autonomy and dynamic nature of the nodes participating in them. We presented new techniques designed to fit the unique requirements of peer-to-peer systems that improve the performance of query evaluation with regards to the communication and processing cost involved. We conclude, in this chapter, with a summary of our primary contributions (Section 7.1) and directions for future work (Section 7.2).

7.1 Summary of Contributions

In addressing the problem of efficient query evaluation in p2p systems, we mainly focused on two system components integral for data management: (i) the indexing mechanism used and (ii) the overlay network that the peers form. Both components have a significant impact on the efficiency of query evaluation and by designing new efficient indexing mechanisms and overlay organizations, we aimed at reducing both the communication and processing cost involved in query evaluation.

Due to the data heterogeneity that is another prominent feature in p2p systems and the widespread use of XML, we focused our study and proposed techniques for systems in which the participants maintain semi-structured data, i.e., XML documents. Thus, we focused on providing novel index structures appropriate for summarizing XML data designed for deployment in large-scale distributed systems.

To this end, we proposed two index structures, (i) the multi-level Bloom filters and (ii) the multi-level Bloom histograms. Both structures are based on the well-known Bloom

filter, which is a compact hash-based structure that supports membership queries and is often used in distributed systems due to its storage efficiency to reduce communication costs. We extended the Bloom filter so as to be suitable for summarizing XML data, that is, we designed the new multi-level Bloom filter and histogram so as to maintain the structural relationships between the elements in the XML document they summarize. The *multi-level Bloom filter* is designed to support boolean queries that check whether a path expression appears in the summarized document. We showed that with limited space overhead compared to the original data, the new structure incurs a very low false positive ratio. In addition, by combining the multi-level Bloom filter with histograms, we derived the *multi-level Bloom histogram* that offers selectivity estimations for path expressions. Similarly to the multi-level Bloom filters, the multi-level Bloom Histogram also incurs a very low false positive ratio and a low estimation error, while requiring only a small fraction of the space occupied by the original data.

We exploited our novel structures and used them as the building blocks for distributed indexes in p2p systems that are used to improve the performance of routing and query evaluation. In particular, we used the multi-level Bloom histogram to construct a *distributed clustered index* over the distributed XML documents maintained by the peers in the system, in which the documents are assigned to clusters according to their structural similarity. We studied two types of clustered indexes, (i) a distributed clustered index constructed based on a *K-Means* clustering algorithm and (ii) a *distributed hierarchical clustered index* constructed based on an incremental divisive clustering algorithm.

To illustrate the benefits of the clustered index with respect to improving the efficiency of query evaluation in p2p systems, we considered two popular query evaluation scenarios, (i) a *top-K query evaluation* and (ii) a *pay-as-you-go*, incremental query evaluation. To cope with heterogeneity and lack of global schema knowledge, we considered approximate query processing over XML data. In particular, we presented a set of algorithms that perform *structural query relaxation* and are applied on the clustered distributed index instead of the actual documents, thus, reducing significantly the cost of relaxation. Furthermore, we presented distributed query evaluation algorithms that exploit the clustered distributed index to reduce the communication and processing cost by enabling the system to reduce the number of peers it needs to consider when evaluating a query. Compared to a randomly partitioned distributed index, we showed through our experimental study, that the clustered distributed index manages to significantly increase the number of peers that do not need to be accessed.

We also used our multi-level Bloom filter to address another important problem in large-scale distributed systems, the problem of *database selection over distributed collections of XML documents*. We considered keyword queries instead of path expressions this time but used lowest common ancestor semantics to interpret the query results so as to maintain the structural properties of the XML data. We associated the relevance of a result to the keyword query with the height of the lowest common ancestor (LCA) node of all the keywords that appear in the query. To reduce the processing cost required, we

use an approximation algorithm for estimating the height of the LCA of a set of keywords by the greatest height of any pair of keywords that appear in the set. By attaining the information of the LCAs of any pair of keywords in a document in a pre-processing phase and maintaining this information, we can efficiently estimate the LCA height for any keyword query that matches a document. To achieve scalability, instead of processing each document separately, we exploit a variation of the multi-level Bloom filter and use that to summarize the information about the LCA nodes of the keywords pairs. We estimate the relevance of each document based on its multi-level Bloom based summary and aggregate over all the documents in a collection. Thus, we are able to provide a ranking of all the collections according to their relevance (usefulness) to a query, which our experimental results show that it is very close to the ranking we would derive if we processed each document separately, but with a much lower processing and storage cost.

With respect to the overlay networks, motivated by real life paradigms in Internet-based applications like social networks and file-sharing systems that show that their users tend to form implicit groups (clusters) based on common interests and similar content, we focused our study on *clustered overlay networks* and models and protocols for their formation and evolution.

We adopted a game-theoretic view and modelled the problem of cluster formation and evolution as a *strategic game* in which the players are the peers that determine their strategy by selecting which clusters to join. The goal of the game is for each peer to select the strategy (set of clusters to join) that minimize a utility function that depends on *query recall* and the cost involved in belonging to a cluster. In accordance to real users in p2p systems that exhibit both selfishness and altruism, we model both *selfish* and *altruistic* behavior and define an appropriate utility function for each. Selfish peers aim at maximizing the recall of their own queries, while altruistic peers aim at maximizing the recall of the queries of the peers that belong in their clusters. Based on the individual utility functions for the peers, we also defined criteria to measure the total system performance with respect to the recall achieved by its peers and the cost required for the maintenance of the clusters.

Besides the cluster overlay formation, we showed that our game is also appropriate for modelling cluster evolution so as to cope with the dynamic nature of the peers. We proposed a fully distributed, uncoordinated *cluster reformulation protocol* that based on local decisions that are made independently by each peer, manages to adapt the system to changing conditions and improve the overall query recall in the system. Through an extensive experimental evaluation, we showed that the uncoordinated protocol we propose, performs as well as a corresponding coordinated one would in terms of both identifying the implicit clusters that appear in a system and also adapting to changing conditions but requires much lower communication overhead and respects the autonomy of peers.

In a nutshell, our main results can be outlined as:

- We showed the importance (and benefits in terms of efficiency) of using appropriate indexing techniques and corresponding query evaluation algorithms that are applied

on the indexes and not require access to the actual data when dealing with large-scale distributed systems.

- We advocated and verified the usefulness of clustering in p2p systems as a means to improve the efficiency of query evaluation, both with regards to clustering data or indexes and clustering peers in the overlay network.
- We offered a novel view of the problem of cluster formation and evolution in p2p systems based on a game theoretic model, that offers important insights of the actual behavior of peers in real p2p systems and the impact on their performance and also emphasizes the importance of considering both content and query workload when dealing with clustering problems.
- We showed with our fully decentralized protocol for cluster formation that uncoordinated local decisions guided by selfish criteria, i.e., to improve the personal gain of a single peer, can be valuable to the greater good in the system, that is, lead to improvement for the entire p2p system.

7.2 Future Work

We now offer some directions for future research on issues that were derived from this thesis but not fully explored yet. We distinguish between short term plans that consist mainly of extensions to our proposed techniques, and long term plans that outline more general ideas for future research related to our work.

Short Term Plans

When dealing with the approximate interpretation of the queries over XML data, we have focused on structural query relaxation and based the construction of our clustered index on the structural similarity of the documents. However, generally, approximate query processing and ranking for XML may include other characteristics of the documents such as values or keywords. This is an important and well studied problem where various techniques including ontologies and thesaurus are used for relaxation and tf*idf measures or other IR techniques for ranking (for example, [88, 128, 45]). An issue that we are interested in investigating in the future is how to combine the structural and other characteristics in our approach. One way to achieve this is to consider them as complementary and treat them independently. In this case, we can maintain our structural clustering and use auxiliary indexes for values. For ranking, an aggregated distance measure based on both structure and value can be used. Structural clustering would still improve pruning, but perhaps in a lesser degree than with pure structural relaxation. Another approach would be to construct the clustered index based on both values and structure using appropriate index structures and clustering methods. Deriving a compact, summary data structure similar to the Depth Bloom Histogram for this case is an interesting problem

for future research. This is a challenging task, since such an index should be compact, merge-able, update-able and provide good selectivity estimations with no false negatives. Clearly, which of the two approach works best depends on the data distribution. If many query results comply to a specific DTD or XML schema, then structural clustering may be appropriate. On the other hand, if documents relative to a specific query comply to numerous different schemes, then a combined approach might be more appropriate.

Regarding the clustered index, we provided an incremental divisive clustering algorithm for the construction of a hierarchical clustered index. Though the construction supports updates and incremental insertions, it is based on the assumption that the general distribution that the indexed documents follow does not drastically change over time. The problem in such a case would be that though the leaf nodes of the hierarchical index may be split or merged to adjust to the updated conditions, the internal nodes of the hierarchy that maintain the routing information remain fixed. Note the information they include is updated, but the nodes remain static. Thus, in cases where the documents distribution changes drastically it may be preferable to apply the clustering algorithm from scratch to build a hierarchy that is better suited to the new content distribution. This re-clustering procedure however would entail large communication costs. We plan to study this problem and design techniques for the dynamic re-clustering of the documents that would not require building the index from scratch.

As far as the cluster formation and evolution is concerned, though our game-theoretic model covers the case in which a peer belongs to more than one cluster, we focused our reformulation protocol and its experimental evaluation in the case of single cluster memberships. We plan to study the problem of belonging to multiple clusters more thoroughly and identify scenarios in which the system reaches stability and optimality in that case. Furthermore, we want to confirm that our protocol performance remains satisfying for multiple cluster memberships or if additional mechanisms for reducing the communication costs are required.

Long Term Plans

In our game model, we used utility functions based on query recall and cluster membership costs to guide the cluster formation game. Recall-based and membership cost criteria were also defined to determine the overall performance-quality of the system. That is, we considered a configuration as a good clustering scheme if it achieved a high recall over the total query workload in the system. However, other criteria such as diversity can be used to determine the quality of the clustering scheme. It would be interesting to explore whether such criteria can also be used for the cluster formation and evolution problem in peer-to-peer systems and how they would be adopted in such cases. It is not straightforward that the good recall-wise clustering schemes our protocol forms would also be considered good under these new criteria. Thus, new models and protocols may be also required in such cases. In addition, we would like to explore the relation of our game model with traditional optimization goals set in clustering. In this respect, we would be

interested in utility functions that by optimizing them we minimize the distance of the peers within a cluster and maximize the distance of the peers that belong to different clusters.

Finally, we want to examine how our game-theoretic model for cluster formation can be applied to other areas such as social networks. A problem similar to cluster formation in such a context is the problem of friends selection. Game theoretic approaches have been applied in p2p systems for link creation which is also a closely related problem. We plan to investigate how our model can be adapted for that problem and how the transitive property, i.e., that the friends of my friend may also be my friends, which is one of the basic ideas motivating the use of clustering in p2p systems, can also be exploited by our protocol to lead to a more efficient process for discovering new friends in social networks.

Bibliography

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva and R. Schmidt, P-Grid: a self-organizing structured P2P system, *Sigmod Rec.* **32(3)** (2003) 29–33.

[2] S. Abiteboul, P. Buneman and D. Suciu, *Data on the Web: From relations to semistructured data and XML*, Morgan Kauffmann Publishers (2000).

[3] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu and T. Milo, Dynamic XML documents with distribution and replication, *Proceedings of the **2003** ACM SIGMOD International Conference on Management of Data* (2003) 527–538.

[4] S. Abiteboul, I. Manolescu and N. Preda, Constructing and querying peer-to-peer warehouses of XML resources, *2nd International Workshop on Semantic Web and Databases* (2004).

[5] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda and C. Sun, XML processing in DHT networks, *Proceedings of the **24**th International Conference on Data Engineering* (2008) 606–615.

[6] A. Aboulnaga, A.R. Alameldeen and J.F. Naughton, Estimating the selectivity of XML path expressions for Internet scale applications, *Proceedings of the **27**th International Conference on Very Large Databases* (2001) 591–600.

[7] E. Adar and B.A. Huberman, Free riding on Gnutella, *First Monday* **5(10)** (2000).

[8] S. Amer-Yahia, L. Lakshmanan and S. Pandit, Flexpath: Flexible structure and full-text querying for xml, *Proceedings of the **2004** ACM SIGMOD International Conference on Management of Data* (2004) 83–94.

[9] S. Amer-Yahia, S. Cho and D. Srivastava, Tree pattern relaxation, *Proceedings of the **8**th International Conference on Extending Database Technology* (2002) 496–513.

[10] F.S. Annexstein, K.A. Berman and M.A. Jovanovic, Latency effects on reachability in large-scale peer-to-peer networks, *Proceedings of the **13**th Annual ACM Symposium on Parallel Algorithms and Architectures* (2001) 84–92.

[11] A. Arion, A. Bonifati, I. Manolescu and A. Pugliese, Path Summaries and Path Partitioning in Modern XML Databases, WorldWide Web **11** (2008) 117–151.

[12] N. Bansal, F. Chiang, N. Koudas and F.W. Tompa, Seeking stable clusters in the blogosphere, *Proceedings of the **33**rd International Conference on Very Large Data Bases* (2007) 806–817.

[13] M. Bawa, G.S. Manku and P. Raghavan, SETS: Search enhanced by topic segmentation, *Proceedings of the **26**th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2003) 306–313.

[14] M. Bender, S. Michel, G. Weikum and C. Zimmer, The MINERVA project: Database selection in the context of P2P search, *Datenbanksysteme in Business, Technologie und Web (BTW) : 11. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS)***(65)** (2005) 125–144.

[15] B. Bloom, Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM* **13(7)** (1970) 422–426.

[16] A. Bonifati, V.P. Bucci and A. Cuzzocrea, XPath lookup queries in P2P networks, *Proceedings of the **6**th Annual ACM International Workshop on Web Information and Data Management* (2004) 48–55.

[17] A. Bonifati, P.K. Chrysanthis, A.M. Ouksel and KU. Sattler, Distributed databases and peer-to-peer databases:Past and present, *Sigmod Record*, **37(1)** (2008) 5–11.

[18] J.M. Bremer and M. Gertz, On distributing XML repositories, *International Workshop on Web and Databases* (2003) 73–78.

[19] P. Buneman , G. Cong, W. Fan and A. Kementsietsidis, Using partial evaluation in distributed query evaluation, *Proceedings of the **32**nd International Conference on Very Large Data Bases* (2006) 211–222.

[20] M. Cai and M. Frank, RDFPeers: A scalable distributed repository based on a structured peer-to-peer network, *Proceedings of the **13**th International Conference on World Wide Web* (2004) 650–657.

[21] J.P. Callan, Z. Lu and W.B. Croft, Searching distributed collections with inference networks, *Proceedings of the **18**th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (1995) 21–28.

[22] P. Cao and Z. Wang, Efficient top-K query calculation in distributed networks, *Proceedings of the **23**rd Annual ACM Symposium on Principles of Distributed Computing* (2004) 206–215.

[23] D. Chamberlin, XQuery: An XML query language, *IBM System Journal*, **41(4)** (2003) 597–615.

[24] S. Chernov, P. Serdyukov, M. Bender, S. Michel, G. Weikum and C. Zimmer, Database selection and result merging in p2p web search, *Databases, Information Systems, and Peer-to-Peer Computing,International Workshops* (2005/2006) 26–37.

[25] CW. Chung, JK Min and K. Shim, APEX: An adaptive path index for XML data, *Proceedings of the **2002** ACM SIGMOD International Conference on Management of Data* (2002) 121–132.

[26] E. Cohen and S. Shenker, Replication strategies in unstructured peer-to-peer networks, *SIGCOMM Comput. Commun. Rev.* **32(4)** (2002) 177–190.

[27] E. Cohen, A. Fiat and H. Kaplan, Associative search in peer to peer networks: Harnessing latent semantics, *Comp. Networks* **51(8)** (2007) 1861–1881.

[28] S. Cohen, J. Mamou, Y. Kanza and Y. Sagiv, XSEarch: A semantic search engine for XML, *Proceedings of the **29**th International Conference on Very Large Data Bases* (2003) 45–56.

[29] G. Cong, W. Fan and A. Kementsietsidis, Distributed Query Evaluation with Performance Guarantees, *Proceedings of the **2007** International SIGMOD Conference* (2007).

[30] B.F. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason and M. Shadmon, Fast index for semistructured data, *Proceedings of the **27**th International Conference on Very Large Data Bases* (2001) 341–350.

[31] A. Crespo and H. Garcia-Molina, Routing indices for peer-to-peer systems. *Proceedings of the **22**nd International Conference on Distributed Computing Systems* (2002) 23-.

[32] A. Crespo and H. Garcia-Molina, Semantic overlay networks for P2P systems, *Technical Report 2003-75*, Stanford InfoLab, 2003.

[33] S.E. Czerwinski, B.Y. Zhao, T.D. Hodes, A.D. Joseph and R.H. Katz, An architecture for a secure service discovery service, *Proceedings of the **5**th Annual ACM/IEEE International Conference on Mobile Computing and Networking* (1999) 24–35.

[34] N. Daswani, H. Garcia-Molina and B. Yang, Open problems in data-sharing peer-to-peer systems, *Proceedings of the **9**th International Conference on Database Theory* (2003) 1–15.

[35] S. DeRose, E. Maler, D. Orchard and B. Trafford, XML linking language (Xlink) version 1.0, *http://www.w3.org/TR/xlink* (2000).

[36] C. Doulkeridis, K. Norvag and M. Vazirgiannis, Desent: Decentralized and distributed semantic overlay generation in p2p networks, *IEEE Journal on Selected Areas in Communications* **25(1)** (2007) 25–34.

[37] A. Fabrikant, A. Luthra, E. Maneva, C.H. Papadimitriou and S. Shenker, On a network creation game, *Proceedings of the **22**nd Annual Symposium on Principles of Distributed Computing* (2003) 347–351.

[38] Facebook, *www.facebook.com*.

[39] R. Fagin, A. Lotem and M. Naor, Optimal aggregation algorithms for middleware, *Proceedings of the **20**th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (2001) 102–113.

[40] L. Fan, P. Cao, J. Almeida and A. Broder, Summary cache: A scalable wide-area Web cache sharing protocol, *IEEE/ACM Transactions Networks* **8(3)** (2000) 281–293.

[41] A. Fast, D. Jensen, and B.N. Levine, Creating social networks to improve peer-to-peer networking, *Proceedings of the **11**th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (2005) 568–573.

[42] C. Fellbaum, *WordNet: An electronic lexical database (Language, Speech an Communication)*, MIT Press, 1998.

[43] Flickr. *www.flickr.com*

[44] J. Freire, J.R. Haritsa, M. Ramanath, P. Roy and J. Siméon, Statix: Making xml count, *Proceedings of the **2002** ACM SIGMOD International Conference on Management of Data* (2002) 181–191.

[45] N. Fuhr and K. Grossjohann, XIRQL: An extension of XQL for information retrieval, *Proceedings of the SIGIR **2000** Workshop on XML and Information Retrieval* (2000).

[46] N. Fuhr and N. Govert, Index compression vs. retrieval time of inverted files for XML documents, *Proceedings of the **11**th ACM CIKM Conference*, (2002) 662–664.

[47] L. Galanis, YW. Shawn, R. Jeffery and D.J. DeWitt, Processing queries in a large peer-to-peer system, *Proceedings of the **15**th International Conference on Advanced Information Systems Engineering* (2003) 273–288.

[48] L. Galanis, Y. Wang, S. Jeffery and D.J. DeWitt, Locating data sources in large distributed systems, *Proceedings of the **29**th International Conference on Very Large Data Bases* (2003) 874–885.

[49] P. Garbacki, D.H.J. Epema and M. Van Steen, Optimizing peer relationships in a super-peer network, *Proceedings of the **27**th International Conference on Distributed Computing Systems* (2007) 31.

[50] M. Gertz and JM. Bremer, Distributed XML Repositories: Top-down Design and Transparent Query Processing, *Technical Report TR CSE-2003-20*, Department of Computer Science, University of California at Davis (2003).

[51] Knowbuddy's Gnutella FAQ, *http://www.rixsoft.com/Knowbuddy/gnutellafaq.html*.

[52] R. Goldman and J. Widom, DataGuides: Enabling query formulation and optimization in semistructured databases, *Proceedings of the **23**rd International Conference on Very Large Data Bases* (1997) 436–445.

[53] L. Gravano, H. Garcia-Molina and A. Tomasic, Gloss: text-source discovery over the Internet, *ACM Trans. on Database Systems* **24(2)** (1999) 229–264.

[54] S.D. Gribble, E.A. Brewer, J.M. Hellerstein and D. Culler, Scalable distributed data structures for Internet service construction, *OSDI'00: Proceedings of the **4**th Conference on Symposium on Operating System Design & Implementation* (2000) 319–332.

[55] S. Gribble, A. Halevy, Z. Ives, M. Rodrig and D. Suciu, What can databases do for P2P?, *Proceedings of the **4**th International Workshop on the Web and Databases, WebDB 2001* (2001) 31–36 (Informal Proceedings).

[56] L. Guo, F. Shao, C. Botev and J. Shanmugasundaram, XRank: Ranked keyword search over XML documents, *Proceedings of the **2003** ACM SIGMOD International Conference on Management of Data* (2003) 16–27.

[57] H. Gylfason, O. Khan and G. Shoenebeck, Chora: Expert-based P2P Web Search, ***5**th International Workshop on Agents and Peer-to-Peer Computing, AP2PC 2006* 74–85 (2008).

[58] S.B. Handurukande, AM. Kermarrec, F.L. Fessant, L. Massouli and S. Patarin, Peer sharing behaviour in the eDonkey network, and implications for the design of serverless file sharing systems, *SIGOPS Oper. Syst. Rev.* **40(4)** (2006) 359–371.

[59] T.D. Hodes, S.E. Czerwinski, B.Y. Zhao, A.D. Joseph and R.H. Katz, An architecture for secure wide-area service discovery, *Proceedings of the **5**th Annual ACM/IEEE International Conference on Mobile Computing and Networking* (1999) 24.

[60] V. Hristidis, Y. Papakonstantinou and A. Balmin, Keyword proximity search on XML graphs, *Proceedings of the **19**th International Conference on Data Engineering* (2003) 367–378.

[61] V. Kantere, D. Tsoumakos, T. Sellis and N. Roussopoulos, GrouPeer:Dynamic clustering of P2P databases, *Information Systems*, **(34)** (2009) 62–86.

[62] Kazaa. *www.kazaa.com*.

[63] M.S. Khambatti, K. Ryu and P. Dasgupta, Efficient discovery of implicitly formed peer-to-peer communities, *International Journal of Parallel and Distributed Systems and Networks* **5(4)** (2002) 155–164.

[64] M.S. Khambatti, K. Ryu and P. Dasgupta, Peer-to-peer communities: Formation and discovery, *Proceedings of the **14**th IASTED International Conference on Parallel and Distributed Computing Systems* (2002) 497–504.

[65] M. Klein, Interpreting XML via an RDF schema, *Chapter in Knowledge Annotation for the Semantic Web*, IOS Press, Amsterdam, 2003.

[66] G. Kokkinidis and V. Christophides, Semantic query routing and processing in P2P database systems: ICS-FORTH SQPeer middleware, *Current Trends in Database Technology - EDBT 2004 Workshops, PhD, DataX, PIM, P2P&DB, and ClustWeb* (2004) 486–495.

[67] G. Koloniari and E. Pitoura, Bloom-based Filters for Hierarchical Data, ***5**th International Workshop on Distributed Data and Structures (WDAS)* (2003).

[68] G. Koloniari, Y. Petrakis and E. Pitoura, Content-Based Overlay Networks of XML Peers Based on Multi-Level Bloom Filters, ***1**st International Workshop on Databases, Information Systems, and Peer-to-Peer Computing* (2003) 232–247.

[69] G. Koloniari and E. Pitoura, Content-Based Routing of Path Queries in Peer-to-Peer Systems,*Advances in Database Technology - EDBT 2004, **9**th International Conference on Extending Database Technology* (2004) 29–47.

[70] G. Koloniari and E. Pitoura, Filters for XML-based Service Discovery in Pervasive Computing,*Computer Journal, Special Issue on Mobile and Pervasive Computing* **47(4)** (2004) 461–474.

[71] G. Koloniari and E. Pitoura, Peer-to-Peer Management of XML Data: Issues and Research Challenges, *Sigmod Record* **34(2)** (2005) 6–17.

[72] G. Koloniari and E. Pitoura, Workload-Aware Clustering of XML Peers, *International Conference on Intelligent Systems And Computing: Theory And Applications* (2006).

[73] G. Koloniari and E. Pitoura, A Clustered Index Approach to Distributed XPath Processing, *Proceedings of the **24**th International Conference on Data Engineering* (2008) 1516–1518.

[74] G. Koloniari and E. Pitoura, Recall-based Cluster Reformulation for Selfish Peers, *Proceedings of the **24**th International Conference on Data Engineering Workshops (NetDB ICDE Workshop)* (2008) 200–205.

[75] G. Koloniari and E. Pitoura, Structural Relaxation of XPath Queries, *Proceedings of the **25**th International Conference on Data Engineering* (2009) 529–540.

[76] G. Koloniari and E. Pitoura, A Recall-Based Cluster Formation Game in Peer-to-Peer Systems, To appear in the *35th International Conference on Very Large Data Bases* (2009).

[77] G. Koloniari and E. Pitoura, Structural Relaxation Over Distributed XML Collections, Submitted for publication.

[78] G. Koloniari and E. Pitoura, LCA-based Selection for Distributed XML Document Collections, *Technical Report* TR 04-2009, Computer Science Department, University of Ioannina (2009).

[79] N. Koudas, M. Rabinovich, D. Srivastava and T. Yu, Routing XML queries. *Proceedings of the 20th International Conference on Data Engineering* (2004) 844.

[80] N. Laoutaris, G. Smaragdakis, A. Bestavros and J.W. Byers, Implications of selfish neighbor selection in overlay networks, *Proceedings of the 26th IEEE International Conference on Computer Communications* (2007) 490–498.

[81] M.L. Lee, L. Yang, W. Hsu and X. Yang, XClust: Clustering XML schemas for Effective Integration, *Proceedings of the 11th International Conference on Information and Knowledge Management* (2002) 292–299.

[82] Y. Li, C. Yu and H.V. Jagadish, Schema-free xquery. *Proceedings of the 30th International Conference on Very Large Data Bases* (2004) 72–83.

[83] W. Lian, N. Mamoulis and D.W. Cheung, A Filter Index for Complex Queries on Semi-structured Data, *Proceedings of the 4th International Conference on Web-Age Information Management*, (2003) 397–408.

[84] W. Lian, D. Cheung, N. Mamoulis and S. Yiu, An efficient and scalable algorithm for clustering XML documents by structure, *IEEE Trans. on Knowl. and Data Eng.*, **16(1)** (2004) 82–96.

[85] A. Loser, F. Naumann, W. Siberski, W. Nejdl and U. Thaden, Semantic overlay clusters within super-peer networks, *1st International Workshop, DBISP2P, on Databases Information Systems and Peer-to-Peer Computing* (2003) 33–47.

[86] A. Loser, W. Siberski, M. Wolpers and W. Nejdl, Information integration in schema-based peer-to-peer networks, *Proceedings of the 15th International Conference of Advanced Information Systems Engineering* (2003).

[87] Q. Lv, P. Cao, E. Cohen, K. Li and S. Shenker, Search and replication in unstructured peer-to-peer networks, *Proceedings of the 16th International Conference on Supercomputing* (2002) 84–95.

[88] A. Marian, S. Amer-Yahia, N. Koudas and D. Srivastava, Adaptive processing of top-k queries in XML, *Proceedings of the **21**st International Conference on Data Engineering* (2005) 162–173.

[89] The MD5 message-digest algorithm, *RFC1321*.

[90] S. Michel, M. Bender, P. Triantafillou and G. Weikum, IQN Routing: Integrating Quality and Novelty for P2P Web Search, *Proceedings of the EDBT* (2006).

[91] S. Michel, P. Triantafillou and G. Weikum, KLEE: A framework for distributed top-k query algorithms, *Proceedings of the **31**st International Conference on Very Large Data Bases* (2005) 637–648.

[92] G. Miller, WordNet: A Lexical Database for English, *CACM* **38(11)** (1995).

[93] D.S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins and Z. Xu, Peer-to-peer computing, *Technical Report HPL-2002-57*, HP Laboratories Palo Alto, 2002.

[94] A. Mislove, M. Marcon, K.P. Gummadi, P. Druschel and B. Bhattacharjee, Measurement and analysis of online social networks, *Proceedings of the **7**th ACM SIGCOMM Conference on Internet Measurement* (2007) 29–42.

[95] A. Mohan and V. Kalogeraki, Speculative routing and update propagation: A Kundali centric approach, *IEEE International Conference on Communications* (2003) 343 - 347.

[96] R. Morris, I. Stoica, D. Karger, M.F. Kaashoek and H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for Internet applications, *IEEE/ACM Trans. on Networking* **11(1)** (2003) 17–32.

[97] T. Moscibroda, S. Schmid and R. Wattenhofer, On the topologies formed by selfish peers, *Proceedings of the **25**th Annual ACM Symposium on Principles of Distributed Computing* (2006) 133–142.

[98] Napster. *http://www.napster.com/*

[99] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst and A. Loser, Super-peer-based routing and clustering strategies for RDF-based peer-to-peer Networks, *Proceedings of the **12**th International Conference on World Wide Web* (2003) 536–543.

[100] The Niagara Project, *http://www.cs.wisc.edu/niagara/data*.

[101] The Niagara Generator, *http://www.cs.wisc.edu/niagara*.

[102] V. Papadimos, D. Maier and K. Tufte, Distributed query processing and catalogs for peer-to-peer systems, ***1****st Biennial Conference on Innovative Data Systems Research* (2003) Online Proceedings.

[103] S. Park and HJ. Kim, A new query processing technique for XML based on signature, *Proceedings of the* ***7****th International Conference on Database Systems for Advanced Applications* (2001) 22-.

[104] N. Polyzotis and M. Garofalakis, Structure and value synopses for XML data graphs, *Proceedings of the* ***28th*** *International Conference on Very Large Data Bases* (2002) 466–477.

[105] N. Polyzotis, M. Garofalakis and Y. Ioannidis, Approximate XML query answers, *Proceedings of the* ***2004*** *ACM SIGMOD International Conference on Management of Data* (2004) 263–274.

[106] N. Polyzotis, M. Garofalakis and Y. Ioannidis, Selectivity estimation for xml twigs. *Proceedings of the* ***20****th International Conference on Data Engineering* (2004) 264.

[107] M. Rabin, Fingerprinting by random polynomials, *Technical report, CRCT TR-15-81*, Harvard University, 1981.

[108] M.V. Ramakrishna, Practical performance of Bloom Filters and parallel free-text searching, *Communications of the ACM* **32(10)** (1989) 1237–1239.

[109] P.R. Rao and B. Moon, Locating XML Documents in a Peer-to-Peer Network using Distributed Hash Tables, To appear in *IEEE Transactions on Knowledge and Data Engineering* (2009).

[110] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Schenker, A scalable content-addressable network, *Proceedings of the* ***2001*** *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2001) 161–172.

[111] S.C. Rhea and J. Kubiatowicz, Probabilistic location and routing, *Proceedings of the* ***21****st Annual Joint Conference of the IEEE Computer and Communications Societies* (2002).

[112] J. Risson and T. Moors, Survey of research towards robust peer-to-peer networks: search methods, *Technical report UNSW-EE-P2P-1-1*, University of New South Wales, 2004.

[113] M. Roussopoulos and M. Baker, CUP: Controlled update propagation in peer-to-peer networks, *Proceedings of the* ***2003*** *USENIX Annual Technical Conference* (2003) 167–180.

[114] N. Sample, B. Cooper , M.J. Franklin, G.R. Hjaltason, M. Shadmon and L. Cohen, The Managing complex and varied data with the IndexFabricTM, *Proceedings of the **18**th International Conference on Data Engineering* (2002) 492–493.

[115] C. Sartiani, P. Manghi, G. Ghelli and G. Conforti. XPeer: A self-organizing XML P2P database system, *Current Trends in Database Technology - EDBT 2004 Workshops (P2P and DB)* (2004) 456–465.

[116] M. Sayyadian, H. LeKhac, A. Doan and L. Gravano, Efficient keyword search across heterogeneous relational databases, *Proceedings of the **23**rd International Conference on Data Engineering* (2007) 346–355.

[117] M. Schlosser, M. Sintek, S. Decker and W. Nejdl, A scalable and ontology-based P2P infrastructure for semantic web services, *Proceedings of the **2**nd International Conference on Peer-to-Peer Computing* (2002) 104.

[118] C. Schmidt and M. Parashar, Flexible information discovery in decentralized distributed systems, *Proceedings of the **12**th IEEE International Symposium on High Performance Distributed Computing* (2003) 266.

[119] L. Sidirourgos, G. Kokkinidis, T. Dalamagas, V. Christophides and T.K. Sellis, Indexing views to route queries in a PDMS, *Distributed and Parallel Databases* **23(1)** (2008), 45–68.

[120] K. Sripanidkulchai, B. Maggs and H. Zhang, Efficient content location using interest-based locality in peer-to-peer systems, *Proceedings IEEE INFOCOM 2003, The **22**nd Annual Joint Conference of the IEEE Computer and Communications Societies* (2003).

[121] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek and H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for Internet applications, *Proceedings of the **2001** Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2001) 149–160.

[122] H. Stuckenschmidt, R. Vdovjak, G. Houben and J. Broekstra, Index structures and algorithms for querying distributed RDF repositories, *Proceedings of the **13**th International Conference on World Wide Web* (2004) 631–639.

[123] D. Suciu, Distributed query evaluation on semistructured data, *ACM Trans. Database Syst.* **27(1)** (2002) 1–62.

[124] K. Tajima and Y. Fukui, Answering XPath queries over networks by sending minimal views, *Proceedings of the **30**th International Conference on Very Large Data Bases* (2004) 48–59.

[125] C. Tang, Z. Xu and S. Dwarkadas, Peer-to-peer information retrieval using self-organizing semantic overlay networks, *Proceedings of the **2003** Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2003) 175–186.

[126] C. Tang, Z. Xu and M. Mahalingam, psearch: information retrieval in structured overlays, *Computer Communication Review* **33(1)** (2002) 89–94.

[127] I. Tatarinov and A. Halevy, Efficient query reformulation in peer data management systems, *Proceedings of the **2004** ACM SIGMOD International Conference on Management of Data* (2004) 539–550.

[128] M. Theobald, R. Schenkel and G. Weikum, An efficient versatile query engine for TopX search, *Proceedings of the **31**st International Conference on Very Large Data Bases* (2005) 625–636.

[129] M. Theobald, R. Schenkel and G. Weikum, Exploiting structure, annotation, and ontological knowledge for automatic classification of XML data, *International Workshop on Web and Databases* (2003) 1–6.

[130] P. Triantafillou and T. Pitoura, Towards a unifying framework for complex query processing over structured peer-to-peer data networks, ***1**st International Workshop, DBISP2P Databases, Information Systems, and Peer-to-Peer Computing* (2003) 169–183.

[131] P. Triantafillou, C. Xiruhaki, M. Koubarakis and N. Ntarmos, Towards high performance peer-to-peer content and resource sharing systems, ***1**st Biennial Conference on Innovative Data Systems Research* (2003) Online Proceedings.

[132] D.K. Vassilakis and V. Vassalos, Modelling real p2p networks: The effect of altruism, *Proceedings of the **7**th IEEE International Conference on Peer-to-Peer Computing* (2007) 19–26.

[133] Q.H. Vu, B.C. Ooi, D. Papadias and A.K.H. Tung, A graph method for keyword-based selection of the top-k databases, *Proceedings of the **2008** ACM SIGMOD International Conference on Management of Data* (2008) 915–926.

[134] Q. Wang and M. Oszu, A data locating mechanism for distributed XML data over P2P networks, *Technical Report, CS-2004-45*, University of Waterloo, School of Computer Science, Waterloo, Canada, 2004.

[135] W. Wang, H. Jiang, H. Lu and J. Yu, Bloom Histogram: Path selectivity estimation for XML data with updates, *Proceedings of the **30**th International Conference on Very Large Data Bases* (2004) 240–251.

[136] World-Wide Web Consortium: Resource Description Framework. *http://www.w3.org/RDF*.

[137] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second edition), *http://www.w3.org/TR/REC-xml*.

[138] L. Xiong and L. Lu, PeerTrust: Supporting Reputation-Based Trust in Peer-to-Peer Communities, *IEEE Transactions on Data and Knowledge Engineering, Special Issue on Peer-to-Peer Based Data Management* **16(7)** (2004) 843–857.

[139] Y. Xu and Y. Papakonstantinou, Efficient keyword search for smallest lcas in xml databases, *Proceedings of the **2005** ACM SIGMOD International Conference on Management of Data* (2005) 527–538.

[140] Y. Xu and Y. Papakonstantinou, Efficient lca based keyword search in xml data, *Proceedings of the **11**th International Conference on Extending Database Technology* (2008) 535–546.

[141] B. Yu, G. Li, K. Sollins, and A.K.H. Tung, Effective keyword-based selection of relational databases, *Proceedings of the **2007** ACM SIGMOD International Conference on Management of Data* (2007) 139–150.

[142] M.J. Zaki and C. Aggarwal, XRules: An effective structural classifier for XML data, *Proceedings of the **9**th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2003) 316–325.

Published Work

1. G. Koloniari and E. Pitoura, Bloom-based Filters for Hierarchical Data, ***5****th International Workshop on Distributed Data and Structures (WDAS)* (2003).

2. G. Koloniari, Y Petrakis and E. Pitoura, Content-Based Overlay Networks of XML Peers Based on Multi-Level Bloom Filters, ***1****st International Workshop on Databases, Information Systems, and Peer-to-Peer Computing* (2003) 232–247.

3. G. Koloniari and E. Pitoura, Content-Based Routing of Path Queries in Peer-to-Peer Systems,*Advances in Database Technology - EDBT 2004,* ***9****th International Conference on Extending Database Technology* (2004) 29–47.

4. G. Koloniari and E. Pitoura, Filters for XML-based Service Discovery in Pervasive Computing,*Computer Journal, Special Issue on Mobile and Pervasive Computing* **47(4)** (2004) 461–474.

5. Y. Petrakis, G. Koloniari and E. Pitoura, On Using Histograms as Routing Indexes in Peer-to-Peer System, ***2****nd International VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing*, (2004).

6. G. Kolonari, Mobile Peer-to-Peer, *Mobile Information Management 2004.*

7. J. Grünbauer, M. Klein, G. Koloniari, G. Samaras and C. Türker, 04441 Working Group - Description and Matching of Services in Mobile Environments, *Mobile Information Management 2004.*

8. G. Koloniari and E. Pitoura, Peer-to-Peer Management of XML Data: Issues and Research Challenges, *Sigmod Record* **34(2)** (2005) 6–17.

9. G. Koloniari, Y. Petrakis, E. Pitoura and T. Tsotsos, Query Workload-Aware Overlay Construction Using Histograms, *Proceedings of the* ***2005*** *ACM CIKM International Conference on Information and Knowledge Management* (2005) 640–647.

10. G. Koloniari and E. Pitoura, Workload-Aware Clustering of XML Peers, *International Conference on Intelligent Systems And Computing: Theory And Applications* (2006).

11. P. Skyvalidas, E. Pitoura, V. Dimakopoulos and G. Koloniari, Replication Routing Indexes for XML, *International Workshop on Databases, Information Systems, and Peer-to-Peer Computing* (2007).

12. G. Koloniari and E. Pitoura, A Clustered Index Approach to Distributed XPath Processing, *Proceedings of the **24**th International Conference on Data Engineering* (2008) 1516–1518.

13. G. Koloniari and E. Pitoura, Recall-based Cluster Reformulation for Selfish Peers, *Proceedings of the **24**th International Conference on Data Engineering Workshops (NetDB ICDE Workshop)* (2008) 200–205.

14. G. Koloniari and E. Pitoura, Structural Relaxation of XPath Queries, *Proceedings of the **25**th International Conference on Data Engineering* (2009) 529–540.

15. G. Koloniari and E. Pitoura, A Recall-Based Cluster Formation Game in Peer-to-Peer Systems, To appear in the *35th International Conference on Very Large Data Bases* (2009).

16. G. Koloniari and E. Pitoura, LCA-based Selection for Distributed XML Document Collections, *Technical Report TR 04-2009*, Computer Science Department, University of Ioannina (2009).

17. G. Koloniari and E. Pitoura, Structural Relaxation Over Distributed XML Collections, Submitted for publication (2009).

Short CV

Georgia Koloniari was born on June 5, 1978 in Thessaloniki, Greece. She received her BSc degree in Computer Science from the Department of Computer Science at the University of Ioannina, Greece in 2001 and her MSc also in Computer Science with specialization in Information Systems from the same department in 2003. Her research interests include data management in peer-to-peer systems and distributed management of XML data.