Efficient Continuous Multi-Query Processing over Data Streams

Efficient Continuous Multi-Query Processing over Data Streams

Lefteris Zervakis

Ph.D. Thesis, Department of Informatics and Telecommunications

University of the Peloponnese, July 2019

Efficient Continuous Multi-Query Processing over Data Streams

Lefteris Zervakis

A thesis submitted in partial fulfillment
for the degree of

Doctor of Philosophy

Department of Informatics and Telecommunications

Doctoral Comittee

Associate Professor Christos Tryfonopoulos, Thesis Supervisor
Professor Spiros Skiadopoulos, Member
Professor Costas Vassilakis, Member

University of the Peloponnese

2019

Acknowledgements

This thesis marks the end of my endeavours as a PhD candidate. I would have not been able to complete such undertaking without the belief and undivided support of my supervisor Christos Tryfonopoulos. I would like to thank him for his support, enthusiasm, and perseverance throughout the years.

Besides my advisor, I would also like to thank the members of my doctoral committee, Spiros Skiadopoulos, Costas Vassilakis, Manolis Koubarakis, Theodore Dalamagas, George Lepouras and Vinay Setty for their comments and suggestions on improvements and extensions of this work.

My sincere thanks also go to Paraskevi Raftopoulou for her support during hard times and her never-ending encouragement to push forward. Paraskevi has provided me with valuable comments and interesting insights for all my research works and more importantly for this thesis.

Special thanks go to my family and friends for their love and encouragement throughout all my life endeavours. Their courage in life and their strength in difficulties was an extra motivation for this effort.

During my PhD I had the opportunity to work as an intern in a great research group at Max-Planck-Institut für Informatik (MPII). A special thanks goes to Prof. Gerhard Weikum for providing me with this notable experience. Additionally, I worked as a research assistant at the Center for Data-Intensive Systems (Daisy), in Aalborg University. I would like to thank Prof. Katja Hose for her supervision, insightful comments and improvements on my work. Special thanks go to Assoc. Prof. Vinay Setty for our long-running and fruitful collaboration.

I would also like to thank the Software and Database Systems Laboratory at University of the Peloponnese for the resources supplied in order to complete my research. Through my PhD program I received financial support from the EICOS

Research Project, Research Center Athena. I would like to thank all the people in the institution for the research grants.

Abstract

In the modern digital era, the creation and availability of new information has increased exponentially. A plethora of information sources, such as news delivery sites, knowledge bases, and social networks, constantly make new content available at an overwhelming pace. To assist users in coping with the vast amount of newly generated information the *Information Filtering* (IF) paradigm was introduced. IF applications aim at assisting users in information discovery and enable users to cope with the information avalanche and the cognitive overload associated with it.

In an IF scenario, users or services, express their information needs (implicitly or explicitly) through appropriate interfaces, tools and languages and submit *profiles* (or continuous queries) to a system or service. In this way, users create subscriptions that are continuously matched (by the system or service) against a stream of newly published content, and generate notifications whenever new items that match users' information needs are published. The filtering problem is of high importance and needs to be solved efficiently, since servers are expected to handle millions of queries and high rates of incoming content.

In our work, we examine the research problem of developing efficient and effective algorithms that are able to capture the nature of information streams through the form of *continuous multi-query answering.* To this end, we choose to explore solutions under the domains of textual information filtering, ontology publish/subscribe systems and evolving graph stream environments. Finally, we design, implement and present a fully-functional information filtering system that showcases the usefulness of the IF paradigm and provides the basis for developers to build added-value IF services in a number of different information domains.

At first we examine the information filtering paradigm, under the scope of textual information filtering while employing the Boolean data model. In this setup clients

subscribe to a server with continuous queries that express their information needs and get notified every time appropriate information is published. To perform this task in an efficient way, servers employ indexing schemes that support fast matches of the incoming information with the query database. However, state-of-the-art indexing schemes are sensitive to the query insertion order and cannot adapt to an evolving query workload, degrading the filtering performance over time. In this line of work, we present an adaptive trie-based algorithm that outperforms current methods by relying on query statistics to reorganize the query database. In our research, we explore query database reorganization techniques and demonstrate that the nature of the constructed tries, rather than their compactness, is the determining factor for efficient filtering performance. Our algorithm does not depend on the order of insertion of queries in the database, manages to cluster queries even when clustering possibilities are limited, and achieves two orders of magnitude filtering time improvement over its state-of-the-art competitors. Finally, we demonstrate that our solution is easily extensible to multi-core machines by providing an implementation in a multi-core environment.

In the continuation of our work, we investigate publish/subscribe ontology systems; we envision a publish/subscribe ontology system that is able to index large numbers of expressive continuous queries and filter them against RDF data that arrive in a streaming fashion. To this end, we propose a SPARQL extension that supports the creation of full-text continuous queries and propose a family of main-memory query indexing algorithms which perform matching at low complexity and minimal filtering time. We experimentally compare our approach against a state-of-the-art competitor (extended to handle indexing of full-text queries) both on structural and full-text tasks using real-world data. The experimental results demonstrate that our approach yields two orders of magnitude faster performance than the competitor in all types of filtering tasks.

Subsequently, in our research we study the domain of evolving graphs that have a wide range of applications involving social networks, knowledge bases and biological interactions. The evolution of a graph in such scenarios can yield important insights about the nature and activities of the underlying network, which can then be utilized for applications such as news dissemination, network monitoring, and content curation. Capturing the continuous evolution of a graph can be achieved by long-standing sub-graph queries. Although, for many applications this can only be achieved by a set of queries, state-of-the-art approaches focus on a single query scenario. Therefore, in this line of work, we introduce the notion of continuous multi-query processing over graph streams and discuss its application to a number of use cases. To this end, we developed a novel algorithmic solution for efficient multi-query evaluation against a stream of graph updates and experimentally demonstrated its applicability. Our results against three baseline approaches and the graph database Neo4j, using real-world and synthetic datasets, confirm a two orders of magnitude improvement of the proposed solution.

Finally, we conclude our research with the design and development of a fully-fledged textual information filtering system coined Ping. The Ping IF system is a fully-functional content-based information filtering system aiming (i) to showcase the realizability of information filtering and (ii) to explore and test the suitability of the existing technological arsenal for information filtering tasks. The proposed system is entirely based upon open-source tools and components, is customizable enough to be adapted for different textual information filtering tasks, and puts emphasis in user profile expressivity, intuitive UIs, and timely information delivery. To assess the customizability of Ping, we deployed it in two distinct information filtering scenarios, while to assess its performance we designed and conducted a series of experiments for both scenarios.

Περίληψη

Στη σύγχρονη ψηφιακή εποχή, η δημιουργία και η διάθεση νέας πληροφορίας γίνεται με ταχείς ρυθμούς. Η επιλεκτική διάχυση πληροφορίας (information dissemination, publish/subscribe) έχει αναπτυχθεί ως το μέσο για την διευκόλυνση της αναζήτησης και έγκαιρης διάδοσης πληροφορίας στους χρήστες, καθώς και της ανακάλυψης νέου και ενδιαφέροντος περιεχομένου.

Τα τελευταία χρόνια, η επιστημονική έρευνα στον τομέα της διάχυσης πληροφορίας έχει επικεντρωθεί στην αναπαράσταση των ενδιαφερόντων των χρηστών που εκφράζονται μέσω της δημιουργίας προφίλ (π.χ., εγγραφές σε υπηρεσίες παροχής ειδήσεων, δημιουργία προφίλ σε κοινωνικά δίκτυα κ.λ.π.) και στην αποτελεσματική και γρήγορη διανομή της πληροφορίας στους χρήστες, όταν αυτή γίνει διαθέσιμη. Ο τεράστιος όγκος δεδομένων όμως που γίνεται διαθέσιμος καθημερινά στον Παγκόσμιο Ιστό απαιτεί αποτελεσματικούς αλγόριθμους τόσο για την αναπαράσταση και ευρετηρίαση των προφίλ (profile creation, profile indexing), όσο και για το φιλτράρισμα της νέας διαθέσιμης πληροφορίας (publication filtering, information dissemination, mutli-query processing). Η παρούσα διατριβή στοχεύει στην επίλυση των παραπάνω προβλημάτων χρησιμοποιώντας σύγχρονες μορφές αναπαράστασης δεδομένων (RDF data, graph data), και προτείνοντας δομές δεδομένων και αλγόριθμους για την διαχείριση του μεγάλου όγκου πληροφορίας.

Η παρούσα έρευνα μελέτησε λύσεις ευρετηρίασης και φιλτραρίσματος πληροφορίας κειμένου βασισμένες σε δεντρικές δομές (trie-based profile indexing), σχεδίασε και ανέπτυξε αλγορίθμους για την ευρετηρίαση δεδομένων μεγάλου όγκου που έχουν ληφθεί από μια πληθώρα συλλογών κειμένων. Οι προτεινόμενοι αλγόριθμοι αξιολογήθηκαν πειραματικά και τα αποτελέσματα που προκύπτουν από την αξιολόγηση υποδεικνύουν βελτίωση έως και δυο τάξεις μεγέθους σε σύγκριση με υπάρχουσες λύσεις της βιβλιογραφίας. Τα αποτελέσματα της έρευνας μας επισημαίνουν ως καίριο

παράγοντα βελτιστοποίησης της αποτελεσματικής απόδοσης του φιλτραρίσματος τις δεντρικές δομές. Πιο συγκεκριμένα, τα αποτελέσματα υποδεικνύουν ότι η μορφολογία και οργάνωση των δεντρικών δομών είναι ο καθοριστικός παράγοντας βελτιστοποίησης, σε αντίθεση με την μέχρι έως τώρα πεποίθηση ότι το μέγεθος των δεντρικών δομών (forest compactness) αποτελεί τον κύριο παράγοντα απόδοσης.

Σε συνέχεια της παρούσας έρευνας, σχεδιάσθηκαν και αναπτύχθηκαν αλγορίθμοι για την ευρετηρίαση και το φιλτράρισμα δεδομένων που αναπαριστώνται στο μοντέλο δεδομένων RDF. Επιπρόσθετα, προτείναμε μια καινοτόμα επέκταση της γλώσσας ερωτήσεων SPARQL, η οποία στοχεύει στην αύξηση της εκφραστικότητας των ερωτήσεων των χρηστών μέσω της παροχής τελεστών κειμένου (full-text operators). Οι αλγόριθμοι που σχεδιάστηκαν και αναπτύχθηκαν αξιολογήθηκαν πειραματικά, και τα αποτελέσματα που προκύπτουν από την αξιολόγηση υποδεικνύουν βελτίωση έως και δύο τάξεις μεγέθους σε σύγκριση με υπάρχουσες καινοτόμες λύσεις της βιβλιογραφίας.

Επιπλέον, η έρευνα μας στόχευσε στη σχεδίαση και ανάπτυξη αλγορίθμων για την ευρετηρίαση και την αξιολόγηση ερωτήσεων σε ροές δεδομένων για γράφους. Η παρούσα έρευνα είναι η πρώτη στη βιβλιογραφία η οποία εισάγει την συνεχή αξιολόγηση πολλαπλών ερωτήσεων (mutli-query processing) πάνω από ροές δεδομένων για γράφους. Πιο συγκεκριμένα, σχεδιάσαμε και αναπτύξαμε τέσσερις νέους αλγορίθμους με σκοπό την μελέτη και αξιολόγηση της απόδοσης διαφορετικών προσεγγίσεων ευρετηρίασης προφίλ. Η αξιολόγηση στόχευσε στην εκτίμηση της απόδοσης των αλγορίθμων σε ένα ευρύ πεδίο εφαρμογών, όπως τα κοινωνικά δίκτυα (Social Networks), τα δίκτυα κίνησης οχημάτων σε αστικά κέντρα (Road Networks), και οι γράφοι αλληλεπιδράσεων πρωτεϊνών (Protein-to-Protein Interaction Graphs), και στην αξιολόγηση και στην σύγκριση των σχεδιασθέντων αλγορίθμων με υπάρχουσες εμπορικές λύσεις. Τα αποτελέσματα της πειραματικής αξιολόγησης τονίζουν την ανάγκη για ανάπτυξη εξιδεικευμένων λύσεων σχεδιασμένων για συνεχή αξιολόγηση ερωτήσεων σε ροές δεδομένων γράφων, καθώς παρατηρήθηκε βελτίωση του χρόνου φιλτραρίσματος κατά δυο τάξεις μεγέθους ανάμεσα στους προτεινόμενους αλγορίθμους και στις πιο απλοϊκές προσεγγίσεις.

Τέλος, η έρευνα μας επικεντρώθηκε στην σχεδίαση και ανάπτυξη ενός καινοτόμου, πλήρως λειτουργικού, συστήματος φιλτραρίσματος πληροφορίας κειμένου, με την ονομασία Ping. Η ανάπτυξη του συστήματος Ping στόχευσε στη μελέτη υπαρχόντων τεχνολογικών λύσεων υπό το φως της διάχυσης πληροφορίας, και στη δημιουργία ενός πλήρως λειτουργικού συστήματος παροχής υπηρεσιών φιλτραρίσματος για τους χρήστες. Η δημιουργία ενός τέτοιου συστήματος αναδεικνύει την εφαρμοσιμότητα προηγμένων τεχνολογικών λύσεων στον τομέα της διάχυσης πληροφορίας.

Contents

List of Figures

List of Tables

List of Abbreviations

HTTP	Hypertext Transfer Protocol
IF	Information Filtering
IR	Information Retrieval
P2P	Peer to Peer
PPIs	Protein-Protein Interactions
Pub/Sub	Publish/subscribe
RDF	Resource Description Framework
RSS	Rich Site Summary
VSM	Vector Space Model
UI	User Interface
XML	Extensible Markup Language

1 Introduction

In this thesis, we examine the problem of efficient multi-query processing under the scope of information filtering. We explore the problems of developing efficient and effective algorithms that are able to capture the nature of information streams through the form of continuous query answering. In this introductory chapter, we define the problem and highlight our contributions.

1.1 Problem Statement

In the modern digital era, the creation and availability of new information has increased exponentially. A plethora of information sources, such as news delivery sites, knowledge bases, and social networks, constantly make new content available at an overwhelming pace. To assist users in coping with the vast amount of newly generated information the *Information Filtering* (IF) paradigm was introduced. IF applications aim at assisting users in information discovery and enable them to cope with the information avalanche and the cognitive overload associated with it.

In an IF scenario, users or services express their information needs (implicitly or explicitly) through appropriate interfaces, tools and languages, and submit *profiles* (or continuous queries) to a system or service. In this way, users create subscriptions that are continuously matched (by the system or service) against a stream of newly published content, and generate notifications whenever new items that match users' information needs are published. To this end, processing multiple queries is of high importance and needs to be solved efficiently, since servers are expected to handle millions of queries and high rates of incoming content.

When studying the problem of multi-query processing under gradually more challenging domains, there is a plethora of open and interesting research questions that arise. In this thesis, we address four interesting research questions in the area of multi-query processing, and provide efficient algorithmic solutions to each one of them:

- Can the current state-of-the-art multi-query processing solutions be pushed forward in terms of efficiency? Does it make sense in the current multi-processor landscape?
- Is it possible to enhance the expressiveness of current multi-query processing solutions, without significant overhead in efficiency?
- Does multi-query processing fit as a computing paradigm to new emerging domains, such as evolving graphs?
- Is the current technological arsenal mature enough to support the design of a general-purpose open-source information filtering system across different domains?

In this thesis, we choose to study the problem of multi-query answering under the scope of information filtering and employ it over three interrelated information domains. To this end, we study and propose efficient algorithmic solutions that support the text-based Boolean data model, ontology-based systems and evolving graphs. Each information domain bears its own distinct restrictions and challenges posed by the nature of the query and data models. The research results reported in this thesis highlight the importance of developing efficient algorithmic solutions for multi-query answering in each information domain. Additionally, in our work we present the design and development of a fully-functional information filtering system. Below, we present a brief overview of the research domains we choose to study; we highlight the challenges and formulate the research questions that arise in each domain.

Multi-Query Processing over Textual Data. Firstly, we choose to investigate efficiency issues under the domain of textual IF. In this context, applications such as news alerts, digital libraries, or RSS feeds, employ mostly textual information, while users express their needs using information retrieval languages (e.g., Boolean combinations of keywords [1] or text excerpts under the Vector Space Model – VSM [1]) and submit *continuous queries* (or *profiles*) to a server, thus, *subscribing* to newly appearing documents that will satisfy the query conditions. In the past, efficiency issues were identified by many researchers that proposed tree and trie-based algorithms for supporting fast filtering under various data models (e.g., flat attribute-based, semi-structured XML) and query languages (e.g., Boolean, VSM), both for main-memory [2–4] and secondary storage [5]. However, all these approaches use a greedy clustering method that is sensitive to the insertion order of submitted queries and do not consider an evolving query workload, which might require the reorganization of the query database to achieve efficient filtering performance. To this end, it is important to investigate and develop algorithms that alleviate the drawbacks of greedy clustering techniques, and thus delivering significant improvements in the resource intensive information filtering process. Having studied the domain of textual IF we transferred our findings to the more complex domain of textual ontology-based systems.

Multi-Query Processing over Ontology-Based Systems. In the domain of ontology-based information filtering research [6–9] has naturally focused more on semantics and has delivered interesting results. What it currently lacks, though, compared to the technological arsenal of the traditional IF research is the support of a complete full-text filtering mechanism, beyond existing regular expression and equality support, with sophisticated algorithms and data structures to minimize processing and memory requirements. Providing a full-text filtering mechanism over ontologies may complement many applications, in knowledge bases, triple stores, graph databases and LOD platforms such as Lotus[1] by enabling users to get notified for information of interest, or by providing a useful moderation/monitoring tool for curators/editors of such systems. Having addressed the most interesting challenges that arise in ontology-based systems under the IF paradigm, we extend our research in a graph-based data model.

Multi-Query Processing over Evolving Graphs. In the domain of graphs, we aim at gaining meaningful and up-to-date insights in frequently updated graphs. It is essential to be able to monitor and detect continuous patterns of interest. There are several applications from a variety of domains that may benefit from such monitoring. In social networks, such applications may involve targeted advertising, spam detection [10, 11], and fake news propagation monitoring based on specific patterns [12, 13]. Similarly, other applications like (i) protein-to-protein interaction patterns in biological networks [14, 15], (ii) traffic monitoring in transportation networks, (iii) attack detection (e.g., distributed denial of service attacks in computer networks), (iv) question answering in knowledge graphs [16, 17], and (v) reasoning over RDF graphs may also benefit from such pattern detection. To this is end, it is necessary to express the required patterns as *continuous sub-graph queries* over (one or many) streams of graph updates and appropriately *notify* the subscribed users for any patterns that match their subscription. Detecting these query patterns is fundamentally a sub-graph isomorphism problem which is known to be NP-complete due to the exponential search space resulting from all possible sub-graphs [18, 19]. The typical solution to address this issue is to pre-materialize the necessary sub-graph views for the queries and perform exploratory joins [20]; an expensive operation even for a single query in a static setting. To this end, in our research we focus on providing efficient algorithms that can capture patterns in large evolving graphs. Capitalizing on our experience from all the previously examined information domains, we finally aimed at building a functioning open-source IF system.

Realizing Information Filtering. Finally, we examine the problem of designing and providing a fully-functional IF system, contrary to previous attempts in the

[1] http://lotus.lodlaundromat.org/

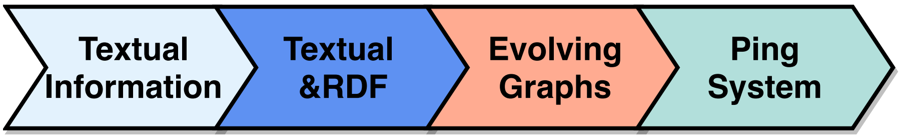


Figure 1.1: A roadmap of the research presented in this thesis.

literature that aimed at providing solely algorithmic solutions [2, 5, 21–24]. The lack of IF tools that would integrate promising solutions and allow developers to use them for building added-value IF services over textual sources or streams, resulted in the lack of prominent IF systems that would act as demonstrators for the usefulness of the IF paradigm. Thus, currently, the only prominent demonstrator of the potential of IF is Google Alerts [25], a proprietary closed-source service built upon the Google ecosystem. Although many users nowadays (mis-)use Google Alerts to monitor the web for marketing (e.g., brand mentions), social listening (e.g., comment follow-up), or even citation counting purposes (e.g., in the context of GoogleScholar), there is clearly a need for an extensible, customizable open-source IF system that could be modified to fit domain-specific IF tasks.

1.2 Solution Outline

In this section, we give an overview of the developed algorithmic solutions that address the open problems of multi-query answering discussed in the previous section. To this end, we present the solution outlines on information filtering when employed under the domains of text processing, ontology systems, and the evolving graphs domain. Finally, we give the outline for developing a fully functional and easily customizable information filtering system. Figure 1.1 presents a short overview of each gradually more challenging research domain that we present in this thesis.

Efficient Continuous Multi-Query Processing over Textual Data

In the domain of *textual information filtering*, clients subscribe to a server with continuous queries that express their information needs and get notified every time appropriate information is published. We concentrate on *textual IF* and present a novel *trie-based*, *main-memory* algorithm for *Boolean IF* that is able to match incoming documents against millions of queries in a few milliseconds. Our methods use linguistically motivated concepts, such as words, to support continuous queries that are comprised of *conjunctions of keywords* and may be used as a basis for query languages that support not only basic Boolean operators, but also more complex constructs, such as proximity operators and attributes. We believe that offering an efficient Boolean filtering service (possibly alongside a more popular model like VSM) is a valuable addition to any text filtering setup. Boolean IR/IF is still the model of choice of advanced users that want total control of their results and is widely supported in systems of major stakeholders like Google's advanced search/alert mechanisms, Oracle's text extender module, and in Apache's text search engine. Such systems, that are meant to cope with a high workload and are designed for efficiency, are possible applications for our work.

The algorithm we developed, coined StaR (an acronym derived from STAtistical Reorganization) is based on the idea to use tries to capture common elements of queries, similarly to [3–5]. However, the key differences with these approaches lie in:

- The collection and utilization of statistics on the importance of keywords in the indexed queries.
- The reorganization of the query database according both to *word* and *query importance.*
- The proposed solution is the first in the literature to consider *database reorganization*, through appropriate word/query statistics, to achieve efficient textual IF under the Boolean model.
- The demonstration that the nature of the trie forest is more important than its compactness when it comes to filtering efficiency.

Interestingly, all previous works [3–5] were aiming at *minimizing the size* of the trie forest, since there was an implicit conjecture that a small forest would result in lower filtering times due to less node visits. Our findings demonstrate that forest size is not the dominating *optimization factor* when it comes to filtering efficiency; contrary, the focus should be put on the nature of the tries and on qualitative characteristics (expressed through heuristics). To this end, Algorithm StaR overcomes the query insertion order problem caused by the greedy query clustering techniques adopted by all other algorithmic solutions [2–5].

Efficient Continuous Multi-Query Processing over Textual and RDF Data

In the continuation of our work, we aimed at studying modern ontology-based pub/sub systems [6–9]. These works focused more on semantics and has delivered interesting results in the domain of content-based information filtering. What they currently lacked, though, compared to the technological arsenal of the traditional pub/sub research was the support of a complete full-text filtering mechanism, beyond existing regular expression and equality support, with sophisticated algorithms and data structures to minimize processing and memory requirements. In order to address the lack of a complete full-text filtering mechanism, we proposed an *extension* of SPARQL with full-text operators, aiming at more expressive continuous queries that are able to support versatile user needs in applications like digital libraries or news filtering. To preserve the expressiveness of SPARQL, we view the full-text operations as an additional filter of the query variables. In our approach, publications are ontology data that contain RDF literals in their property elements. A full-text expression is evaluated against a literal, and supported expressions involve the usual Boolean operators (i.e., conjunction, disjunction, negation), as well as word proximity and phrase matching as in Chang et al. [26].

In order to address the filtering problem in the context of ontology-based systems, we developed Algorithm RTF (acronym for RDF Text Filtering), a family of trie-based, main-memory, (continuous) query indexing algorithms that support SPARQL queries with full-text constraints and are able to filter incoming publications in a few milliseconds. We propose indexing methods that exploit the commonalities between continuous queries at indexing time and leverage on the natural properties of RDF during the filtering procedure. To the best of our knowledge, our work is the first in the literature that:

- Proposes an *extension* of SPARQL with full-text operators, aiming at more expressive continuous queries.

- Proposes a family of algorithms that is able to support SPARQL queries with full-text constraints.

- Studies and extends iBroker developed by Park et al. [9], a state-of-the-art query indexing and RDF publication filtering algorithm, with full-text capabilities and compare it against our approach both on structural and full-text filtering.

Efficient Continuous Multi-Query Processing over Evolving Graphs

In the context of developing efficient multi-query processing algorithms and thus, capturing interesting insights on the nature of data in large continuously-evolving graphs, it is necessary to express the required restrictions in a concise manner. We choose a query subscription language in the form of *continuous sub-graph queries* that is able to capture information over (one or many) streams of graph updates and appropriately *notify* the subscribed users for any patterns that match their subscription. Detecting these query patterns is fundamentally a sub-graph isomorphism problem which is known to be NP-complete due to the exponential search space resulting from all possible sub-graphs [18, 19]. The typical solution to address this issue is to pre-materialize the necessary sub-graph views for the queries and perform exploratory joins [20]; an expensive operation even for a single query in a static setting.

One simple approach to avoid processing all the (continuous) queries upon receiving a graph update is to index the query graphs using an inverted-index at the granularity of edges. While this approach may help us quickly detect all the affected queries for a given graph update, we still need to perform several exploratory joins to answer the affected queries. On the contrary, if we first identify the maximal sub-graph patterns shared among the queries instead, we can minimize the number of operations necessary to answer the queries and this will consequently reduce the query answering time. We address this gap in the literature by proposing a novel *algorithmic solution*, coined Algorithm TriC (TRIe-based Clustering) to index and cluster continuous graph queries. In TriC, we first decompose queries into a set of directed paths such that each vertex in the query graph pattern belongs to at least one path (path covering problem [27]). However, obtaining such paths leads to redundant query edges and vertices in the paths; this is undesirable since it affects the performance of the query processing. Therefore, we are interested in finding paths which are shared among different queries, with minimal duplication of vertices. The paths obtained are then indexed using 'tries' that allow us to minimize query answering time by (i) quickly identifying the affected queries, (ii) sharing materialized views between common patterns, and (iii) efficiently ordering the joins between materialized views affected from the update. To this end, in our work:

- We aim at providing a modern scalable solution (Algorithm TriC) which groups queries based on their shared patterns and thus, expect to deliver significant performance gains. To the best of our knowledge, none of the existing works provide a solution that exploits common patterns for continuous multi-query answering.

- We design and develop advanced baseline solutions that employ "inverted indexing techniques" for capturing commonalities among query sets and utilize them to study different approaches during the experimental evaluation time.

- We experimentally demonstrate that our proposed solution, Algorithm TriC, provides a speedup of two orders of magnitude in query answering time, compared against the advanced baselines, as well as when compared against a production-ready graph database (Neo4j) that do not exploit the common sub-graph patterns in the queries.

Realising IF in an Open-Source System

In the last part of our research, we focused on providing a full-fledged, customizable, open-source IF system, coined Ping, that makes use of state-of-the-art tools and web technologies; we concentrate on providing an operational system that is designed and implemented on IF-specific requirements. To this end, we present a system equipped with profile administration (e.g., creation, modification, submission), publication management capabilities (e.g., collection, filtering), different content delivery options (e.g., email or on-site notifications), (interval- or batch-based) monitoring of different types of textual data, and an intuitive user interface. The front-end of the system is built upon modern Internet technologies, while the back-end relies on the well-established Apache Solr[2] platform. Ping is designed with flexibility and customizability in mind; developers may use it to easily create textual IF engines for different domains, parameterize and deploy it for IF-specific tasks over their own textual information sources, or use it as a building block for added-value services. To demonstrate the customizability of Ping, we deployed and experimented with it in two different textual IF scenarios: the DBLP[3] database for scientific publications and the textual part of the DBpedia[4] open-sourced knowledge graph. Using Ping, we easily created an IF system that allows users to express their information needs and stay notified for new and interesting publications. To this end, our work on the Ping system can be summarized as follows:

- We present Ping, a novel, fully-functioning IF system build entirely upon open-source components; the proposed system is able to support complex IF tasks in a variety of domains.

- We showcase the realizability of the developed system on two different domains (textual IF on scientific publications and crowd-sourced encyclopedia articles), and experimentally assess its performance.

[2] http://lucene.apache.org/solr/

[3] https://dblp.uni-trier.de/

[4] https://wiki.dbpedia.org

1.3 Contributions

Studying the problem of multi-query answering under gradually more challenging information domains has yielded a plethora of interesting and novel research insights. To this end, in this section we give an overview of our contributions, as well as detailed lists of contributions for each information domain. In the context of multi-query answering when designing algorithmic solutions for IF systems, our research makes the following important contributions:

- It studies and proposes novel algorithmic solutions in the domain of text-based IF, that surpass current state-of-the-art approaches. The research yields important insights that challenge the status quo with regard to efficiency optimization strategies and take into account multi-processor setups.

- It builds upon the groundwork conducted in text-based IF systems, and introduces full-text support in ontology-based IF. The research proposes, a full-text extension for the SPARQL query language, while it yields the first multi-query processing algorithmic solution in the literature that supports full-text operators over the RDF data model as a first class citizen.

- It is the first in the literature that breaks ground in multi-query processing over evolving graphs. The research capitalizes on the knowledge gathered from previous research insights and builds novel algorithmic solutions to efficiently address the resource-demanding multi-query processing over graph streams.

- It exploits the important insights and research experience gathered, and employs it in the development of a modern open-source IF solution that can be deployed under multiple information domains.

Below, we further discuss the aforementioned high-level contributions and give details in regard to the advances achieved in each information domain.

Initially in our research, we focus on the domain of textual information filtering under the Boolean model, and highlight the drawbacks of greedy query indexing algorithms and their inability to scale under large and evolving query sets. Additionally, our work identifies the importance of efficient data structure construction and demonstrates the effect of different organization strategies in filtering performance. In the context of textual Boolean information filtering, our work makes the following advances in the current state-of-the art:

- It presents a novel trie-based query indexing and reorganization algorithm, Algorithm STAR, that supports Boolean IF up to 96% faster than state-of-the-art competitors. The query reorganization process is highly efficient and takes less than a few seconds on a commodity PC to reorganize, as many as, $500K$ queries in a database comprised of millions of queries.

- It extensively studies and identifies different reorganization options for the trie indexes and demonstrates the importance of query insertion order in the construction of the indexing structure. It evaluates different reorganization strategies and showcases their effect in filtering efficiency using two different real-world datasets and both synthetic and real query sets. Through the experimental process it is demonstrated that constructing tries with rare words at the higher level of the trie leads to improved filtering performance due to early pruning at filtering time.

- It proves through the experimental evaluation, contrary to previous works, that the nature of the constructed tries, rather than their compactness, is the determining optimization factor for efficient filtering performance, especially in datasets with rare clustering opportunities.

- It provides parallelization of the filtering process to suit modern multi-core processors, and identifies two different parallelization options and presents the experimental evaluation results.

Subsequently, our research identified the importance of providing full-text filtering capabilities under ontology-based systems. To this end, we proposed a SPARQL extension with full-text operators, and developed a family of continuous query indexing algorithms that can complement existing application scenarios. Thus, in the context of ontology-based system and multi-query answering, our work makes the following contributions:

- It proposes a SPARQL extension with full-text operators that supports Boolean, word proximity, and phrase matching operators. The SPARQL extension aims at providing users with expressive tools through full-text subscriptions beyond existing regular expression and equality support.
- It provides a family of continuous query indexing algorithms, coined RTF, that can support the proposed full-text SPARQL extension and can efficiently filter queries against incoming RDF publications.
- It extends a state-of-the-art competitor Algorithm iBroker [9], an ontology-based pub/sub filtering solution and thus, enables it to support the full-text operations.
- It identifies and evaluates different algorithmic alternatives for query indexing, while it assesses their performance with a real-word data set against the extended version of Algorithm iBroker.
- It demonstrates that Algorithm RTF is more than two orders of magnitude faster for the structural and more than one order of magnitude faster for the full-text filtering tasks, under different evaluation setups.

In continuation of our research, we studied and formalized the problem of continuous multi-query answering over graph streams. Our research aimed at providing algorithmic solutions that can capture insights on the nature of data in large and continuously evolving graphs. Our work is the first approach in this area that makes the following important advances:

- It studies and formalizes the problem of continuous multi-query answering over graph streams. Furthermore, it identifies, presents and motivates *application scenarios* from various domains, that could benefit from the proposed continuous multi-query answering paradigm.

- It proposes a novel trie-based query graph clustering algorithm, coined TriC, that is able to efficiently handle large numbers of continuous graph queries by resorting on (i) the decomposition of continuous query graphs to covering paths and (ii) the utilization of tries for capturing the common parts of those paths.
- It proposes two algorithmic solutions that utilize inverted indexes for the query answering, since no prior work in the literature has considered continuous multi-query answering.
- It identifies variations of the main algorithmic solution and the baseline approaches, which utilize caching strategies and demonstrates the effect of such solutions on the problem at hand.
- It deploys and extends the well-established graph database Neo4j [28], under the mutli-query evaluation paradigm, and it utilizes Neo4j as a baseline solution during the experimental evaluation time.
- It experimentally evaluates the proposed solution using three different datasets from social networks, transportation, and biology domains, and compare the performance against the three baselines. To this end, it demonstrates that the trie-based solution can achieve up to two orders of magnitude improvement in query processing time.

Concluding this thesis, we present Ping, a novel, fully-functioning IF system build entirely upon open-source components; the proposed system is able to support complex IF tasks in a variety of domains, while it highlights the need for and the importance of developing efficient IF-specific machinery to facilitate higher-level IF systems. To this end, our work and development of the Ping system makes the following advancements:

- It presents a system that is flexible enough *(i)* to be deployed as a standalone solution on different textual IF tasks and domains or *(ii)* to be used as a building block for other added-value services.

- It showcases the realizability of an IF system under two different domains (textual IF on scientific publications and crowd-sourced encyclopedia articles), while it experimentally assesses its performance.

- It provides an implementation of the information filtering functionality over the Solr framework, which is primarily designed for information retrieval tasks, and highlights the lack of native information filtering tools.

1.4 Thesis Outline

This thesis is formulated of eight distinct chapters, where the first is the current introductory chapter. The rest of this thesis is organized as follows.

Chapter 2 presents the related research in the domains of textual and content-based information filtering systems, as well as the literature regarding graph-based solutions under the scope of multi-query evaluation. Chapter 3 presents our work conducted in the domain of *textual information filtering*, identifies the importance of *efficient information filtering* and presents the algorithmic solutions developed to efficiently solve the problem at hand. Chapter 4 presents our work conducted in the domain of content-based information filtering systems, where we identify the importance of *efficient* and *effective information filtering*, and present the algorithmic solutions developed to address these issues. Chapter 5 presents our work conducted in the domain of evolving graphs and graph streams, where we introduce the notion of *continuous multi-query processing over graph streams* and discuss its applications to a number of use cases, while we present the novel algorithmic solutions developed to address these applications. Chapter 6 presents Ping, a fully-functional information filtering system for scientific publications, where we showcase the realizability of information filtering and explore the suitability of the existing technological arsenal for information filtering tasks. Finally, Chapter 7 summarizes the thesis and discusses future directions for this research.

2
Related Research

In this chapter, we present the work related to our research, the works discussed span from a wide range of applications as the information filtering paradigm is ubiquitous and ever present. In this thesis, we examine the information filtering paradigm under three distinct application scenarios: (i) textual information filtering, (ii) content-based information filtering, and (iii) graph-based information filtering.

2.1 Text-Based Information Filtering

In recent years, *information filtering* (IF) applications (also known as *information dissemination* or *publish/subscribe*), such as news alerts, weather monitoring, and stock quotes, have gained popularity. Such applications assist users to cope with the information avalanche and the cognitive overload associated with it. For the case of news alerts, digital libraries, or RSS feeds, where the data of interest is mostly textual, users express their needs using information retrieval languages (e.g., Boolean combinations of keywords [1] or text excerpts under the Vector Space Model – VSM [1]) and submit *continuous queries* (or *profiles*) to a server, thus, *subscribing* to newly appearing documents that will satisfy the query conditions. The server will then be responsible for *notifying* the subscribed users *automatically* whenever a new document that matches their information needs is published. Publishers can be news feeds, digital libraries, or even users who post new items to blogs, social media, and Internet communities. This functionality is very different from information retrieval (IR) applications like search engines [1]. Specifically, in IR when a query is posed, a single search is executed and the current matching data items are presented to the user. Contrary, in IF the server indexes the *user queries* rather than the data and evaluates newly published data items against the stored continuous queries.

In more detail, the problem of information filtering may be defined as follows: given a database DB of continuous queries that reside on a server and an incoming document d, retrieve all queries $q \in DB$ that match d. The filtering problem is of high importance and needs to be solved efficiently, since servers are expected to handle millions of user queries and high rates of published documents. Efficiency issues were identified by many researchers that proposed tree and trie-based algorithms for supporting fast filtering under various data models (e.g., flat attribute-based, semi-structured XML) and query languages (e.g., Boolean, VSM), both for main-memory [2–4] and secondary storage [5].

In the following sections we present works that have focused on developing text-based information filtering algorithmic solutions and data structures. We choose to classify the research presented in two main sets, works that developed solutions for *centralized* systems and works that focus on providing *decentralized* solutions.

2.1.1 Centralized Text-Based Information Filtering

Historically, work on selective dissemination of information started by a 1958 article of Luhn [29], where a “Business Intelligence System” is described. In the proposed concept, individual users would have their interests described in profiles, and a text selection system would produce lists of new documents that would allow users to choose between ordering a new document or not. At that day, the selection module was described using the terms *selective dissemination of new information*.

One of the first filtering systems based on the Boolean model to address performance was LMDS [30] that relied on least frequent trigrams for query indexing. In LMDS, profiles are indexed under the least frequent trigram, whereas documents are represented as a sequence of trigrams. At filtering time a table lookup determines which profiles match the incoming document. Since false positives may incur, a second stage is necessary to determine the actual matches.

One of the first works to describe an IF system capable of scaling up to large filtering tasks was presented by Bell and Moffat [31]. In this work, the authors proposed an IF system that is able to scale up to thousands of queries, by combining solutions already present in the literature and extending those works in order to support the VSM model. The authors built upon their previous work in [32] that proposed a document matrix for indexing incoming documents and subsequently utilize said matrix during filtering time, while their work extended Algorithm SQI of Yan and Garcia-Molina [21] with VSM functionalities. Finally, the proposed approach was able to create a scalable information filtering system, that can efficiently answer incoming publications against a high number of user profiles.

The work conducted by Yan et al. [5] was one of the first works to highlight the importance of efficient Selective Dissemination of Information (SDI) systems. In their work, the authors recognize and discuss data structures traditionally employed for IR systems that are not deemed suitable for IF scenarios. The work asserts the importance of developing efficient data structures and algorithms that can scale for large Information Filtering scenarios. Bearing this in mind the authors propose and study four indexing structures and document matching algorithms, that aim at efficiently addressing the Information Filtering scenario. The four indexing structures proposed at the time were: (i) a *Brute-Force* method that indexes the continuous queries in the disk, (ii) a *Counting method* that utilizes an inverted index to store the queries in order to reduce the number of queries to be examined, (iii) a *Key method* that selects random words as keys for the profiles and thus reducing the association of frequent words with queries, and finally (iv) a *Trie-Based* solution that indexes the queries in a trie structure and aims at storing compactly the query database by sharing the common prefixes of the queries. Furthermore, the authors study the effect of ranking information in the data structures presented.

The experimental evaluation yields that Brute-Force method as the worst performer, while the Tree-Based solution is deemed as the fastest of the proposed solutions.

The work conducted by Callan et al. [33] was on one of the first that highlighted and tried to address the key design differences of IF systems when compared against traditional IR approaches. Thus, the authors designed and developed an IF system, coined InRoute, that utilized inference networks [34] in a similar manner as the IR system INQUERY [35]. In InRoute, query sets and incoming documents were represented by inference networks, while the filtering process was executed by employing belief propagation techniques. Additionally, the authors developed a profile selection technique, coined MinTerm, which estimated the minimum number of terms (of a given query) that has to match before an incoming document can be considered for evaluation. Moreover, a technique for incrementally calculating corpus statistics was proposed, as opposed to existing IR techniques that required an a priori calculation of the statistics. InRoute was able to outperform existing solutions in the literature that utilized index-less or inverted-index solutions, that primarily excel in information retrieval scenarios.

In the same spirit, Franklin and Zdonik [36] highlighted the importance of IF-tailored solutions and aimed at recognizing and clearly define distinct aspects of the IF paradigm. In their work, the authors study the IF paradigm during its pick (i.e., during the decade of the 90's), while they bring into spotlight the limitations of current technologies, where solutions traditionally used by pull-based systems (IR) are applied on push-based (IF) applications. The authors advocate that information dissemination is only a part of a greater picture of the information delivery realm and define three core aspects of the problem: (i) the information push when compared against the information pull nature of information discovery, (ii) the aperiodic and periodic nature of information delivery, and (iii) the unicast and the one-to-many approaches employed for delivery of information. That are three characteristics of information delivery systems that need to be addressed efficiently in order to develop a robust and fast information dissemination system. However, the authors assert that current technologies (e.g. the HTTP protocol) are designed around pull-based principles, and when applied on the push-based paradigm (through polling techniques) render them unsuitable for the problem at hand. This work solidifies the importance of information delivery technologies specifically designed for the IF paradigm with performance and scalability as its main targets.

In the same manner, Altinel et al. [37] assert that the modern information dissemination applications suffer from scalability issues due to the asymmetric data flow generated from request-response protocols (e.g. the HTTP protocol). Thus, the proposed request-response based solutions are deemed inappropriate for serving as information dissemination mechanisms. To address these challenges, the authors develop a Dissemination-Based-Information System (DBIS) that acts as a middleware layer on information delivery networks. Altinel et al. were the first to provide an information dissemination system equipped with a plethora of information delivery mechanisms through a unified solution. These information delivery mechanisms, included information brokers responsible for query answering, services that were responsible for profile indexing and information filtering services, notifications delivery solutions, as well as, tools for managing the content and topology of the network.

In continuation of their work, Yan et al. developed the Stanford Information Filtering Service (SIFT) [38]; an IF service that aimed at providing users with long-term subscription capabilities. The SIFT service would index the user subscriptions and evaluate them continuously against incoming publications, while it would forward appropriate notifications when new publications that satisfied user subscriptions emerged. In this work, the authors describe the underlying research conducted in order to develop the SIFT information system. To this end, the authors discuss and analyze the query indexing data structures and workload distribution techniques that were developed and evaluated to address the high computational needs of SIFT. Additionally, the authors demonstrate the importance of utilizing data structures that are specifically designed for information dissemination and highlight that traditional IR approaches are not deemed suitable for information dissemination scenarios. Yan and Garcia-Molina, achieved in presenting a real-world system that is able to scale to thousands of profiles and handle thousands of incoming documents on a daily basis.

Publish/subscribe (pub/sub) in the database field focused on the performance of systems with data models based on attribute-value pairs and query languages based on attributes with arithmetic/string comparison operators (e.g., [2, 39–41]). The main focus of the work conducted by Fabret and Jacobsen [39] was on providing efficient filtering solutions for the information filtering problem. Inspired by the work in the field of database systems, they propose main-memory data-structures that aim at optimizing trigger execution functions in relational database systems. To this end, the authors focused on providing efficient data structures, caching techniques and query answering algorithms for enhancing trigger execution. Their solution was aimed at enhancing processor caching strategies in order to minimize the cost of evaluating incoming events against query sets. Additionally, the authors proposed and developed a schema-based clustering technique to minimize the number of subscriptions examined during event processing.

In the spirit of improving information filtering performance Fischer and Kossmann [2] developed and evaluated two distinct batching techniques for clustering and reordering incoming publication events in order to achieve high filtering throughput. The key idea presented in their work focuses on identifying messages with a degree of similarity and clustering them in batches. Subsequently these message batches are evaluated together, resulting in reducing the number of hits performed on the underlying index and eventually the filtering time. The proposed solution exhibits significant reduction in evaluation times but increases significantly the time required during the post-filtering phase. To this end, the authors proposed an additional mini-batching solution that aimed at generating smaller groups of akin messages in order to alleviate the setbacks of the first batching technique. The developed solutions demonstrate an improvement of up to one order of magnitude against solutions in the literature that do not employ any batching strategy.

Campailla et al. [41] focus on providing efficient information filtering algorithms through the utilization of Binary Decision Trees. In their approach, the authors, define each subscription as a set of atomic boolean formulas, while they utilize Binary Decision Diagrams (BDDs) to index the boolean formulas. The key idea behind the utilization of BDDs is to capture common sub-expressions present in atomic boolean formulas of the query set, evaluate these common elements in bulk, and thus provide a fast filtering execution time. Finally, the utilization of the query clustering technique demonstrates scaling capabilities of up to half a million user subscriptions.

The research conducted by Sadoghy et al. [42, 43] is a work in the area of IF that focuses on subscriptions expressed as boolean predicates. In their works, the authors designed and developed a novel boolean expression tree indexing structure, coined BE-Tree. The BE-Tree data structure aims at solving the problem of efficient boolean expression answering, over a high dimensional space. The solution employs space partitioning techniques based on the attributes present in predicates of subscriptions. Additionally, BE-Tree generates subsumption hierarchies of continuous boolean expressions and filters them against incoming events. The BE-Tree solution is coupled with a self-adjustment technique, that adapts the tree data structure based on the workload present at the system. The main focus of this data structure, which lies on arithmetic and string operations, is not directly applicable on the textual IF paradigm.

Overall, in the context of textual information filtering, works have focused on efficiency issues where many researchers proposed tree and trie-based algorithms for supporting fast filtering under various data models and query languages (e.g., Boolean, VSM), both for main-memory [2–4] and secondary storage [5]. Additionally, other works have focused on providing efficient solutions under less expressive data models [2, 39–43]. The majority of these approaches rely on greedy clustering methods that are sensitive to the insertion order of submitted queries and do not consider an evolving query workload, which might require the reorganization of the query database to achieve efficient filtering performance. To this end, it is important to investigate and develop algorithms that alleviate the drawbacks of greedy clustering techniques, and thus delivering significant improvements in the resource intensive information filtering process.

2.1.2 Decentralized Text-Based Information Filtering

In their work, Koubarakis et al. [44] adopt the information dissemination paradigm over a peer-to-peer (P2P) network. In such a scenario, the users utilize middle-agents to post their continuous profiles, while publishers make use of the aforementioned agents to push out new content. Agents are responsible for indexing the profiles, routing them to other agents, as well as, filtering the incoming publications against the profile database and sending appropriate notifications to the users. This work highlights and addresses the importance of data models that are designed for decentralized textual information dissemination. To this end, the authors propose three distinct data models namely WP, AWP and AWPS for textual information dissemination, while they highlight two important aspects of the problem at hand, firstly the satisfiability of profiles (i.e., deciding if a document satisfies a profile) and secondly the efficiency problem (where agents must be able to handle millions of profiles and thousands of incoming publications efficiently).

In order to address the efficiency problem of information dissemination, the authors extend their previous work ([44]) by developing algorithms designed for fast textual information filtering. Tryfonopoulos et al. [3] focused on developing information filtering algorithms for the AWP data model. In their work they describe a new algorithm coined BestFitTrie and LCWTrie that was able to index long standing queries by organizing them under a trie data structure. BestFitTrie was able to provide IF capabilities for incoming publications with text and attribute fields. The solution proposed was one of the first in the literature that was able to perform information filtering while supporting equality operators and word proximity operators in the same manner as IR-based models.

Tryfonopoulos et al. [4] investigate the importance of query clustering and the effect of query organization in trie structures. The authors quantify the quality of query clustering by proposing a metric that is able to capture and characterize queries with discreet values. This metric can be utilized in order to determine if a query that is below or over a certain threshold is “properly-clustered” or “well-clustered” respectively. In addition to this metric, the authors develop a reorganization algorithm that is able to relocate poorly clustered queries in better positions inside the trie structure. Their solution coined ReTrie is build upon the existing solution in the literature BestFitTrie [3] and is able to outperform it along other state-of-the-arts works by up to 20%. This work is the first in the literature that investigates clustering quality and highlights its importance as an optimization factor in the data structures, as well as to demonstrate its effects on the performance of the filtering process.

2.1.3 Personalization and Effectiveness

Many IF efforts focused more on appropriate representations of user interests [45, 46] and on improving filtering effectiveness [47, 48]. In [45], behavior monitoring and substring indexing are used to decide which documents match user interests. Moreover, [47, 48] explores an ensemble of methods from machine learning, aiming at increasing filtering effectiveness in an IF setup. Other approaches included statistical filtering systems, such as [49] that uses Latent Semantic Indexing to filter incoming documents and [50] that utilizes network-based profile representations to better identify user interests and cope with the curse of dimensionality in VSM. Adaptive filtering [51, 52] focuses also on profile effectiveness and considers the adaptation of VSM queries and their dissemination thresholds. In order to enhance user information discovery, [53] developed a novel statistical latent class model that applies user/item grouping to deliver better content recommendations/predictions. Moreover, sophisticated user profiling has also been used to promote personalized IR systems (e.g., [54]) that focus on improving retrieval effectiveness. While recently, works have focussed in improving recommendations and predictions inside social networks [55–57] .

Morita and Shinoda [45] advocate the importance of recognizing users' needs through the formation of accurate profiles and determining which pieces of information are important to the users, thus classifying them useful or not. In their research the authors propose two techniques in order to improve information delivery effectiveness. The first technique requires users to annotate publications delivered to them as interesting or not, while the second proposed technique employs user behavior monitoring. In this scenario, user monitoring takes into account user engagement factors, such as time spent reading an information piece normalized by the its size, readability of information articles (denseness) and total count of articles left for the user to read. Finally, the authors demonstrate that the combination of the two developed techniques can improve the effectiveness of the system and thus, deliver more relevant information to the end users. However, a drawback of this approach lies in the fact that it can not effectively determine information needs that are not previously expressed by the users.

In [47] the authors try to solve the problem of estimating the probability of relevance of a document against a query. Their efforts lie on providing combinations of statistical classifiers that can effectively determine the relevance probability of an incoming document, as well as improve the filtering time. Li et al. [48] try to increase the effectiveness of information filtering systems by reducing the mismatch and overload issues. Thus, they develop a two-stage approach that aims primarily to utilize topic filtering in order to reduce the size of the incoming document set, while the second stage of the solution employs taxonomy matching in order to locate relevant documents.

Rao et al. [24] aim at providing personalized information delivery through continuous information filtering. In their work, the authors propose a solution that can effectively and efficiently determine the top-k most relevant publications that arrive in an IF system within a sliding time window. The proposed solution was designed to solve the problem of continuous top-k query answering over documents by capturing the subsumption relationships present in the query set. To this end, the authors propose a graph-based indexing structure responsible to capture the common relations of VSM queries, and subsequently share the filtering results among similar queries. Additionally, the solution is coupled with a document indexing structure that serves as an archive of already filtered documents. The main purpose of the document archive lies on locating similarities among new incoming publications and previously satisfied ones, resulting in faster location of affected queries and pruning of the query search space. The two proposed data structures are combined into a unified solution, while the experimental evaluation demonstrates that it outperforms existing suggestions in the literature [38, 58] that employ inverted index solutions.

Chang et al. [46] focus on improving the time needed to update query graph indexing structure proposed in [59]. The authors utilize data mining methods that take into account the weight of the keywords in a profile and determine if a query represents a long-term or short-term interest of the user. By making such distinction they are able to index appropriately the short-term queries and provide faster query updates.

2.2 Content-Based Information Filtering

In a pub/sub system, users (or services that act on users' behalf) express their interests by submitting a continuous query and wait to be notified whenever a new event of interest occurs. The vast majority of modern pub/sub services and systems are typically content-based (contrary to previous decades, where they used to be topic/channel based); subscribers express their interest on the content of the publication (be it structure or data/text values) by appropriately specifying constraints in the submitted continuous queries.

In the early days of content-based pub/sub the structure of a publication was nothing more than a (usually static) collection of named attributes with values of different types (e.g., text) [4, 38]. As XML gained popularity and started becoming the standard for data/information representation and exchange on the web, various XML-based pub/sub systems have, naturally, arised [22, 60–63]. In those systems, publications were expressed in XML and extensions of XPath/XQuery were used to express continuous queries. All research in the field focused mainly on the structural/value matching between (indexed) continuous queries and incoming publications, but has largely ignored semantics. This gave rise to ontology-based pub/sub systems [6–9] that typically used RDF [64] for representing publications and SPARQL [65] extensions/modifications for expressing user interests through continuous queries. While other works have focused on increasing the effectiveness of retrieval and data exploration [66–68].

Ontology-based pub/sub systems research [6–9] has naturally focused more on semantics and has delivered interesting results. What it currently lacks, though, compared to the technological arsenal of the traditional pub/sub research is the support of a complete full-text retrieval mechanism, beyond existing regular expression and equality support, with sophisticated algorithms and data structures to minimize processing and memory requirements.

In the following sections, we present influential work from the information filtering domain that focused on developing algorithmic and data structure solutions for content-based and ontology-based systems.

2.2.1 Centralized Content-Based Information Filtering

Current state-of-the-art solutions proposed in information dissemination systems employ approaches and techniques that utilized simple key word matching and "bag of words" techniques. These approaches mainly focused on efficiency and scalability rather than providing users with expressive tools and relevant results (information delivery effectiveness). In order to address these issues, Altinel et al. [23] developed the XFilter information filtering system that aimed at enhancing the expressiveness of long standing user profiles. The XFilter system supported user profiles expressed in the XPath query language, while the incoming publications were expressed in the XML markup language. To this end, the authors developed a novel indexing mechanism for representing XPath queries by making use of Finite State Automatons (FSA) coupled with an XML filtering algorithm. The proposed solutions were able to support expressive queries while the experimental evaluation demonstrated high effectiveness under various workloads.

In continuation of the work done in XFilter, the authors capitalize on the high probability that the elements of a query set share common structural restrictions among each other. The approach employed by the XFilter system does not exploit these commonalities and thus, it leads to redundant processing and increased filtering time. In order to alleviate this problem the authors developed the YFilter system [22]. The YFilter system, similarly to XFilter, aims at efficiently filtering incoming XML documents against user queries while reducing redundant calculations during filtering time. YFilter builds a unified Nondeterministic Finite Automaton (NFA) for the entirety of the query database, as opposed to XFilter approach, where each query is represented under its corresponding FSA. The authors demonstrate that YFilter improved filtering time by orders of magnitude compared to the single NFA per query approach that is employed by XFilter. Additionally, the authors address the issue of processing value-based predicate restrictions under NFA data structures and explore two techniques namely "InLine" and "Selection Postponed". The former evaluation technique, follows a classic Relation Database approach that aims at evaluating the value-based restrictions earlier in the query plan, while the latter aims at evaluating the value-based restrictions after a final state is reached on the FSM. Finally, the experimental evaluation asserts that the "Selection Postponed" can yield significant performance improvement. This work conducted in YFilter highlights the importance of locating and clustering common elements of the query set as this approach yields great improvements in the filtering throughput.

Similarly to the works presented before, Green et al. [69] aim at providing efficient continuous XPath answering over incoming XML Documents. The proposed solution, in a similar fashion to YFilter, employs a single Deterministic Finite Automatons (DFAs) approach to index the entirety of the query database at hand. Subsequently the constructed DFA is probed, during the filtering phase, to locate and evaluate the indexed queries against the incoming XML document. However, the main bottleneck when evaluating a DFA lies on the exponential growth of its size and the main memory requirements when constructing it under an "eager" approach. In order to alleviate this drawback the authors propose a novel approach that "lazily" calculates the DFA states. The latter approach differs the calculation of DFA states up until they are necessary. While on the other hand, the “eager” approach constantly updates the entirety of the DFAs states. As it is natural, the “lazy” DFA approach proposed reduces the memory requirements, when compared against the “eager” approach. Finally, the experimental evaluation demonstrated that this approach outperforms the XFilter algorithm by 4 orders of magnitude.

Chan et al. [60, 70] aim at providing efficient XML document filtering while enabling users with expressive query formulation through XPath queries. In the research conducted, the authors developed an indexing trie structure, coined XTrie, that supports complex XPath expressions, while the XTrie indexing structure is able to support matching of XML data in ordered and unordered formats. The key idea behind their approach lies in indexing parts (substrings) of the XPath queries during the indexing phase and thus reducing the search space during the filtering phase. The proposed research demonstrates high filtering throughput but its main drawback lies in the fact that the performance gains are heavily dependent on selecting the appropriate substring that is going to represent the query.

The S-ToPSS [6] system was among the first solutions that supported pub/-sub functionality in an ontological context. S-ToPSS was designed to enhance the matching process aiming at semantically similar but syntactically different information present in publications and user subscriptions. This was achieved by identifying synonyms and utilizing concept taxonomies and hierarchies. Additionally, S-ToPSS utilizes mapping functions to determine relations between attribute-value pairs. These methods can be applied separately or as a whole for better semantic matching results. The successor of S-ToPSS, G-ToPSS [8], focused on information dissemination of RDF data on ontologies, emphasizing on scalability and fast filtering of RDF data. G-ToPSS represented publications as directed labeled graphs, while a two-level hash table was used for the subscriptions. The system was employed to filter incoming publications in the form of RDF metadata (e.g. RSS). The matching algorithm was based on the traversal of the publication and subscription graphs.

In the same spirit, the Ontology-based Pub/Sub (OPS) system [7] supported events with complex data structures and aimed for a uniform representation. Subsequently, user subscriptions and publication events were processed into RDF graphs and thereafter indexed or filtered respectively. OPS utilized graph matching algorithms that traversed publication and subscription graphs. Finally, OPS examined the matching trees that emerged from the graph traversal to determine the matching subscriptions. The extension of OPS [71] focused on matching accuracy by utilizing concept models to enhance the publication filtering semantically.

Nguyen et al. [40] consider the problem of filtering incoming documents expressed in XML format. More specifically the authors develop a solution for discovering changes that occur in XML and HTML pages, which constantly evolve and are indexed by the Xyleme [72] system. To this end, the authors propose a subscription language that is aimed at capturing changes inside the XML and HTML documents. The users of the system can make use of the developed language and pose queries (subscriptions) that can capture the changes of the document database. Finally, an algorithmic solution that utilizes hash-trees is provided for processing the subscriptions against the incoming new or updated publications.

Although all these works focus on the problem of supporting pub/sub functionality in ontology systems, none has considered supporting any form of text extension. Park et al. [9] developed iBroker, an OWL-based pub/sub mediator focused in filtering publications from OWL ontologies against a set of stored SPARQL queries. iBroker matched incoming events generated from an ontology against user queries by resorting on an inverted index to represent the graph that indexed user subscriptions. Although there is no text support, iBroker is able to perform string matching using the inverted index mentioned above.

Additionally, the need for indexing and querying RDF data efficiently led to the development of RDF-3X [73]. RDF-3X engine focused on efficient triple indexing, presented a SPARQL execution engine coupled with a query optimizer that utilized statistics in order to recognize frequent paths. However, the aforementioned tools are applicable in an IR context as opposed to our approach that is designed for a pub/sub scenario.

Finally, during the last decade a variety of information push technologies that enhance IF services with content-based capabilities have emerged. The sparqlPuSH [74] system presented an architecture for monitoring update in RDF stores, whereas pushing notifications of real-time data changes through RSS/Atom feeds to subscribed users. The Sparkwave [75] system was built to perform continuous pattern matching over RDF streams by supporting expressive pattern definitions, sliding windows and schema-entailed knowledge. The C-SPARQL [17] extension enabled the registration of continuous SPARQL queries over RDF streams, thus, bridging data streams with knowledge bases and enabling stream reasoning. Moreover, works have focused on visualization techniques for ontology-based systems [76–79].

In the domain of ontology-based information filtering research has naturally focused on developing solutions that enhance the discovery of new information. However, there is an evident lack of a complete full-text filtering mechanism, beyond existing regular expression and equality support, with sophisticated algorithms and data structures to minimize processing and memory requirements. Providing such a full-text filtering mechanism over ontologies can potentially complement applications in knowledge bases, triple stores, and LOD platforms [1].

2.2.2 Decentralized Content-Based Information Filtering

With the advent of distributed and P2P computing, decentralized ontology pub/sub systems naturally emerged. The first P2P pub/sub system based on RDF data was build by Chirita et al. [80]; they provided language L, a language that aimed for both publication and subscription creation. The system utilized a super-peer architecture where super-peers were responsible for the routing of content determined by the RDF schema, property or value, while peers were responsible for specific schemas and properties. At publication time, super peers routed the data to the responsible

[1] http://lotus.lodlaundromat.org/

peers for the filtering process; the performance gain was achieved by utilizing the content similarities present in subscriptions.

Similarly, Liarou et al. [81] studied the problem of evaluating multi-predicate conjunctive queries in pub/sub systems; the aim of the system was to distribute the load of the matching process into a P2P network. The speed of the matching process was achieved by distributing efficiently the triples in the network nodes; where multiple peers were responsible for the indexing and matching of a single subscription while being organized in a chain-like manner.

In the same spirit, an RDF-based pub/sub P2P network was build by Pelegrino et al. [82] to study the messaging paradigm. The system supported the creation of queries by making use of SPARQL and publications by using RDF data. Users' subscriptions were indexed into a peer, determined by the CAN protocol. Data from a publication event that concerned a peer was stored while the event was forwarded to other peers. Although the filtering process became resource intensive faster notifications were achieved.

Kaoudi et al. [83] presented a study for distributed RDF reasoning and query answering. The work in [83] focused on implementing, optimizing, and evaluating forward and backward chaining over a distributed hash-table for a subset of SPARQL, managing to draw conclusions about performance and scalability over Internet-sized P2P networks.

Jacobsen et al. [84] proposed and developed PADRES, a distributed content-based publish/subscribe system that aimed at high availability and performance. The authors recognized and addressed three characteristics of an efficient content-based publish/subscribe system: (i) the routing technologies, (ii) the robustness of the network, and finally (iii) the management and deployment tools. To this end, PADRES leverages content-based routing techniques to alleviate restrictions posed by the topology of the network, coupled with a dynamic routing strategy that utilizes alternative routes to evenly balance traffic among the nodes of the network, similar to the approach in [85]. The subscription language employed by PADRES, utilizes simple predicates that express the users queries, advertisements and publications. An interesting feature of PADRES was the support of both historic and future data through employing partitioned databases. PADRES was designed for fault-tolerance, thus it collected information regarding the broker topology and the subscriptions routes in order to improve connectivity and decide on alternate routing paths, thus, enabling to select alternate routes when node failures occur. Furthermore, the proposed solution employed load estimation methodologies for load balancing of the network. Finally, the authors developed a set of tools for PADRES that included monitoring, deployment and administrative tools in order to ensure the apt operation of the distributed network. In the same spirit, [86] propose an architecture for scalable P2P pub/sub networks, while in [87] the authors study the deployment of a pub/sub system over cloud services. Furthermore, the work presented in [88] addresses the problem of increasing the efficiency of the pub/sub system and satisfiability of users. Finally, in [89] the authors study and analyse the underlying pub/sub mechanism of the commercial music platform Spotify.

2.2.3 Commercial Content-Based Information Filtering

There is a plethora of influential content-based systems that were developed in order to assist information extraction, discovery and management.

The most prominent example is the Apache Jena framework[2] that provides capabilities for indexing, managing, and querying RDF data sets. The Jena framework provides RDF APIs for creating and managing RDF graphs, as well as OWL APIs that enable semantic extension of the indexed RDF data. Apache Jena enables information access through its querying mechanism that supports the SPARQL query language, while it aims at providing high throughput through its scalable triple store.

An other influential system, is the Virtuoso server[3] a middleware and database engine that aims at providing a unified solution for traditional databases and triplestores. Notably the Virtuoso server supports RDF, XML, free-text and Relational database management system (RDBMS) under a single database engine.

[2] http://jena.apache.org/

[3] http://virtuoso.openlinksw.com/

In the context of content-based systems, Virtuoso supports the indexing and management of RDF and Linked data, while it provides users with SPARQL, XQuery and XPath quering capabilites.

Finally, two notable examples of content-based commercial systems is the AllegroGraph[4] and the OntoText GraphDB[5]. AllegroGraph is a triplestore, that supports the storage and manage of RDF triples, while OntoText GraphDB is a semantic graph database, that provides indexing, querying and visualization tools over RDF data.

2.3 Graph-Based Systems

In recent years, graphs have emerged as prevalent data structures to model information networks in several domains such as social networks, knowledge bases, communication networks, biological networks and the World Wide Web. These graphs are *massive* in scale and *evolve* constantly due to frequent updates. For example, Facebook has over 1.49 billion daily active users who generate over 500K posts/comments every 60 seconds and 4 million likes every minute resulting in massive updates to the Facebook social graph[6].

To gain meaningful and up-to-date insights in such frequently updated graphs, it is essential to be able to monitor and detect continuous patterns of interest. There are several applications from a variety of domains that may benefit from such monitoring. In social networks, such applications may involve targeted advertising, spam detection [10, 11], and fake news propagation monitoring based on specific patterns [12, 13]. Similarly, other applications like (i) protein interaction patterns in biological networks [14, 15], (ii) traffic monitoring in transportation networks, (iii) attack detection (e.g., distributed denial of service attacks in computer networks), (iv) question answering in knowledge graphs [17], and (v) reasoning over RDF graphs [90] may also benefit from such pattern detection.

[4] http://franz.com/agraph/allegrograph/

[5] http://ontotext.com/products/graphdb/

[6] Facebook quarterly update http://bit.ly/2BIM30d

In the following section, we present works that have focused on graph matching, be it applied on the information retrieval paradigm or the information filtering perspective.

Structural graph pattern search using graph isomorphism has been studied in the literature before [18, 19]. In [91], the authors propose a solution that aims at reducing the search space for a single query graph. The solution identifies candidate regions in the graph that can contain query embeddings, while it is coupled with a neighborhood equivalence locating strategy to generate enumerations. In the same spirit [92] aims at reducing the search space in the graph by exploiting syntactic similarities present on vertex relationships. In [93], the authors consider the sub-graph isomorphism problem when multiple queries are answered simultaneously. However, these techniques are designed for static graphs and are not suitable for processing continuous graph queries on evolving graphs.

The problem of continuous sub-graph matching has been considered in [94] but the authors (i) assume a static set of sub-graphs to be matched against update events, (ii) use approximate methods that generate false positives, and (iii) apply the solutions on small (evolving) graphs. An extension to this work considers the problem of uncertain graph streams [95], targeting applications like wireless sensor networks and protein-protein interactions. These solutions are not suitable for answering large number of continuous queries on graphs with high update rates. Moreover, to address the difficulty of sub-graph isomorphism they resort to approximate matching, which results in undesirable false positives. An extension to this work considers the problem of uncertain graph streams [95], targeting applications like wireless sensor networks and protein-protein interactions.

There are a few publish/subscribe solutions on ontology graphs proposed in [7, 8], but they are limited to the RDF graphs and RDF subscriptions. Distributed pub/sub middleware for graphs has recently been proposed in [96], but the authors do not consider graph structure (they limit subscriptions to node attributes and node distance constraints). Finally, in [97] the problem of evaluating graph constraints between publishers and subscribers is presented and applied to a distributed Web advertising scenario.

Another relevant area of research is graph streams; in [98, 99], algorithms to identify correlated graphs from a graph stream are proposed. This differs from our setup since (i) a sliding window that covers a number of consecutive batches of stream data records is used and (ii) the authors target at identifying subgraphs with Pearson correlation coefficients higher than a given threshold. In [100], the authors propose a continuous pattern detection in graph streams with snapshot isolation. However, this solution considers only single queries at a time and the patterns detected are also approximate, without considering the existing graph.

The work in [101] provides an exact subgraph search algorithm that exploits the *temporal* characteristics of representative queries for online news or social media monitoring. This algorithm exploits the structural and semantic characteristics of the graph through a specialized data structure coined the Subgraph Join Tree. While this is similar to the pub/sub scenario, the emphasis is on efficient search mechanisms for time-stamped events, rather than filtering of streaming graph data. An extension of this work, considers continuous query answering with graph patterns over dynamic multi-relation graphs [102]. In this work a query is decomposed into smaller sub-graphs by the Subgraph Join Tree data structure, while the selectivity distribution of the subgraphs is taken into account when generating the Join Tree. Subsequently, by utilizing the selectivity of each subgraph, the data graph is searched in order to determine if the query graph has been satisfied. This approach varies from our setting, as it aims at efficiently answering each query individually, while the criteria of selectivity is utilized to develop a enhance the search mechanism to explore the data graph.

Finally, in [20] the authors try to perform subgraph matching over a billion node graph by proposing graph exploration methods in the distributed memory cloud of Trinity [103]. The proposed methodologies for processing the query does not require an indexing structure, while the graph exploration techniques allow for minimal joining operations. The aforementioned works although they provide efficient work while the approach falls under the category of graph exploration and not the pub/sub nor the streaming category.

While the aforementioned works are similar to the query evaluation scenario, the emphasis is on efficient search mechanisms, rather than continuous answering over streaming graph data. Sub-graph properties such as clustering coefficient and density have been considered with respect to evolving graphs before (e.g., top-k densest sub-graph maintenance [104] and dynamic community detection based on clustering coefficient [105]). Finally, there are works in graph evolutionary network analysis [106], but in pub/sub the focus is not on maintenance/ analysis of the (evolving) graph.

Gillani et al. [107], propose a solution for continuous graph pattern matching over knowledge graph streams is presented. The proposed data model contains continuous subgraph patterns that pose structural constraints into evolving graphs, while the graph model is based on knowledge graph events, i.e. sets of triples published in knowledge bases. The proposed solution utilizes finite automatons to represent the continuous query graph patterns and to perform the filtering process. However, [107] uses a sliding window approach that misses matching events, while each continuous query graph pattern is evaluated separately. To the best of our knowledge none of the existing works exploit the common sub-graph patterns to scale up to thousands of simultaneous continuous queries. The automatons generated for each query graph pattern are utilized during the publication filtering time in order to determine which queries have matched and generate the subgraphs that have satisfied the patterns. Additionally, the algorithm implements a join-and-explore approach to answer the queries graphs. The proposed solution supports event-based evaluation, as well as, incremental evaluation of the query graph patterns. Similarly to previously discussed approaches each query graph is answered separately. More specifically, a query plan (i.e. an automaton) is generated for each query graph pattern and executed separately during publication filtering. Thus, the commonalities of the query graph patterns are not taken into account and are not explored. As a result the number of continuous graph patterns answered simultaneously is extremely low. Although the solution is dependent on the window size when incrementally evaluating a query graph pattern results are missed when the window overflows.

In the domain of graphs, there is an evident lack of solutions that aim at continuously monitoring graph streams and locate, in a timely-fashion, patterns of interest that form during the evolution of the graph. In the literature, research has focused on providing solutions under the assumption of static graphs [91–93], or when considering the simultaneous detection of a few distinct patterns [103, 107]. To this end, it is necessary to develop a solution that efficiently matches multiple continuous patterns over (one or many) streams of graph updates and appropriately notify the subscribed users for any patterns that match their subscription. In this work we focus on providing efficient algorithms that can capture patterns in large evolving graphs.

2.4 Conclusions

This chapter surveys related literature under three distinct research domains: (i) textual information filtering, (ii) content-based information filtering, and (iii) graph-based multi-query answering. In the following chapters, we present our solutions and contributions to each problem described in Section 1.1.

3
Efficient Continuous Multi-Query Processing over Textual Data

In the previous chapters we described the paradigm of multi-query answering in the context of information filtering and presented a variety of its application scenarios. Additionally, we highlighted two important aspects of the information filtering problem, commonly studied in the literature, its efficiency and its effectiveness. In this chapter, we focus on the case of *textual information filtering*, where we identify the importance of *efficient information filtering*. To this end, we present a novel algorithmic solution developed for efficient query indexing and reorganization. We identify different reorganization strategies applied on the query database and demonstrate their effect on the performance efficiency factor. We demonstrate that, contrary to previous literature works, the nature of the constructed tries, rather than their compactness, is the determining optimization factor. Finally, we examine the parallelization process of the information filtering process, identify different parallelization scenarios and present our findings and conclusions. The results of this chapter have been published in [108].

The rest of this chapter is formed of six distinct sections. Section 3.1 provides the motivation for this work, as well as, the description of the research problem addressed. Section 3.2 presents the data model under which we choose to solve the problem. Section 3.3 presents our proposed solution, Algorithm StaR and discusses possible extensions. Section 3.4 presents state-of-the-art algorithms in the literature that were utilized as competitors in our experimental evaluation scenarios. Section 3.6 presents our experimental evaluation and comparison against existing approaches. Finally, Section 3.7 concludes this chapter by discussing the results, provides the outlook, and presents future directions of this research.

3.1 Motivation

In recent years, *information filtering* applications (also known as *information dissemination* or *publish/subscribe*), such as news alerts, weather monitoring, and stock quotes, have gained popularity. Such applications assist users to cope with the information avalanche and the cognitive overload associated with it. For the case of news alerts, digital libraries, or RSS feeds, where the data of interest is mostly textual, users express their needs using information retrieval languages (e.g., Boolean combinations of keywords [1] or text excerpts under the Vector Space Model – VSM [1]) and submit *continuous queries* (or *profiles*) to a server, thus, *subscribing* to newly appearing documents that will satisfy the query conditions. The server will then be responsible for *notifying* the subscribed users *automatically* whenever a new document that matches their information needs is published. Publishers can be news feeds, digital libraries, or even users who post new items to blogs, social media, and Internet communities. This functionality is very different from information retrieval applications like search engines [1]. Specifically, in information retrieval when a query is posed, a single search is executed and the current matching data items are presented to the user. Contrary, in information filtering the server indexes the *user queries* rather than the data and evaluates newly published data items against the stored continuous queries.

In more detail, the problem of information filtering may be defined as follows:

Definition 3.1 ***Problem Definition***
Given a database DB of continuous text-based queries that reside on a server and an incoming document d, retrieve *all queries $q \in DB$ that match d.*

The filtering problem is of high importance and needs to be solved efficiently, since servers are expected to handle millions of user queries and high rates of incoming published documents. Efficiency issues were identified by many researchers that proposed tree and trie-based algorithms for supporting fast filtering under various data models (e.g., flat attribute-based, semi-structured XML) and query languages (e.g., Boolean, VSM), both for main-memory [2–4] and secondary storage [5]. However, all these approaches use a greedy clustering method that is sensitive to the insertion order of submitted queries and do not consider that an evolving query workload might require the reorganization of the query database to achieve efficient filtering performance.

In our work, we concentrate on *textual IF* and present a novel *trie-based*, *main-memory* algorithm for *Boolean IF* that is able to match incoming documents against millions of queries in a few milliseconds. Our method uses linguistically motivated concepts, such as words, to support continuous queries that are comprised of *conjunctions of keywords* and may be used as a basis for query languages that support not only basic Boolean operators, but also more complex constructs, such as proximity operators and attributes. We believe that offering an efficient Boolean filtering service (possibly alongside a more popular model like VSM) is a valuable addition to any text filtering setup. Boolean IR/IF is still the model of choice of advanced users that want total control of their results and is widely supported in systems of major stakeholders like Google's advanced search/alert mechanisms, Oracle's text extender module, and in Apache's text search engine. Such systems, that are meant to cope with a high workload and are designed for efficiency, are possible applications for our work.

The algorithm we developed, coined StaR (an acronym derived from STAtistical Reorganization), is the first solution in the literature to consider *database reorganization* (through appropriate word/query statistics) to achieve efficient textual IF under the Boolean model. The main idea behind the proposed algorithm is to use tries to capture common elements of queries, similarly to [3–5]. However, the key differences with these approaches lie in *(i)* the collection and utilization of statistics on the importance of keywords in the indexed queries, *(ii)* the reorganization of the query database according both to *word* and *query importance*, and *(iii)* the demonstration that the nature of the trie forest is more important than its compactness when it comes to filtering efficiency. Interestingly, all previous works [3–5] were aiming at *minimizing the size* of the trie forest, since there was an implicit conjecture that a small forest would result in lower filtering times due to less node visits. Our findings demonstrate that forest size is not the dominating optimization factor when it comes to filtering efficiency; contrary, the focus should be put on the nature of the tries and on qualitative characteristics (expressed through heuristics). To this end, Algorithm StaR overcomes the query insertion order problem caused by the greedy query clustering techniques adopted by all other algorithmic solutions [2–5].

In the light of the above, our contributions may be summarized as follows:

- We present a *novel* query indexing and reorganization algorithm that supports Boolean IF up to 96% faster than state-of-the-art competitors. The query reorganization process is highly efficient and takes less than a few seconds on a commodity PC to reorganize, as many as, 500K queries in a database comprised of millions of queries.

- We identify different *reorganization options* for the trie indexes and demonstrate the importance of query insertion order in the construction of the indexing structure. We also show that constructing tries with *rare* words at the higher level of the trie leads to improved filtering performance due to early pruning at filtering time.

- We demonstrate, contrary to previous works, that the *nature* of the constructed tries, rather than their *compactness*, is the determining factor for efficient filtering performance, especially in datasets with rare clustering opportunities.

- We experimentally evaluate different reorganization strategies and showcase their effect in filtering efficiency using two different *real-world datasets* and both *synthetic* and *real* query sets.

- We extend the presented algorithm implementation by parallelizing the filtering process to suit modern multi-core processors. We identify two different parallelization options and experimentally evaluate their performance.

In this section, we presented the motivation behind our work and gave a brief overview of the algorithmic solutions we developed in order to solve the efficiency problem of information filtering. In the following section, we present the data model we employed in our approach.

3.2 Data Model

The Boolean model is still widely adopted in Information Retrieval and Information Filtering applications, as it is the model of choice for advanced users. In the Boolean model, users can pose queries (or profiles) comprised of Boolean expressions. In text-based IR and IF systems, users utilize terms coupled with Boolean operators (*AND*, *OR* and *NOT*) to formulate their queries and thus express their information needs. In this paradigm, each document is represented as a set of words [1]. To this end, the users can capitalize on the Boolean Model and achieve high control of the results delivered to them. The Boolean model, is widely supported in commercial systems developed by major stakeholders, like Google's *Advanced Search* [109] and *Alerts* [25] mechanisms; Oracle's text extender module [110]; and in Apache's Lucene [111] and Solr [112] text search engines. Such systems, that are meant to cope with a high workload and are designed for efficiency, are possible applications for our work.

The Query Model. In our query model we support two distinct categories of *keywords*, *simple textual terms* and *complex terms*. Simple textual terms are single words utilized to pose constraints on incoming publications, while complex textual terms (often referred as phrases) are ordered sets of terms that explicitly define the order of the terms that should appear in an incoming publication. To this end In our approach, we choose to support user queries that are formulated by simple textual terms and complex terms with conjunction operators. To this end, a user query q_i is formally defined as follows:

$$q_i = kwrd_1 \wedge kwrd_2 \wedge \cdots \wedge kwrd_n$$

The Publication Model. In the context of our approach, an incoming publication p_i is a document comprised of simple words. Thus, a publication is represented as an ordered set of simple terms. To this end, an incoming publication p_i is formally defined as follows:

$$p_i = \{term_1, term_2, \ldots, term_m\}$$

In our information filtering model, we utilize simple terms to describe the data publications, while the information filtering algorithms, presented in the following section, can match an incoming publication expressed in simple terms, against a query database.

In this section, we described the data model we adopted in our approach. In the following sections we describe the algorithmic solutions developed to solve the information filtering problem. Finally, in order to accommodate the interested user, we provide an effectiveness comparison of the Boolean model compared against the VSM model later in Section 3.6.7.

3.3 The Algorithm StaR

In this section, we describe the algorithmic solution developed in order to efficiently solve the information filtering problem. We consider queries that comprise of conjunctions (i.e., sets) of keywords, as described in Section 3.2, and present Algorithm StaR that relies on the reorganization of the query database to achieve low filtering times. Subsequently, we outline existing trie-based solutions and discuss extensions of our algorithm to support more expressive queries.

The key idea of Algorithm StaR is to collect statistics on the importance of keywords in the indexed queries and reorganise the query database according both to keyword and query importance. This results in *four variations* of the algorithm that are described later in this section. Algorithm StaR operates in two phases:

Query Identifier	Query Terms
q_1	{olympic, games}
q_2	{olympic, games, rio}
q_3	{olympic}
q_4	{olympic, rio}
q_5	{olympic, committee}
q_6	{olympic, committee, president}
q_7	{olympic, rio, stadium}
q_8	{olympic, congress, rio}
q_9	{euro, cup, france, paris}
q_{10}	{olympic, committee}

Table 3.1: Set of conjunctive user queries.

- The *indexing phase*, where the initial query indexing takes place, presented in Section 3.3.1. The Pseudocode for the indexing phase of Algorithm StaR is performed by Algorithm Index and is presented in Figure 3.2.

- The *reorganization phase*, where *only the newly indexed* queries in the database are reorganized by utilizing the *statistics* collected during the indexing phase, presented in Section 3.3.2.

Note that during the indexing phase new queries are stored only temporarily waiting for the reorganization phase; thus, no statistical information is used during the initial query placement (indexing phase) as the final placement of queries based on query statistics will be decided later (reorganization phase). For scalability reasons, Algorithm StaR does not reorganize the complete query database, but, only a tunable amount of newly indexed queries.

3.3.1 Query Indexing Phase

Initially, we will consider queries that comprise of conjunctions (i.e., sets) of keywords like the ones presented in Table 3.1. Subsequently in Section 3.5, we will also consider more expressive query languages. To index queries, Algorithm StaR

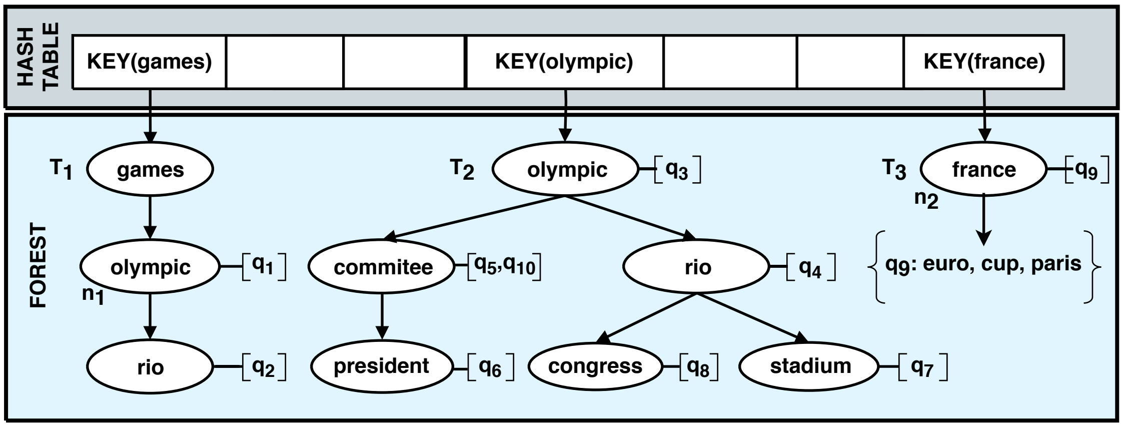


Figure 3.1: The Forest data structure during the indexing phase of Algorithm StaR.

uses two data structures: a *forest of tries* that organizes the keywords of queries and a *hash table* that provides efficient access to the roots of the tries in the forest. For instance, the queries of Table 3.1 are organized in the trie structures of Figure 3.1. Each trie node n:

- Stores a keyword of a query, denoted by $\mathit{kwrd}(n)$.
- If the keywords in a path from the root to node n spell out a query q then n also stores a reference to q. The list of all references stored in node n is denoted by $\mathit{id}(n)$.
- If n is a leaf node then n also stores one list for each query q, denoted by $\mathit{uexp}(n, q)$, containing the keywords of query q that are not already included in the path from the root to n.

For instance consider Figure 3.1, where $\mathit{kwrd}(n_1) = \textsf{olympic}$ and $\mathit{id}(n_1) = [q_1]$, i.e. node n_1 stores also a reference to query q_1. Consider also node n_2 of trie T_3 that stores query $q_9 = \{\textsf{euro, cup, france, paris}\}$. Notice that $\mathit{kwrd}(n_2) = \{\textsf{france}\}$, and that $\mathit{uexp}(n_2, q_9) = \{\textsf{euro, cup, paris}\}$. Finally, note that n_2 contains all keywords of q_9 (since $q_9 = \mathit{kwrd}(n_2) \cup \mathit{uexp}(n_2, q_9)$), thus, it also maintains a reference to q_9.

The purpose of list $\mathit{uexp}(n, q)$ is to allow for the delayed creation of nodes in a trie; this allows us to choose which keywords from the $\mathit{uexp}(n, q)$ list will become the child of current node n depending on the queries that will arrive (and be indexed in this trie) later on. Note that the intersection of all *uexp* lists stored at a node n is the empty set, since if there was a common keyword among them it would have been expanded to a new node. Additionally, for all *uexp* lists $|uexp(n, q)| > 1$ holds, i.e., lists with exactly one keyword are automatically expanded to trie nodes.

The forest of tries is populated in order to store queries compactly by exploiting their *common keywords*. When a new query q arrives, Algorithm StaR considers its keywords and inserts them in a (new or existing) trie in the forest. For this task, StaR selects the best trie T in the forest and the best node n in that trie to insert q (the insertion process is described later in the section). To this end, StaR uses the concept of *node reusability*, denoted by $nr(q, T)$, that quantifies the percentage of q's keywords that are stored in a path starting from the root of T and also used by other queries. More formally, $nr(q, T) = \frac{|path|}{|q|}$, where $|path|$ is the size of the longest path from the root of trie T that contains only keywords of q participating to other queries and $|q|$ is the number of keywords in q. It follows that $0 \leq nr(q, T) \leq 1$, and generally when $nr(q, T)$ is close to 0, trie T is considered as a poor candidate for q because only a small fraction of terms in q will be stored in existing nodes of T. Contrary when $nr(q, T)$ is close to 1, trie T is considered as a good candidate, because a large fraction of terms in q will be stored in existing nodes of T. Node reusability extends the clustering ratio concept [4] with the constraint that keywords should be present in other queries and promotes the frequent or rare keywords towards trie roots, depending on the configuration of Algorithm StaR (discussed in Section 3.3.2).

Example 3.1 *Let us consider the queries and their organization illustrated in Figure 3.1. We have $nr(q_1, T_1) = \frac{2}{2}$, since the 2 keywords of q_1 are both stored in a path starting from the root of T_1 and also used in a different query (i.e., q_2). We also have $nr(q_2, T_1) = \frac{2}{3}$, since only 2 keywords of q_2 (out of 3) are stored in a path starting form the root of T_1 and also used in a different query (i.e., q_1), as keyword* rio *is used solely for q_2. Similarly, $nr(q_3, T_2) = \frac{1}{1}$, $nr(q_4, T_2) = \frac{2}{2}$, $nr(q_5, T_2) = \frac{2}{2}$, $nr(q_6, T_2) = \frac{2}{3}$, $nr(q_7, T_2) = \frac{2}{3}$, $nr(q_8, T_2) = \frac{2}{3}$, $nr(q_9, T_3) = \frac{0}{4}$, and $nr(q_{10}, T_2) = \frac{2}{2}$.*

```
Algorithm: Index
Input: A query $q = \{k_1, \ldots, k_t\}$
Result: Store $q$ in Forest

$currentNR \leftarrow 0$;
$position \leftarrow$ Null;

// For all candidate tries $T$
foreach $trie\ T\ with\ root(T) = k \in q$ do
    // DFS traversal for all possible storage positions
    foreach $node\ n \in T\ such\ as\ kwd(n) \in q$ do
        calculate $(nr(q, T))$;
        // If a better position is found store it
        if $currentNR < nr(q, T)$ then
            $currentNR \leftarrow nr(q, T)$;
            $position \leftarrow n$;

// If $q$ cannot be indexed in any existing trie
if $position = Null$ then
    create trie $T'$ with $root(T')$ such as $kwrd(T') \in q$;
    // Index $q$ in $root(T')$
    $id(root(T')) \leftarrow q$;
    // Put the rest in $uexp(T', q)$
    $uexp(T', q) \leftarrow q \setminus kwrd(T')$;
else
    // If there are no common keywords
    if $uexp(position, p) \cap q = \emptyset$ then
        // Index $q$ in $position$
        $id(position) \leftarrow id(position) \cup q$;
        // Put the rest in $uexp(position, q)$
        $uexp(position, q) \leftarrow q \setminus \{k_1, ..., k_y\}$;
    else // Else expand the common keywords
        expand $uexp(position, p) \cap q \setminus K$;
        // Index $q$ and $p$ at the leaf node
        $id(m) \leftarrow q \cup p$;
        // Remove $p$ from $id(position)$
        $id(position) \leftarrow id(position) \setminus p$;
        // Put the rest in two new uexp lists
        $uexp(m, q) \leftarrow q \setminus \{k_1, ..., k_x\}$;
        $uexp(m, p) \leftarrow p \setminus \{k_1, ..., k_x\}$;

// Gather statistics for query reorganisation
gatherStats($q$);
```

Figure 3.2: Pseudocode for the query indexing phase (Algorithm Index) performed by Algorithm StaR.

The algorithm for inserting a new query proceeds as follows. The first query that arrives, creates a trie with a randomly chosen keyword as the root; the remaining keywords are stored at the *uexp* list of the root (Figure 3.2, lines $10 - 12$). The second query will consider being stored at the existing trie or create a new trie (Figure 3.2, lines $9 - 21$). In general, to insert a new query q, StaR iterates through its keywords and utilizes the hash table to find all *candidate tries*; i.e., tries having a root storing a keyword of q (Figure 3.2, lines $3 - 8$). To compactly store q, StaR then chooses the trie T among the candidates for which q insertion maximizes $nr(q, T)$ (Figure 3.2, lines $5 - 8$). To compute $nr(q, T)$, StaR performs a depth-limited search with depth limit $|q| - 1$ in *all* candidate tries. This search finds node n in T where q should be inserted (Figure 3.2, lines $4 - 8$). Note that the chosen path from the root to n is the longest path in T that exclusively contains keywords of q. If more than one tries maximize $nr(q, T)$, StaR randomly chooses one.

To complete insertion, the path from the root of trie T to node n, that already stores the identifier of a query p and the set of keywords K, is then extended with new child nodes having as keywords the intersection of $uexp(n, p)$ and $q \setminus K$ (Figure 3.2, line 18). If all keywords in q are contained in $K \cup uexp(n, p)$ then *(a)* the keywords in $q \setminus K$ are expanded to trie nodes to create a path from node n to a trie node m (Figure 3.2, line 19), *(b)* node m becomes a new leaf in trie T, *(c)* $id(m)$ will contain the reference to query p (previously stored in $id(n)$) plus a reference to q (Figure 3.2, line 20), and *(d)* reference to p is removed from $id(n)$. In this way, list $uexp(n, p)$ is fully expanded to trie nodes, query q is indexed in this subtrie under all its keywords, and node m now indexes two query identifiers, namely q and p. Otherwise, if some keywords of q are not contained in $K \cup uexp(n, p)$, then the common keywords are expanded to trie nodes to create a path from node n to node m, and node m will store two new *uexp* lists, namely $uexp(m, p)$ and $uexp(m, q)$ (Figure 3.2, line 21). Additionally, $id(m)$ will contain references to both p and q, while p is removed from $id(n)$. Notice that $uexp(m, p)$ will contain the remaining set of keywords of $K \cup uexp(n, p)$ that are not contained in q and $uexp(m, q)$ will contain the remaining set of keywords of q that are not contained in $K \cup uexp(n, p)$.

Keyword k	$sprt(k)$	Keyword k	$sprt(k)$
olympic	9	president	1
games	2	euro	1
rio	4	cup	1
committee	3	france	1
stadium	1	paris	1
congress	1		

Table 3.2: Statistics of keywords gathered from the queries in Figure 3.1.

Also due to this node expansion process, $uexp(m, p) \cap uexp(m, q) = \emptyset$. Finally, if no keywords of q are contained in $uexp(n, p)$, then a new $uexp(n, q)$ list is created in node n and a reference to q is added in $id(n)$. The complete pseudocode for the query indexing phase of Algorithms StaR is presented in Figure 3.2.

Example 3.2 *Figure 3.1(b) shows the forest of tries created when inserting the queries $q_1, \ldots, q_{10}$ (shown in Figure 3.1) in that order. The first query q_1 creates trie T_1 and is indexed under the (randomly chosen) keyword* games*. The second query q_2 does not create a new trie, but, is indexed under T_1, since this maximizes its node reusability $nr(q_2, T_1)$. The third query q_3 cannot be indexed in T_1, since it does not contain the keyword* games*, thus, a new trie T_2 is created and q_3 is indexed under the keyword* olympic*. Similarly,* StaR *inserts the remaining queries.*

The time complexity of Algorithm StaR when indexing a new query q with t distinct words is $\mathcal{O}(t^t)$, since StaR uses a depth-first search strategy (with the maximum depth bound by the number of distinct query words) and visits only sub-tries that have one of the query words as root.

Finally, during the indexing phase, Algorithm StaR collects statistics about the frequency of occurrence of keywords in queries, which are then utilized in the reorganization phase (described in the following section). Table 3.2 presents the statistics collected by Algorithm StaR.

3.3.2 Query Reorganization Phase

Reorganization is a periodic procedure that initiates at given time intervals, after a given number of query insertions, or when a criterion is met (e.g., when a certain percentage of queries have low node utilization). Any of the above options may be implemented in the context of Algorithm StaR; for simplicity we have selected to reorganize the query database after the insertion of Q new queries. It should be noted that, contrary to Algorithm ReTrie [4] which relocates only poorly indexed queries, StaR reorganizes only those queries inserted since the last database reorganization.

To reorganize queries, Algorithm StaR utilizes a scoring mechanism to modify the order of insertion of queries in the database and to favor the indexing of queries under frequent or infrequent keywords in the tries. It utilizes the *support* of a keyword k (denoted by $sprt(k)$), which represents the number of queries in the forest that contain the keyword k, to identify the frequent and infrequent keywords among the queries indexed in the forest. Using the support of its keywords, we define the *score* of a query $q = \{k_1, \ldots, k_t\}$, denoted by $score(q)$, as $score(q) = \sum_{i=1}^{t} sprt(k_i)$. As we show later on, the score of a query plays an important role to the reorganization phase. Note that we do not normalize $score(q)$, as the size of a query q plays an important role in the reorganization phase since it affects trie construction.

Example 3.3 *Let us consider the queries and the index of Figure 3.1, and assume that Algorithm* StaR *collected the frequencies illustrated in Table 3.2. According to these frequencies, we have* $score(q_1) = sprt(\textsf{olympic}) + sprt(\textsf{games}) = 9 + 2 = 11$. *Table 3.3 presents the total scores of queries* $q_1, ..., q_{10}$ *as calculated by Algorithm* StaR.

Query Identifier	$score(q_i)$	Query Identifier	$score(q_i)$
q_1	11	q_6	13
q_2	15	q_7	14
q_3	9	q_8	14
q_4	13	q_9	4
q_5	12	q_{10}	12

Table 3.3: Query scores as calculated by Algorithm StaR.

To maintain *sprt* (resp. *score*), Algorithm StaR utilizes a hash table, denoted by *statSprt* (resp. *statSco*), that contains keywords k (resp. queries q) as keys and support of keywords $sprt(k)$ (resp. scores of queries $score(q)$) as values (similarly to Tables 3.2 and 3.3). StaR employs these statistics to reorganize newly inserted queries in the query database as follows. StaR re-indexes the newly inserted queries $\{q_1, \ldots, q_s\}$ from the existing forest by sorting them in descending order according to $score(q_i)$, where $score(q_1) \geq score(q_i) \geq score(q_s)$. Thus, queries with the highest score are inserted first in the forest; this variation of StaR is identified as StaR-H. Respectively, Algorithm StaR could re-index all the newly inserted queries by sorting them in ascending order according to $score(q_i)$, where $score(q_1) \leq score(q_i) \leq score(q_s)$. Thus, queries with the lowest score are inserted first in the forest; this variation of StaR is identified as StaR-L. According to StaR-H, the new order of insertion will be $q_2, q_7, q_8, q_4, q_6, q_5, q_{10}, q_1, q_3, q_9$, while StaR-L uses the inverse order. As we will show in Section 3.6, the problem of query insertion order is important; the first queries to be indexed define the *clustering opportunities* for the subsequent ones.

Apart from defining the insertion order of queries, StaR also utilizes the support of a keyword *sprt* to influence the construction of tries in the following way. The query insertion algorithm described in the previous section is modified so that when a query $q = \{k_1, \ldots, k_t\}$ is indexed in a *new trie* T' (because there is no other trie having a root with a keyword in $\{k_1, \ldots, k_t\}$), the most frequent keyword in q is chosen as the root of T', while the rest of the keywords remain in the *uexp* list. Additionally, when a query q is indexed under a node n of trie T because it maximizes its node reusability $nr(q, T)$, the path from the root to n is extended with nodes containing the most frequent keyword from the $uexp(n, q)$ list. In this way, StaR creates tries that index the most frequent keywords near the roots, while the rare keywords are pushed deeper in the trie. It is important to note that *node reusability* is still the criterion for deciding where to index a query, while keyword support is used to solve ties between equally good (or poor) positions in existing tries and to enforce roots of new tries. This indexing scheme creates

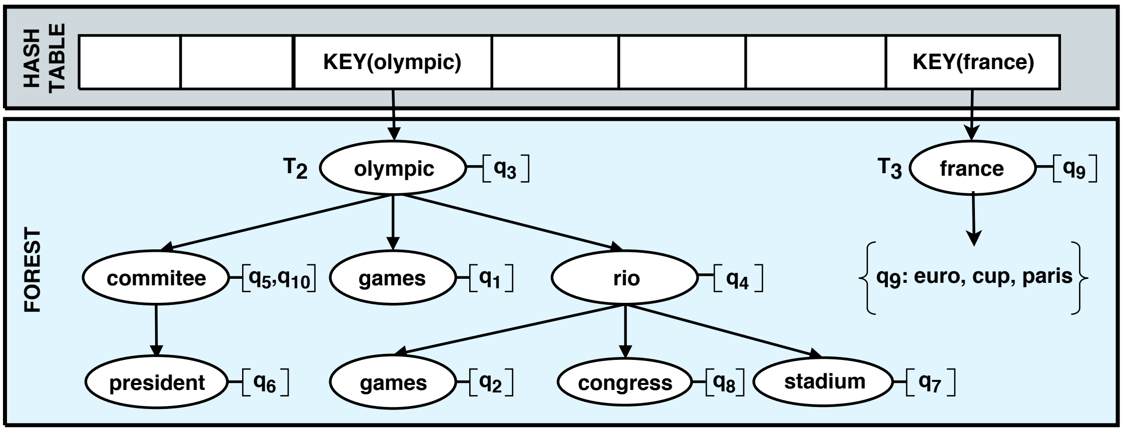


Figure 3.3: The Forest data structure after the reorganization phase of Algorithm StaR-LR for the queries of Figure 3.1.

a new variation for Algorithm StaR, identified as StaR-F. In the same spirit, Algorithm StaR is modified to influence the insertion of query q based on its most rare keywords. In this case, the most rare keyword of a query q is chosen as the root of a *new trie* T' and new paths with nodes containing the most rare keywords from the *uexp* lists are created. The last variation of Algorithm StaR, is identified as StaR-R. As we will show in Section 3.6, the frequency of keywords plays an important role in filtering time.

Words near the roots	Query score and insertion order	
	High-to-low	Low-to-high
Frequent	StaR-HF	StaR-LF
Rare	StaR-HR	StaR-LR

Table 3.4: Variations for Algorithm StaR

The above options that define the indexing order of the queries and influence the construction of tries by using keyword frequency, provide four distinct variations for Algorithm StaR, identified as StaR-HF, StaR-HR, StaR-LF, and StaR-LR. Each variation has its own characteristics, which are summarized in Table 3.4, and requires a different parameter setting. All these options, along with the filtering performance of each variation are discussed in Section 3.6.

Figure 3.3 shows the Forest data structure after the reorganization phase of Algorithm StaR-LR is executed. Notice that compared to the previous forest (shown in Figure 3.1), all queries are now indexed under two tries and utilise one trie node less (10 trie nodes in the initial forest vs. 9 trie nodes in the reorganized one). Finally, notice that node reusability of q_1 is reduced from $nr(q_1, T_1) = \frac{2}{2}$ (Figure 3.1) to $nr(q_1, T_2) = \frac{1}{2}$ (Figure 3.3), while it remained the same for the rest of the queries. As we will demonstrate in Section 3.6, the most important factor for the filtering performance of the algorithms is the nature of the created forest and the way it is constructed, rather than the trie compactness and the high node reusability values.

The time complexity of Algorithm StaR, when reorganizing a set of newly indexed queries Q with at most t distinct words each, is bound by $\mathcal{O}(QlogQ + Qt^t)$, since StaR has to sort the queries according to their score and reinsert them in the trie forest.

3.3.3 Filtering Incoming Documents

When a document d is published, the filtering procedure for Algorithm StaR is performed by Algorithm Filter illustrated in Figure 3.4. For each *distinct* keyword k_j of d maintained in a linked list created at the preprocessing step of d (Figure 3.4, line 2), the trie of Forest that has keyword k_j as root is traversed in a depth-first search manner (Figure 3.4, line 3). Notice that only subtrees having as root the keyword k_j contained in document d are examined (since only these may contain potentially matching queries), and a hash table (also created at the preprocessing step of d) that indexes all distinct keywords of d is used to identify them quickly. At each node n of a trie, the $id(n)$ list gives implicitly all queries that match the incoming document d (Figure 3.4, lines $5 - 8$). To identify all qualifying queries, this procedure is repeated for all the keywords of d.

The time complexity of filtering for Algorithm StaR is $\mathcal{O}(t^t)$ for a document d with t distinct words, since StaR uses a depth-first search strategy (with the maximum depth bound by the number of distinct words in the document) and visits only sub-tries that have one of the document words as root. For this traversal,

```
Algorithm: FILTER
Input: A document $d = \{k_1, \ldots, k_t\}$
Output: A list of queries $match = \{q_i, \ldots, q_j\}$
$match \leftarrow$ Null;
// Use a linked list for distinct keywords of $d$
foreach distinct keyword $k \in d$ do
    foreach trie $T$ with $root(T) = k \in d$ do
        foreach node $n \in T$ do
            // Use a hash table representation of $d$ to check this
            if $kwrd(n) \in d$ then
                if $uexp(n, q) \subseteq d$ then
                    // The queries stored here match $d$
                    $match \leftarrow match \cup id(n)$;
                    // Traverse trie in DFS
                    $n \leftarrow children(n)$;
            else
                // Else do not search in sub-tries
                prune $n$;
// Return list of matched queries
return $match$;
```

Figure 3.4: Pseudocode for the document filtering phase (Algorithm FILTER) performed by Algorithm STAR.

STAR performs $\mathcal{O}(t^t)$ probes to the hash table representation of document d; this leads to an overall filtering time complexity of $\mathcal{O}(t^t)$.

In a pub/sub system the publication events are more frequent compared to the subscription events, i.e., the information flow is constantly high, contrary to subscriptions that are updated at a lower rate. Additionally, the filtering procedure is a process that does not affect the structure of the FOREST and is executed in a serialized manner. In the following section, we present and investigate two parallelization variations of STAR that speedup filtering.

3.3.4 Parallelization of the Filtering Process

An elegant way of enhancing the performance of Algorithm STAR is by parallelizing the filtering process. Such an improvement is critical as filtering algorithms are expected to process high volumes of incoming information as efficiently as possible. Here we identify two proof-of-concept parallelization variations of Algorithm StaR.

Document parallelization (DocPar) is a straightforward solution where a free thread Th_f of the processor is assigned to execute the filtering process for an incoming document d_i. Thread Th_f executes the filtering process for d_i as described in Algorithm Filter; if a new document d_n arrives at the system it is assigned to another unoccupied thread. Thereby, every available thread in the system can be utilized to improve the filtering performance. In this way, we avoid the sequential filtering of a queue of incoming publications and significantly reduce their service time.

Contrary to the document parallelization approach, root parallelization (RootPar) assigns a set of available threads $\{Th_i, \ldots, Th_n\}$ to serve the Forest during the filtering time. Each thread Th_d is dedicated to a random sub-set of roots $\{T_j, \ldots, T_m\}$ of the Forest. When a document d_i is published the filtering procedure is executed as described in Algorithm Filter, while the only difference is that when a trie T with $root(T) = k \in d_i$ is located the traversal of that trie is handled by the thread that is assigned to trie T (Figure 3.4, line 3). Thus, the Forest can be searched simultaneously by more than one threads.

Both approaches extend StaR to multi-core environments and allow it to exploit shared memory capabilities of modern hardware to perform faster filtering, opposed to the sequential approach where a single thread traverses all the matching tries T in order to examine possible matching queries.

3.4 Competitors

To evaluate our algorithm (presented in the previous sections), we implemented two trie-based competitors from the existing state-of-the-art solutions in the literature: *(i)* Algorithm ReTrie [4] that employs partial query reorganization for poorly clustered queries and *(ii)* Algorithm Tree [5] that does not employ any form of query reorganization and indexes queries in a deterministic way (i.e., based on the order of insertion).

3.4.1 Algorithm ReTrie

Algorithm ReTrie [4] organizes queries into tries and maintains a data structure that monitors the number of poorly clustered queries in the system (i.e., queries with only a few words clustered in the trie). When a certain threshold of poorly clustered queries is reached, the reorganization process is triggered and all poorly indexed queries are examined and re-indexed. By choosing to reposition only poorly indexed queries, Algorithm ReTrie misses many available reorganization options and is bound to use the existing tries (new trie creation is very rare). Time complexity for Algorithm ReTrie is $\mathcal{O}(t^t)$ for the query indexing phase and $\mathcal{O}(Qt^t)$ for query reorganization, where Q is the number of queries to be reorganized and t is the number of distinct query words. Similarly the time complexity of filtering is $\mathcal{O}(t^t)$, where t is the number of distinct words in the document.

The main differences between Algorithms StaR and ReTrie are as follows. Algorithm StaR *(i)* uses a query indexing mechanism that builds tries based on statistical information about query words, *(ii)* destroys poorly performing tries and creates new ones based on query scores, *(iii)* repositions newly inserted queries only, and *(iv)* emphasizes trie shape rather than trie compactness. Following our running example, Algorithm ReTrie would not reposition any query in the forest of Figure 3.1, although better alternatives are available as presented in Figure 3.3.

3.4.2 Algorithm Tree

Algorithm Tree [5] organizes queries in tries by relying on common subsets of queries, without employing frequency information or resorting to query reorganization. Contrary to both ReTrie and StaR algorithms, that seek for the best position in the available tries to index a new query, Algorithm Tree places the query in deterministic fashion, by sorting query words alphabetically in an effort to increase the common subsets of words. This deterministic query placement misses many good indexing positions for the queries, as it emphasizes insertion time over query clustering. Time complexity for Algorithm Tree is $\mathcal{O}(tlogt)$ for the query indexing phase and $\mathcal{O}(t^t)$ for the filtering phase, where t is the number of distinct query and document words respectively.

The implemented competitors highlight the different aspects examined by our research work. Compared to Algorithm ReTrie, Algorithm StaR demonstrates the difference between partial and extensive query reorganization, and emphasizes the importance of statistics in the reorganization process. Additionally, Algorithm Tree is implemented to demonstrate the difference between deterministic and non-deterministic query placement, and its lack of reorganization in the filtering efficiency.

3.5 Supporting Richer Query Languages

Algorithm StaR is easily extendible to more sophisticated data models and query languages by adding appropriate data structures and modifying the filtering process accordingly. In this section, we outline the necessary additions and modifications to support attributes and proximity operators.

Attributes may be introduced by creating a hash table (that will use the attribute name as key) and a forest of tries (like the ones presented in Figures 3.1 and 3.3) for each attribute A in the data model. Access to each data structure will then be provided by a hash table that will use the attribute name as key. Similarly, we may use one table per attribute to maintain statistics of words in each attribute. Reorganization will then be executed independently for each attribute; when a document d is published, the filtering procedure for Algorithm StaR is modified so that for each attribute A in the document and for each keyword k in A, the trie in the forest of A that has the word k as root is traversed using the filtering algorithm of Section 3.3.3. Finally, list $id(n)$ in each trie node n gives implicitly all queries that match d for attribute A; thus, a query q will match a document d if q matches all attributes of d.

Different types of proximity formulas (e.g., operators NEAR of Lycos and Altavista, '*' operator of Google, or proximity formulas of arbitrary distance intervals as in [113]) may also be easily supported by StaR. The words that are operands in proximity formulas are stored in the forest of tries (since proximity is a stricter form of conjunctive queries), while the distance intervals are stored in a separate data structure. Proximity formulas are initially evaluated as conjunctive queries, and the satisfaction of word order and distance is evaluated separately using an algorithm like [44]. Other useful query components, like equality, disjunction, and negation, are also straightforward to support in the current indexing scheme. Finally, notice that handling more complex semantics is possible through semi/fully-automated query generation as in recommender systems, or through query expansion/augmentation techniques with the aid of taxonomies or dictionaries.

3.6 Experimental Evaluation

In this section, we proceed in presenting a series of experiments that aim at providing a quantified efficiency comparison of our proposed solutions against state-of-the-art competitors. To this end, we compare the filtering performance of Algorithm StaR, described in Section 3.3, to efficiently solve the IF problem against the trie-based Algorithm ReTrie [4] and Algorithm Tree [5]; both solutions were described in detail in Section 3.4.

Since we are examining an information filtering scenario, it is important to investigate the behavior of the algorithms when varying both the query database and the document data sets. Thus, we choose to simulate four distinct experimental setups, that aim at providing a detailed performance evaluation. Specifically, we carefully designed two experimental setups that examine the performance of the proposed algorithms under two different query databases, each with different properties; these setups allow us to test the behavior of the algorithms under varying queries and a constant document set. The third experimental setup examines the proposed algorithms when varying the document set, but, considering a specific query database. Finally, the fourth experimental setup examines the performance of our solutions under a real-world query set, the Million Query Track [114] and the ClueWeb09 corpus.

Description	Value
Vocabulary size	$3.14M$
Average document size (words)	53
Maximum document size (words)	$14,425$
Minimum document size (words)	1
Maximum word size (letters)	57
Minimum word size (letters)	1

Table 3.5: Characteristics of the DBpedia corpus.

3.6.1 Experimental Setup

In this section we present the data sets and discuss the way we proceeded in generating the query sets. Furthermore, we present the underlying algorithmic configuration, since each algorithm is dependent on different parameters. Finally, we provide the technical configuration as well as, the metrics employed in our evaluation.

Data and Query Sets

For the evaluation we used two different real-world datasets and both synthetic and real query sets as described below.

The *DBpedia* Corpus. The first dataset used in our experiments is based on the *DBpedia* corpus, which consists of a wide and thematically unfocused set of documents; it contains more than $3.7M$ documents, has a total vocabulary of $3.14M$ words, and its average document size is 53 words. Each document is an extended Wikipedia abstract downloaded from the *DBpedia* website (`http://wiki.dbpedia.org/Downloads39`). Table 3.5 summarizes some key characteristics of the *DBpedia* corpus.

The *Clue Web 09* Corpus. The second dataset used of our experimental setup is the ClueWeb09 corpus, a collection of web pages crawled form the World Wide Web. ClueWeb09 contains approximately a billion web pages and constitutes a wide and thematically unfocused set of documents.

Continuous Query Sets

As no database of real-life continuous queries was available to us, except by obtaining proprietary data (e.g., by a news alerting system), we constructed three query sets with different properties discussed below, similarly to [4, 5]. In an information filtering scenario it is important to investigate the performance of the proposed algorithms under varying query sets as much as varying document sets. Specifically our first two experimental setups were based on a single document set, taken from *DBpedia* corpus, but utilized two query databases with different properties. These two setups, namely *the general query collection* and *the focused query collection*, will allow as to test the behavior of the examined Algorithms under varying queries. The third experimental setup was based on documents from *DBpedia* corpus, where the query database maintains its properties but the documents nature changes allowing us to examine varying document sets. Finally, the fourth experimental setup was based on the real-world query set, Million Query Track [114] and the ClueWeb09 corpus.

The general query collection

The *general query collection* contains queries formed by conjunctions of different terms; each term conjunct is selected equiprobably among the set of words forming the *DBpedia* corpus vocabulary ($3.14M$) and the set of wikipedia document titles. Due to the nature of the *DBpedia* corpus and the corresponding vocabulary size, the constructed queries are expected to cover a wide variety of topics and, thus, share few common words between them. This restricts clustering opportunities and makes this setting a *stress test* for the filtering performance of the algorithms, as they are forced to identify and exploit the few commonalities between the indexed queries. For this query set, we select $50K$ documents' extended abstracts from *DBpedia* and use them as the incoming documents.

The focused query collection

The *focused query collection* is constructed by selecting $50K$ thematically related extended abstracts from *DBpedia* and using the $46K$ distinct words appearing in those documents. As these queries become more focused and the vocabulary of the query database is restricted, more clustering opportunities appear. In this setting,

	Collection	
Description	General	Focused
Vocabulary size	$762K$	$46K$
Average query length	$3-5$	$3-5$
Complex terms size	$2.8M$	0
Average complex term length (words)	2.47	0
Complex terms per query length	50%	0%

Table 3.6: Characteristics of general and focused query collections.

the performance of the different algorithms is expected to be similar, as all will exploit the many clustering opportunities offered. Notice that the $50K$ incoming documents utilized in this section are the same ones used in the general query collection, since we aim to study the behavior of our algorithms when varying the query set.

Table 3.6 summarizes some key characteristics of the *general* and *focused query collections*. More specifically the vocabulary size of the general query collection is $726K$ unique terms, while the vocabulary size of the focused query collection is much more restricted at $46K$ terms. Additionally, each query set, has three subsets of queries, each subset has an average length of 3, 4, and 5 terms. The general query collection contains $2.8M$ complex terms of an average length of 2.47 simple terms, while the focused query collection does not contain any complex terms. Finally, 50% of the terms that formulate a query, in the general focused collection, are complex terms.

Algorithm Configuration

There is a number of system parameters, which affect the performance of the presented algorithms that have to be determined and set. For our evaluation, we use a clustering ratio of 0.8 for ReTrie (selected after an exhaustive scan of all possible parameter values), while query reorganization for under-clustered queries is invoked every $I_Q = 125K$ query insertions. Regarding StaR, the reorganization of

Parameter	Description	Baseline value
Q_L	Average number of words per query	5
D_L	Average number of words per document	53
I_Q	Number of incoming queries	$500K$
I_D	Number of incoming documents	$50K$
DB	Number of queries indexed in the database	$3M$
Th	Number of threads used for the filtering process	1

Table 3.7: Parameters' description and their baseline values.

each variant is invoked when $I_Q = 500K$ new queries are indexed for StaR-LF and StaR-LR, and when $I_Q = 125K$ and $I_Q = 250K$ for StaR-HF and StaR-HR respectively. To measure the performance of the filtering process, we utilise the general query collection in our experimental setup to compare the performance of the two proposed approaches.

Finally, for the parallelization of the filtering process we utilized a set of 6 threads available in the processor. The baseline values for each tunable parameter in the experimental evaluation are: *(i)* average query length $Q_L = 5$, *(ii)* average document length $D_L = 53$, *(iii)* number of incoming queries $I_Q = 500K$, (iv) number of incoming documents $I_D = 50K$, *(v)* query database size $DB = 3M$, and *(vi)* threads used in the filtering process $Th = 1$. Table 3.7 summarizes the parameters examined in our experimental evaluation. For more details about the parameter setting we refer the interested reader to [4, 5].

Metrics Employed

In our evaluation, we use the *number of nodes* in the forest of tries to measure the quality of clustering for each algorithm. As all algorithms index the same query database, a lower number of trie nodes indicates a more compact clustering.

Additionally, we use *filtering time* to measure the filtering performance of each algorithm, i.e., the amount of time needed to locate all continuous queries satisfied by an incoming document. We also present the algorithms' *throughput* to study their performance as the query database size increases, i.e., the amount of filtered data per second. Finally, we measure *insertion* and *reorganization time* to identify the time needed to index and reorganize queries, and give the *memory requirements* for each algorithm.

Technical Configuration

All the algorithms shown in the experiments of this section were implemented in C++. For the parallelization of the filtering process, C++ library <thread> was used. An off-the-shelf PC with a Core $i7$ 3.6GHz processor and $8GB$ RAM running Ubuntu Linux 14.04 was used. The time shown in the graphs is wall-clock time and the results of each experiment are averaged over 10 executions to eliminate any fluctuations in time measurements. All algorithms under consideration index the same query database, and filter the same set of incoming documents.

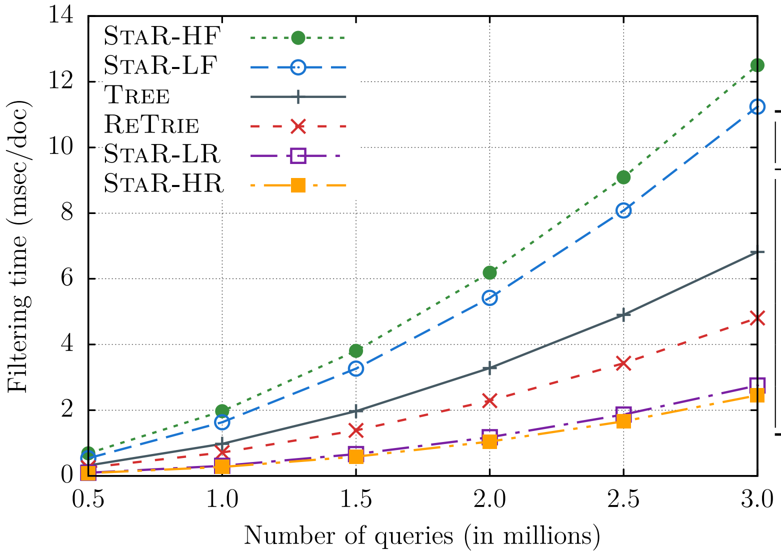
3.6.2 Results for the General Query Collection

In this section, we present the results of the evaluation process for the general query collection described earlier and highlight the most significant findings for the proposed algorithms.

Comparing Filtering Time

This section discusses the results concerning the filtering time required to match an incoming document against a database of queries. The time shown in the graphs represents the average time spend to filter a collection of $I_D = 50K$ documents with $D_L = 53$ words.

Figure 3.5 shows the average time in milliseconds needed to filter a collection of $I_D = 50K$ documents with $D_L = 53$ words against a database of different size when indexing queries with $Q_L = 5$ terms. Observe that filtering time increases for all algorithms as the query database size increases. Algorithms StaR-LR and



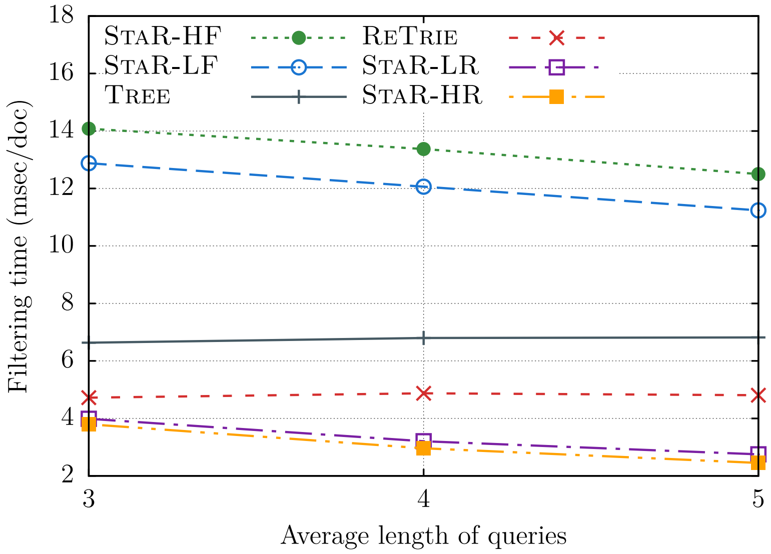
Parameter	Value
Q_L	5
D_L	53
I_Q	$500K$
I_D	$50K$
DB	$0.5M - 3M$
Th	1

Figure 3.5: Comparing the filtering time efficiency, of all algorithms, when varying the size of query database DB.

StaR-HR (that store rare words near the roots of tries) achieve the lowest filtering times, suggesting better performance than their counterparts Algorithms StaR-HF and StaR-LF (that store frequent words near the roots) and competitor Algorithms ReTrie and Tree. Additionally, Algorithms StaR-HF, StaR-LF, and Tree are more sensitive to query database size changes than the rest of their competitors since frequent words are stored near trie roots, which requires traversing more tries at filtering time.

In more detail, Algorithm StaR-LR filters incoming documents 74.75% faster, when compared against Algorithm ReTrie and 147% faster than Algorithm Tree. Moreover, StaR-HR outperforms ReTrie by 96.14% and Tree by 178.15%. On the other hand, Algorithms StaR-HF and StaR-LF are slower than Algorithms ReTrie and Tree. More specifically, Algorithm StaR-HF needs 61.5% more time to filter an incoming document than ReTrie and 45.46% more time than Tree. Finally, StaR-LF shows similar performance, executing the filtering process 57.22% slower than ReTrie and 39.33% slower than Tree.

Figure 3.6 shows the filtering time for queries of different length. It is worth noting, that all algorithms (except Algorithm Tree that remains unaffected) improve their filtering performance when the query size is increased, as longer



Parameter	Value
Q_L	$3-5$
D_L	53
I_Q	-
I_D	$50K$
DB	$3M$
Th	1

Figure 3.6: Comparing the filtering time efficiency, of all algorithms, when varying the average query length Q_L, for a query database of $DB = 3M$ queries.

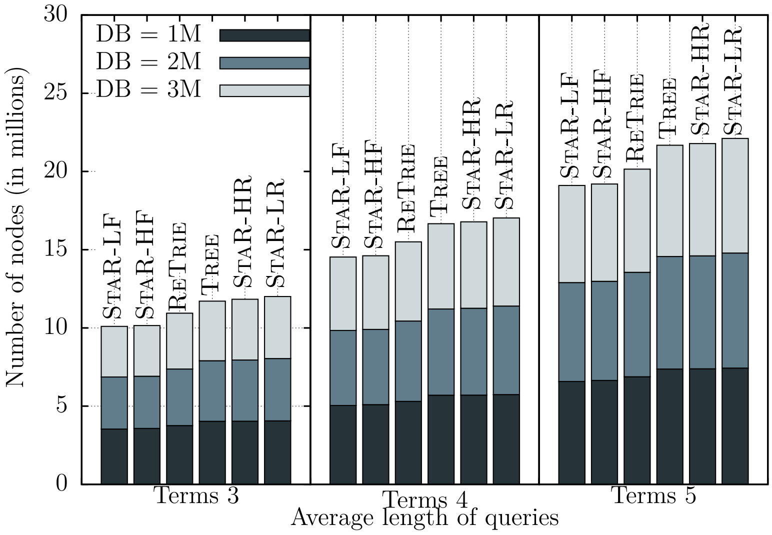
# of queries	Algorithm							
	STAR-LR		STAR-HR		STAR-LF		STAR-HF	
	Time (msec/doc)	Increase (times)	Time (msec/doc)	Increase (times)	Time (msec/doc)	Increase (times)	Time (msec/doc)	Increase (times)
$1M$	0.20	–	0.087	–	0.16	–	0.33	–
$10M$	0.28	0.4x	0.34	2.9x	0.81	4x	11.56	34x
$100M$	2.31	10x	3.66	41x	5.24	32x	39.72	119x

Table 3.8: Filtering scalability in a big data setup for all variations of Algorithm STAR.

queries provide better indexing alternatives and more opportunities for pruning of tries at filtering time. In addition, Algorithms STAR-HF and STAR-LF present a high decrease in filtering time which is attributed to their poor filtering performance that has more margin for improvement. On the other hand, Algorithms STAR-HR, STAR-LR, and RETRIE exhibit a smaller decrease in filtering time when the query length is increased.

Comparing Filtering Time in a Big Data Setup

Table 3.8 reports the results for a stress test of the proposed algorithms under a big data setup. To do so, we conducted a filtering experiment in an Intel Xeon



Parameter	Value
Q_L	$3-5$
D_L	53
I_Q	$500K$
I_D	$50K$
DB	$1M$, $2M$, $3M$
Th	1

Figure 3.7: Comparing the number of trie nodes, when varying the average query length Q_L and the query database size DB.

$2.7GHz$ server with $264GB$ RAM, using up to two orders of magnitude more queries and the whole *DBpedia* document collection as the stream of documents to be filtered. The resulting experiment ended up filtering a stream of $3.7M$ documents (totaling an uncompressed size of $5.5GB$) against a query database of $100M$ queries (totaling a size of 7GB) for all variations of our algorithms. In the "Time" column of each algorithm we report the average filtering time per document for each query database, while in the "Increase" column we report the increase (in number of times) for the $10M$ and $100M$ queries against the base case of $1M$ queries. Our findings show that our solution is scalable: for a 10 (resp. 100) times increase in the query database size, the corresponding increase in filtering time of our best solution is no more than 0.4 (resp. 10) times.

Comparing Trie Compactness

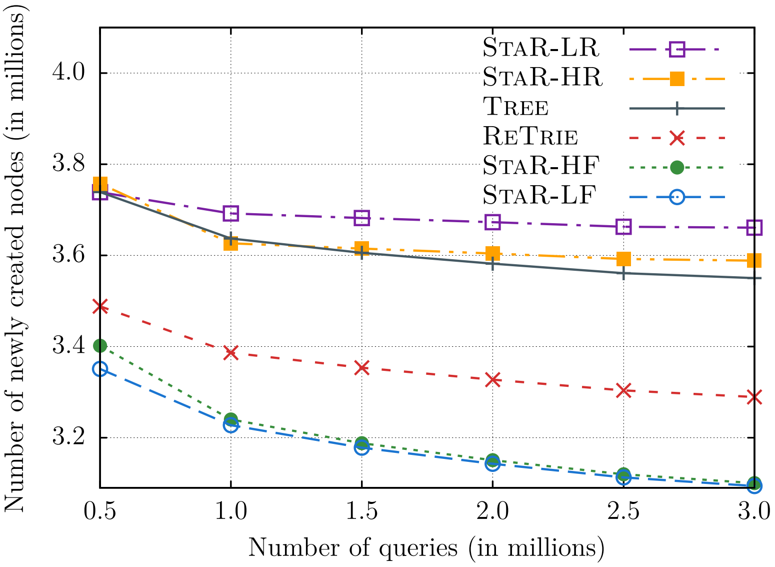
Studying the number of new trie nodes created by each algorithm will demonstrate the compactness of the created tries and consequently, the query clustering achieved. In this section, we aim at examining the relationship between the compactness of the forest and the filtering performance of each algorithm.

Figure 3.7 shows the number of nodes created by each algorithm for databases containing 1, 2 and 3 millions queries of different average lengths. Observe that

StaR-LR, StaR-HR and Tree create relatively larger forests, while StaR-HF and StaR-LF create relatively smaller ones. Also, increasing the number of query length, from 3 to 5 terms on average, results in an increase in the forest size. Interestingly, the difference in trie nodes between the algorithms remains unchanged. This can be explained as follows. StaR-LR and StaR-HR create relatively large forests as they index rare words towards the roots of tries and frequent words towards the leaves. In this way, the lower levels of tries tend to repeat words with high frequency of occurrence, thus, creating many tries with higher fanout and lower node reusability. Contrary, Algorithms StaR-HF and StaR-LF create relatively small forests as they push the most frequent words towards the roots of the tries. This, creates more compact tries as many repeated words are indexed in the same trie node, thus, resulting in high node utilization. Algorithm Tree creates large forests as it utilizes a naive query placement technique and implements no reorganization. Finally, Algorithm ReTrie creates an average-sized forest, as it focuses on query clustering rather than word statistics to index queries.

The results in Figure 3.7 suggest that StaR-HF creates 4.94% (948K) less nodes than ReTrie and 12.88% (2.474M) less nodes than Tree. Similarly, StaR-LF creates 5.46% less nodes than ReTrie and 13.44% less nodes than Tree. On the other hand, StaR-HR creates 7.5% more nodes than ReTrie and 0.5% more nodes than Tree. Additionally, StaR-LR creates a larger forest by 8.86% nodes compared to ReTrie and by 1.96% nodes compared to Tree.

Figure 3.8 shows the increase in forest size after the insertion and reorganization of $500K$ queries with $Q_L = 5$. As expected, results are similar to those of Figure 3.7, as all algorithms demonstrate the behavior discussed above and rank with regard to node increase. One important finding emanating from this comparison is that the rate of node creation is decreasing as new queries arrive, since all algorithms use existing trie nodes to index the new queries. Additionally, Algorithms StaR-HF, StaR-LF and ReTrie are less sensitive to query insertion, demonstrating a continuously decreasing number of newly created nodes compared to StaR-LR and StaR-HR. This is due to the exploitation of frequent words between queries



Parameter	Value
Q_L	5
D_L	53
I_Q	$500K$
I_D	$50K$
DB	$3M$
Th	1

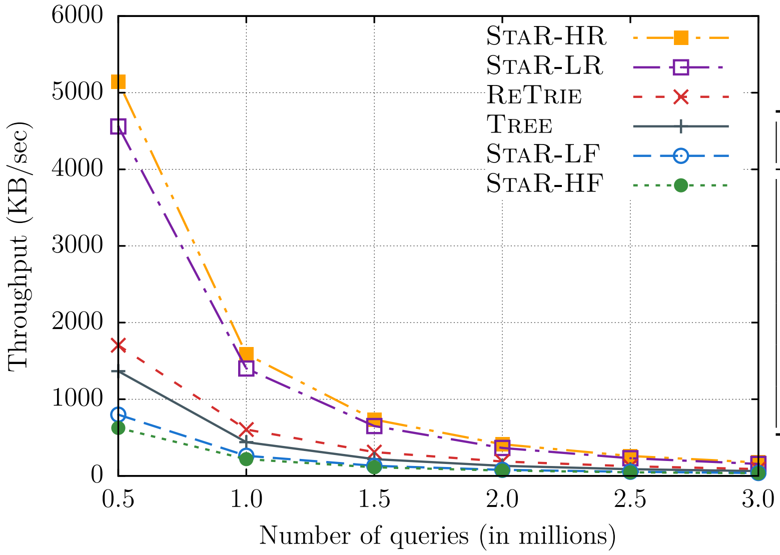
Figure 3.8: Comparing newly created trie nodes, when inserting queries in a query database of varying size DB.

and the construction of more compact tries compared to Algorithms StaR-LR and StaR-HR, where query clustering is affected by infrequent words.

All the above observations lead us to the conclusion that the algorithms which store frequent words near the roots of the tries (i.e., StaR-HF and StaR-LF) tend to create smaller forests compared to their counterparts and competitors. In the next section, we will demonstrate that the creation of compact tries is not always a clear indicator for filtering performance, as filtering time is also affected by the shape of tries and the words contained in the incoming documents.

Comparing Efficiency Against Compactness

Comparing the results of node creation and filtering time, we can *infer* that the nature of the forest and the way it is constructed has a more significant effect on filtering performance than trie compactness, since it is shown (Figures 3.5, 3.6, 3.7 and 3.8) that the algorithms creating the less compact forests result in the lowest filtering times. This can be explained as follows. Algorithms ReTrie, StaR-HF, and StaR-LF tend to locate and group queries under common words and create compact forests with frequent words near the trie roots. At filtering time, incoming documents match many trie roots, thus, needing to traverse more tries to examine



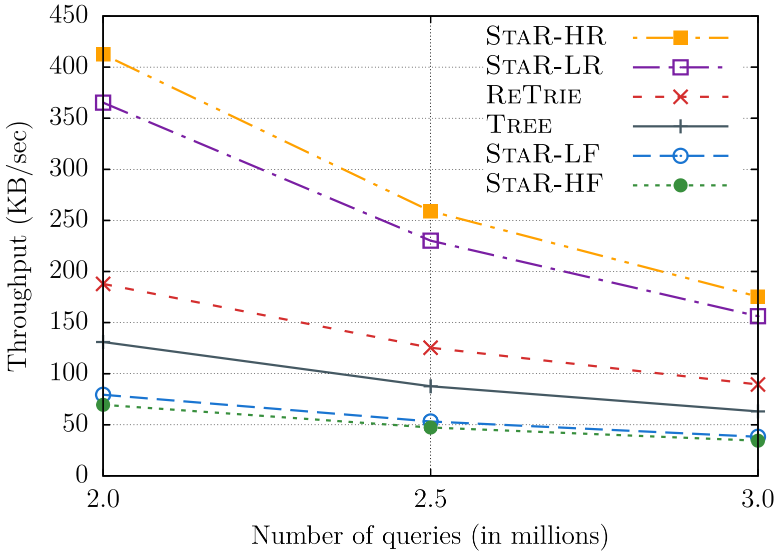
Parameter	Value
Q_L	5
D_L	53
I_Q	500K
I_D	50K
DB	0.5$M-3M$
Th	1

Figure 3.9: Comparing filtering throughput, when varying the query database size DB.

all possible query matches. Additionally, for each trie, the filtering process cannot prune many subtries, as the infrequent words are stored near the trie leaves. This results in the traversal of the whole trie structure all the way to the leaves of every subtrie to determine whether a query is relevant to the incoming document or not.

Contrary, Algorithms StaR-LR and StaR-HR locate and store rare words in nodes closer to trie roots, thus, organizing queries in subtries under their common (rare) words. This organization of queries, prevents the filtering process to visit many tries, and prunes subtries much earlier, due to the low probability of a rare word being present in an incoming document.

Notice also that the utilization of query score allows us to enforce at indexing time a query order based on query importance. In this way, queries consisting of many words ordered from rare to frequent (i.e., Algorithm StaR-HR) are inserted first during the reorganization phase allowing the creation of more tries. Those tries are then used as the guides that push frequent words further down the trie structure. This approach causes StaR-HR to be 10.9% more efficient in terms of filtering time when compared to StaR-LR.



Parameter	Value
Q_L	5
D_L	53
I_Q	$500K$
I_D	$50K$
DB	$2M - 3M$
Th	1

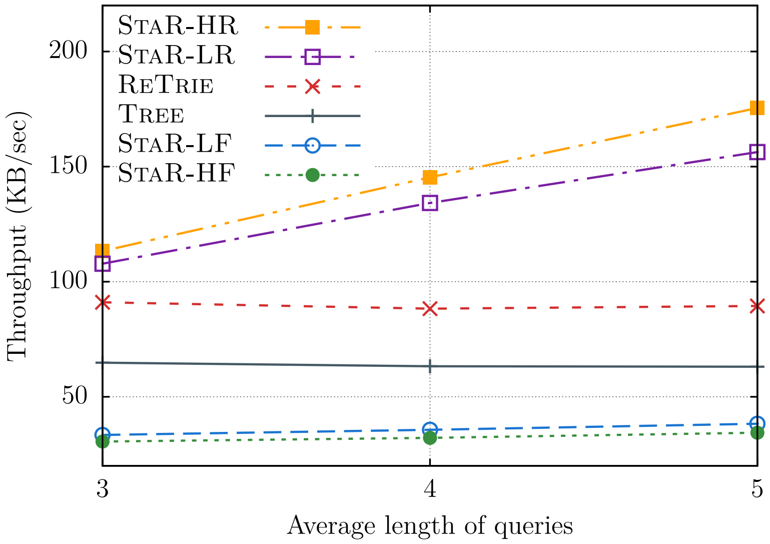
Figure 3.10: Comparing filtering throughput, when varying the query database size DB for $2M$ till $3M$ (Enhanced view of Figure 3.9).

Comparing Filtering Throughput

In this section, we present the findings of the algorithms throughput and their performance as the database of queries increases. Figures 3.9 and 3.10 shows the throughput in KB/sec needed to filter $I_D = 50K$ incoming documents against a query database of different size indexing queries with 5 terms. Notice that throughput decreases for all algorithms rather rapidly as the query database size increases. Algorithms StaR-HR and StaR-LR achieve higher throughput than the other variants of StaR (i.e., StaR-HF and StaR-LF) and their competitors Tree and ReTrie. Observe that Figure 3.10 is an enhanced view of Figure 3.9 for a clearer presentation of throughput results for $[2M, 3M]$ queries.

Figure 3.11 shows the filtering throughput for queries of different lengths. As expected all algorithms (except Tree) exhibit an increase in filtering throughput as longer queries provide better indexing opportunities. Algorithms StaR-HR and StaR-LR present higher increase in terms of filtering throughput, which is attributed to the nature of tries they create, namely the pruning of tries at filtering time due to the existence of rare words near trie roots.

Notice that the throughput of all algorithms decreases with the increase in the



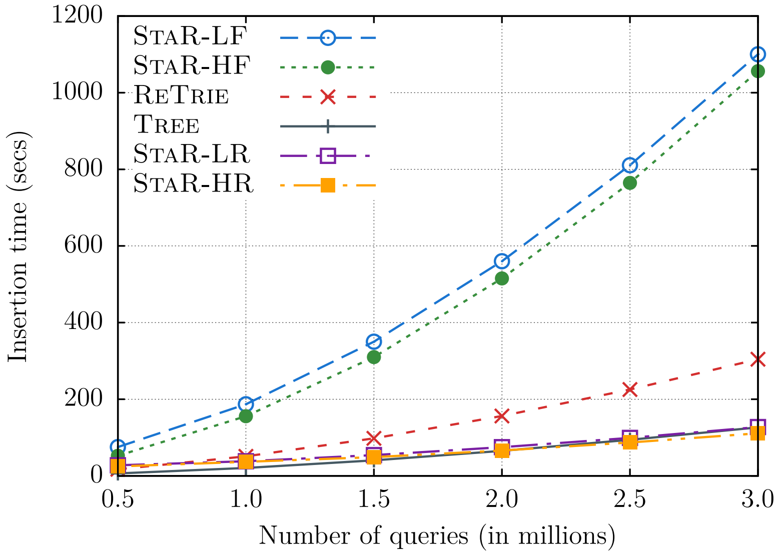
Parameter	Value
Q_L	$3-5$
D_L	53
I_Q	$500K$
I_D	$50K$
DB	$3M$
Th	1

Figure 3.11: Comparing filtering throughput, when varying the average query length Q_L, for a database size of $DB = 3M$ queries.

database size, since each incoming document has to be matched against more queries indexed in the data structures of the algorithms. Moreover, the throughput of all algorithms (apart from StaR-HR and StaR-LR) remains relatively unaffected by the increase in the average query length, since these algorithms focus on forest compactness and thus their filtering throughput is not affected by the number of query words. Contrary, Algorithms StaR-HR and StaR-LR (that place rare words at the top of the tries) benefit from longer queries, since they may exploit more pruning opportunities.

Comparing insertion time

In this section, we discuss the time needed to insert a query and reorganize databases of different sizes. Figure 3.12 shows the time in seconds required to insert $I_Q = 500K$ queries with $Q_L = 5$ terms in databases of varying size. We observe that the insertion time of all algorithms increases with the query database size. Algorithms StaR-HF and StaR-LF need more time to insert new queries in the existing database since they need to examine more tries as possible indexing locations. This happens due to the nature of the tries, which index frequent words near the trie roots, thus, create more indexing opportunities. Algorithm ReTrie requires less time to insert



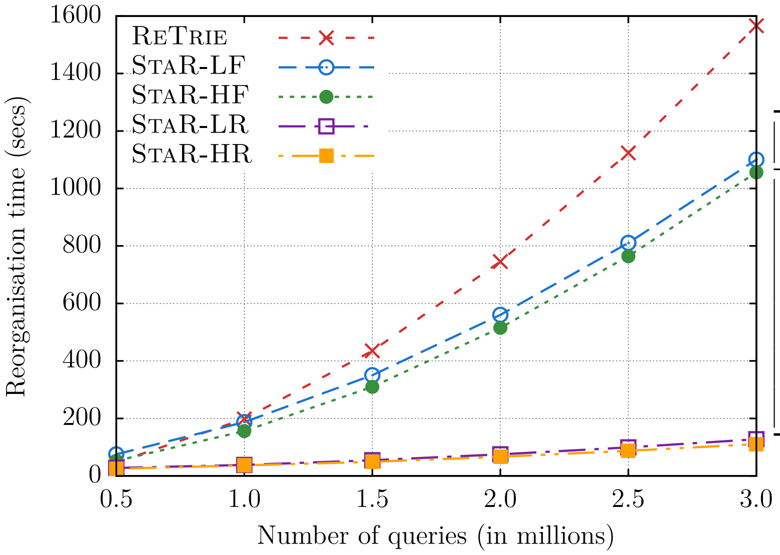
Parameter	Value
Q_L	5
D_L	-
I_Q	$500K$
I_D	-
DB	$0.5M - 3M$
Th	1

Figure 3.12: Comparing insertion time, when varying the query database size DB, for a query set of average length $Q_L = 5$ terms.

new queries in the database than STaR-HF and STaR-LF, as it has less indexing opportunities. Next, Algorithm Tree requires less time to insert queries in the database as it places queries deterministically in the tries. Finally, Algorithms STaR-LR and STaR-HR tend to explore less candidate tries as rare words in trie roots exclude many clustering possibilities.

Comparing Reorganization Time

In this section, we measure the time needed to reorganize $I_Q = 500K$ for varying database sizes. Figure 3.13 presents the time needed to reorganize a query database for each of the presented algorithms (except Tree that does not consider any query index reorganization). Algorithms ReTrie, STaR-LF and STaR-HF need more time to reorganize $I_Q = 500K$ queries compared to the rest of the examined algorithms, and the time needed increases as the query database increases. This can be explained as follows. As the database indexes new queries the clustering opportunities for queries increase. Algorithm ReTrie that aims at reorganizing poorly clustered queries has to scan the whole query database to locate them and subsequently identify better indexing positions. Similarly, Algorithms STaR-LF and STaR-HF are affected by frequent words in the higher levels of the forest

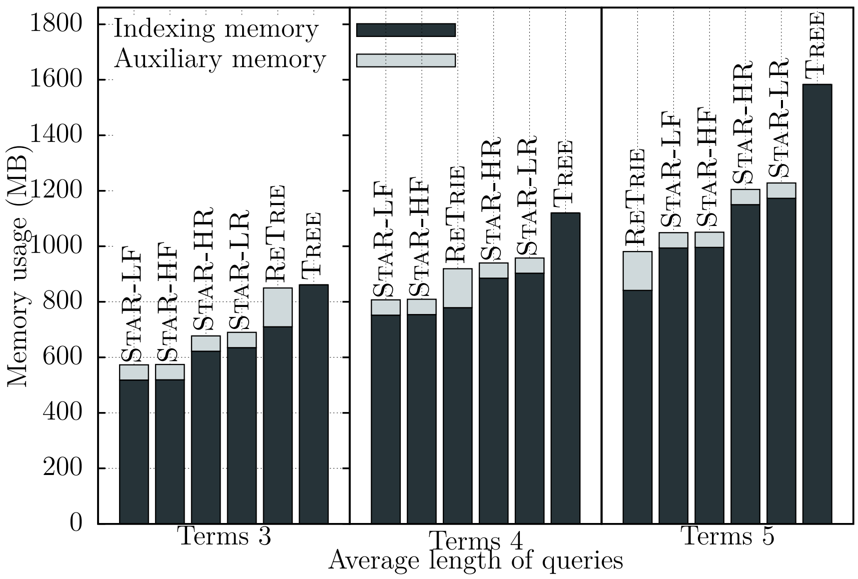


Parameter	Value
Q_L	5
D_L	-
I_Q	$500K$
I_D	-
DB	$0.5M - 3M$
Th	1

Figure 3.13: Comparing the reorganization time, when varying the query database size DB, for a query set of average length $Q_L = 5$ terms.

resulting in an extensive search on the query database at reorganization time. Notice that, as the database grows in size, the re-indexing options increase, thus, resulting in increased reorganization time. Contrary, Algorithms StaR-LR and StaR-HR are slightly affected by the increase in database size, due to the use of infrequent words near trie roots.

To test the efficacy of full query database reorganization we have considered such a scenario for Algorithms ReTrie and StaR-LR (our fastest performing solution); in our setup both algorithms executed a complete reorganization for a query database of $3M$ queries with $Q_L = 5$ terms. The total reorganization times obtained were 68 minutes (a three orders of magnitude or 2626 times increase from the partial reorganization solution) for Algorithm ReTrie and 24 minutes (a four orders of magnitude or 11673 times increase from the partial reorganization solution) for Algorithm StaR-LR. The respective gain in filtering time was a 96% decrease for Algorithm ReTrie and a 95% decrease for Algorithm StaR-LR in filtering time. These results suggest that choosing a complete database reorganization provide a substantial gain in filtering time, however a full query database reorganization is an expensive choice, when taking into consideration the three (Algorithm ReTrie) and four (Algorithm StaR-LR) orders of magnitude increase in reorganization time.

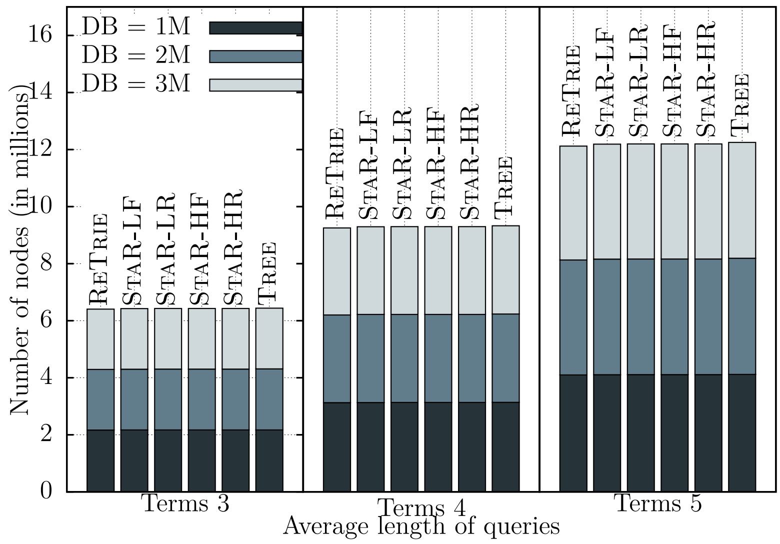


Parameter	Value
Q_L	$3-5$
D_L	-
I_Q	-
I_D	-
DB	$3M$
Th	1

Figure 3.14: Comparing memory requirements, for a query database of size $DB = 3M$ and average query length $Q_L = 5$ terms.

Thus, when employing this approach (in a real world scenario) this substantial trade-off should be taken into consideration.

In order to assess the memory requirements of the presented algorithms, we have also designed and executed suitable experiments. Figure 3.14 exhibits a good overview of the results for $DB = 3M$ queries and varying Q_L terms. For a database of $DB = 3M$ and $Q_L = 5$, Algorithm ReTrie has the lowest memory requirements needing 981 MB for storing the query database and all indexing components, while Algorithms StaR-LF and StaR-HF need approximately 1 GB of memory to store this information. Algorithms StaR-LR and StaR-HR need more than 1.1 GB of main memory due to the non-compact forests they create, and Algorithm Tree needs around 1.5 GB memory to store the query database. The excessive memory requirements of Algorithm Tree (compared to StaR and ReTrie) are explained as follows. Algorithm Tree creates a new node for every term that can not be indexed into an existing trie. In contrary, StaR makes use of the *uexp* structure (as described in Section 3.3.1), which allows the delay of the node creation, thus, sorting the word in a list of strings and resulting to less indexing memory requirements. Notice also, that StaR's variants and ReTrie have extra memory requirements in order to keep the appropriate data in auxiliary structures



Parameter	Value
Q_L	$3 - 5$
D_L	-
I_Q	$1M$
I_D	-
DB	$1M - 3M$
Th	1

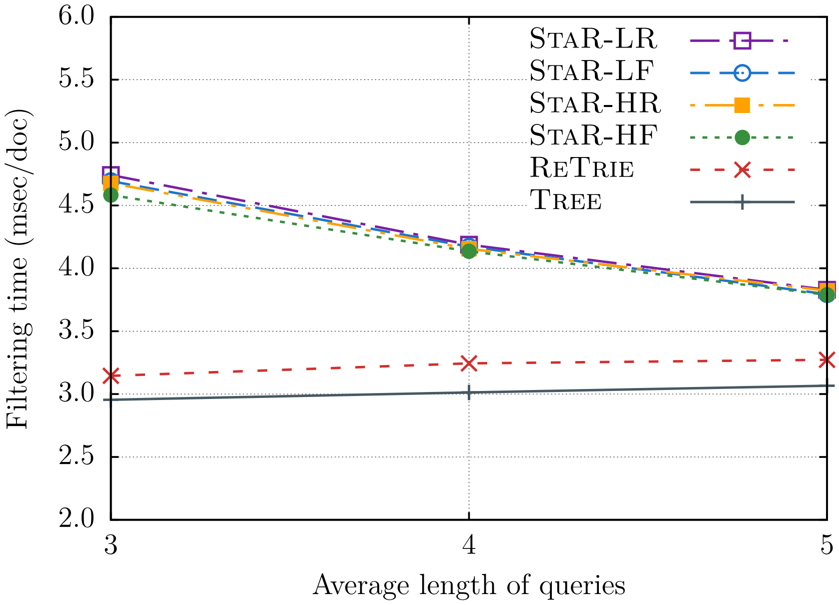
Figure 3.15: Comparing the number of trie nodes, when varying the average query length Q_L and the query database size DB, for the *focused query collection.*

for their reorganization phase; Algorithm ReTrie requires 150 MB and StaR's variants 50 MB extra (shown in Figure 3.14).

3.6.3 Results for the Focused Query Collection

In this set of experiments, we examine the performance of the algorithms under the *focused query collection*, as described in Section 3.6. To this end, we present the most interesting results and insights that arise in this experimental setup.

Figure 3.15 shows the number of nodes created by each algorithm for databases containing 1, 2 and 3M queries of different lengths. Observe that all algorithms behave similarly in terms of nodes created; the differences in the sizes of the resulting forest among the examined algorithms did not exceed 1%. This was expected as the small vocabulary of the focused collection created many clustering opportunities for all algorithms. In such a setting it is not important where to index a given query, as most queries will eventually be well-clustered. Naturally, the differences in filtering time (shown in Figure 3.16) are not significant (notice the small scale of the y-axis) with Algorithms Tree and ReTrie performing slightly better than the StaR variants.



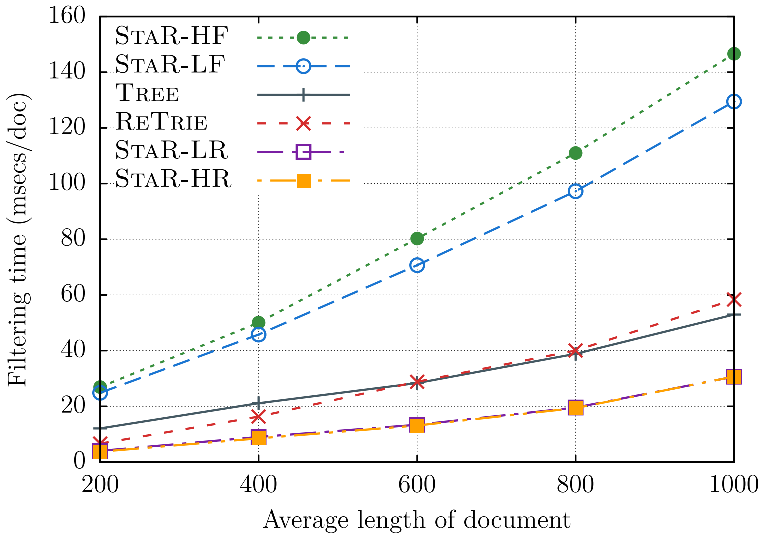
Parameter	Value
Q_L	3 – 5
D_L	53
I_Q	-
I_D	$50K$
DB	$3M$
Th	1

Figure 3.16: Comparing the filtering time efficiency, of all algorithms, when varying the average query length Q_L, for a query database of $DB = 3M$ queries, for the *focused query collection.*

The most interesting conclusions concerning the performance of the algorithms are extracted from the cross-comparison results of the examined query collections in Figures 3.6 and 3.16 . By comparing the filtering performance of the examined algorithms, we observe that Tree, ReTrie, StaR-HF, and StaR-LF are very sensitive to vocabulary variations; the increase in filtering time is 122%, 46%, 229%, and 196% respectively when increasing the vocabulary size, for the same query database size and query length. On the other hand, Algorithms StaR-LR and StaR-HR present a decrease in filtering time, since word statistics for bigger vocabularies contain more information to be exploited. Finally, comparing the absolute filtering times for the two collections, we conclude that Algorithms StaR-LR and StaR-HR deliver a steady filtering efficiency independently of the vocabulary size used.

3.6.4 Results for the Varying Document Length Collections

A key characteristic of the documents selected for the previous evaluations is their average length, which is 53 words for the *DBpedia* corpus. As we also want to

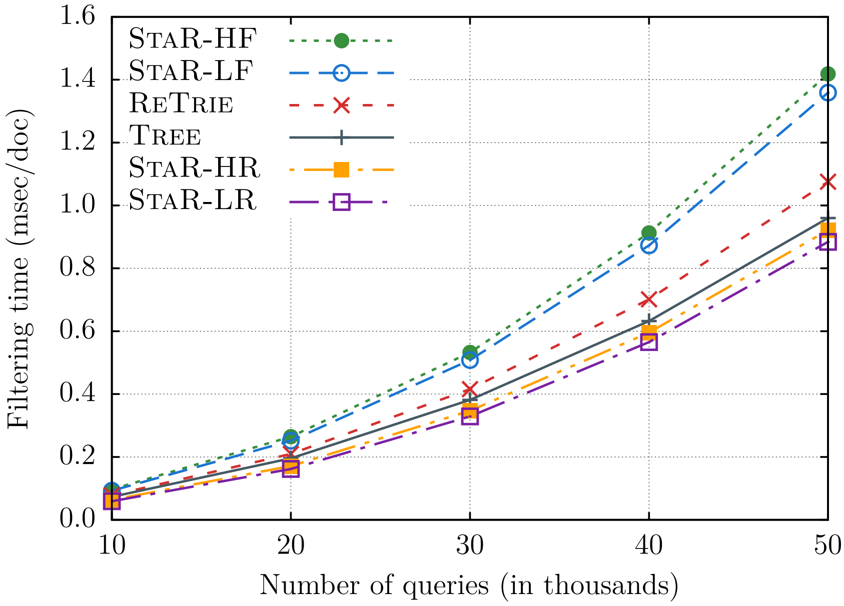


Parameter	Value
Q_L	5
D_L	200 – 1000
I_Q	-
I_D	100
DB	$3M$
Th	1

Figure 3.17: Comparing the filtering time efficiency, of all algorithms, when varying the average document length D_L, for a query database of $DB = 3M$ queries, for the *varying document length collection.*

examine the behavior of the algorithms when filtering larger documents, we select 100 documents with average length of 200, 400, 600, 800, and 1,000 words and present the most interesting results regarding the observed filtering times by the examined algorithms. Notice that, documents with a high number of words are expected to increase the probability of matching with the stored queries, resulting to a deeper search at our trie-based data structures. The query database used in this setup is the general query collection described in Section 3.6.1.

Figure 3.17 shows the time in milliseconds needed to filter documents with varying average length, when storing $DB = 3M$ queries with average length of $Q_L = 5$ words. As expected, the filtering time increases with the increase in incoming documents length. All algorithms though, present the same behavior discussed earlier in this section, except RETRIE that exhibits higher sensitivity to document size variation; as the average length of the incoming documents increases Algorithm RETRIE gradually needs more filtering time compared to TREE. This happens because Algorithms RETRIE, STAR-HF, and STAR-LF tend to group queries under common words and create forests with the majority of common words near the roots. Thus, due to larger document length, Algorithms RETRIE, STAR-HF, and STAR-LF are forced to visit the lower levels of the trie, i.e., near the leaves



Parameter	Value
Q_L	5
D_L	53
I_Q	$500K$
I_D	$50K$
DB	$3M$
Th	1

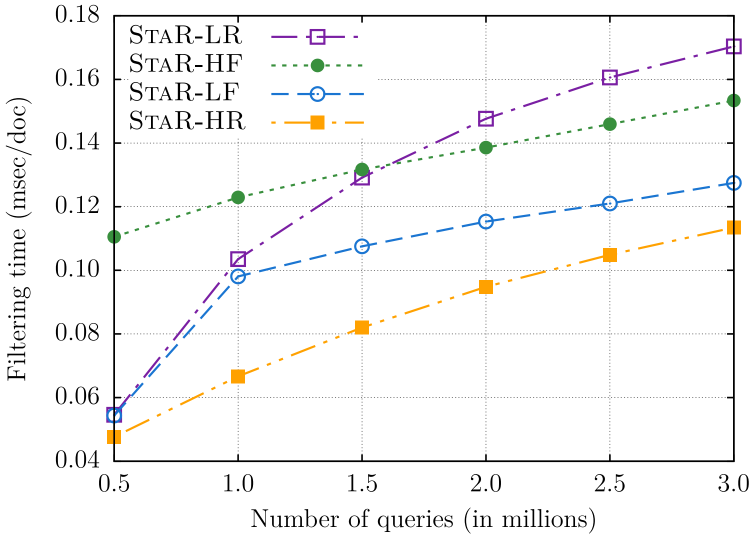
Figure 3.18: Comparing the filtering time efficiency, of all algorithms, when varying the query database DB, for the *Million Query Track* dataset.

where the fan out is greater due to poor clustering. The presented results suggest that StaR-LR and StaR-HR perform 70% better in filtering time compared to ReTrie and Tree. These differences in filtering times hold for all sizes of document collections, allowing us to conclude that StaR-HR and StaR-LR present a steady filtering efficiency that remains relatively unaffected from the document size.

3.6.5 Results for the Million Query Track Dataset

In this section, we present the most interesting results concerning the *Million Query Track* dataset (as described in Section 3.6.1).

Figure 3.18 presents the average time in milliseconds needed to filter a collection of $I_D = 50K$ incoming documents against a query database of increasing size. In this scenario, all algorithms exhibit a similar behavior as in the general query collection presented earlier. Although, Algorithms StaR-HR and StaR-LR maintain their low filtering time in this query collection, Algorithm ReTrie preforms slightly worse, while Algorithm StaR-LR is faster compared to StaR-HR. More specifically, StaR-HR filters incoming documents 16.7% faster than ReTrie and 4.1% faster than Tree. Moreover, StaR-LR outperforms ReTrie by 21.6% and Tree by



Parameter	Value
Q_L	5
D_L	53
I_Q	$500K$
I_D	$50K$
DB	$0.5M - 3M$
Th	6

Figure 3.19: Comparing the filtering time efficiency of DocPar, of all variations of StaR, when varying the size of query database DB.

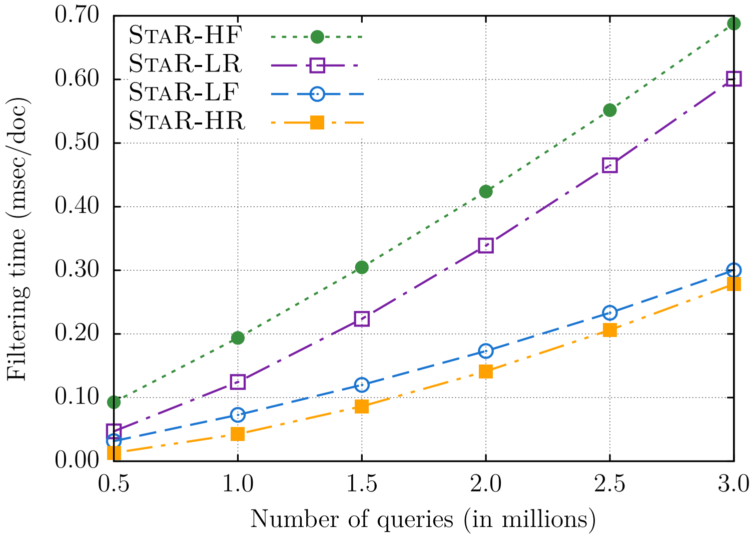
8.5%. Finally, we observe that Algorithms StaR-HR and StaR-LR maintain their high filtering efficiency under real-life data and query sets (that are very different from the previous experimental setup).

The close performance of the StaR variants in the *Million Query Track* dataset is due to the small vocabulary of the queries and is in line with our previous findings (for the *focused query collection*) on how query vocabulary affects filtering time (see Figure 3.16 and the explanation for this in Section 3.6.3). Moreover, the fact that ReTrie performs worse than Tree is attributed to the document length of the *ClueWeb09* corpus, which is much larger (1506 words) than the average document length in the *DBpedia* corpus. This ReTrie sensitivity to document size verifies our previous findings for the *DBpedia* corpus which are presented and discussed in Figure 3.17 and Section 3.6.4 respectively.

3.6.6 Results for the Parallelization of Filtering

In this section, we present the results concerning the two parallel filtering implementations of the four variations of Algorithm StaR, as described in Section 3.3.4.

Figures 3.19 and 3.21 present the results for the DocPar approach. In this approach, each document that arrives at the system is assigned to an unoccupied

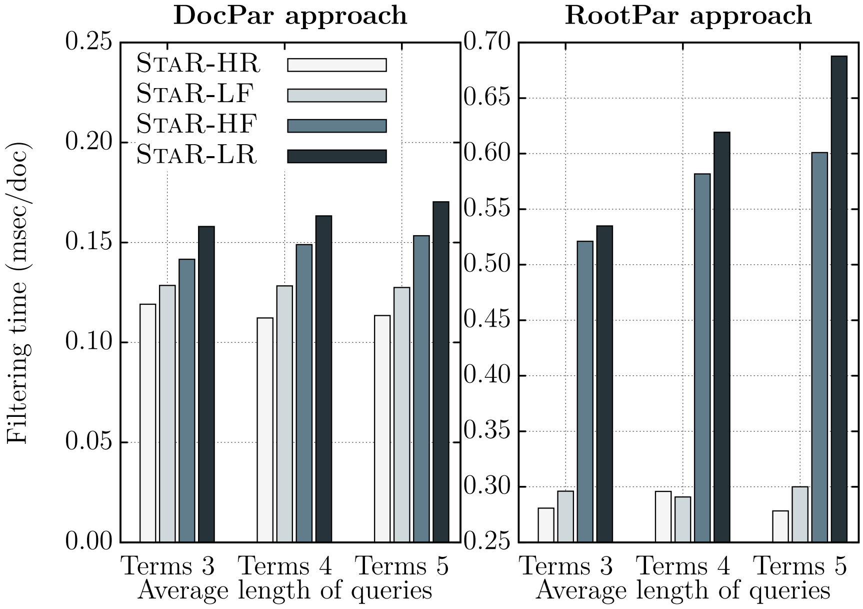


Parameter	Value
Q_L	5
D_L	53
I_Q	$500K$
I_D	$50K$
DB	$0.5M - 3M$
Th	6

Figure 3.20: Comparing the filtering time efficiency of RootPar, of all variations of StaR, when varying the size of query database DB.

thread. In our assessments we utilized 6 threads, thus allowing 6 documents to be filtered concurrently by the same processor. Figure 3.19 shows the average time in milliseconds needed to filter the general query collection of $I_D = 50K$ documents with $D_L = 53$ words against database of different size when indexing queries with $Q_L = 5$ terms. Figure 3.21 presents the filtering time for queries of different length; the variations of Algorithm StaR are significantly faster compared to their non-parallel counterparts (Figures 3.5 and 3.6).

Figures 3.20 and 3.21 present the results for the RootPar approach. In this approach, each thread is responsible for a set of roots present in the Forest. Thus, each root traversal is executed from a different thread during the filtering of a document. In our assessments we utilized 6 threads, allowing us to split the total number of roots into 6 even parts and assign each part to a single thread. Figure 3.20 shows the average time in milliseconds needed to filter the general query collection of $I_D = 50K$ documents with $D_L = 53$ words against database of different size when indexing queries with $Q_L = 5$ terms. In Figure 3.21 we give the filtering time for queries of different length; the variations of Algorithm StaR are significantly faster compared to their non-parallel counterparts (Figures 3.5 and 3.6). In more detail, algorithms StaR-LR and StaR-HR need 78.15% and 88.64%



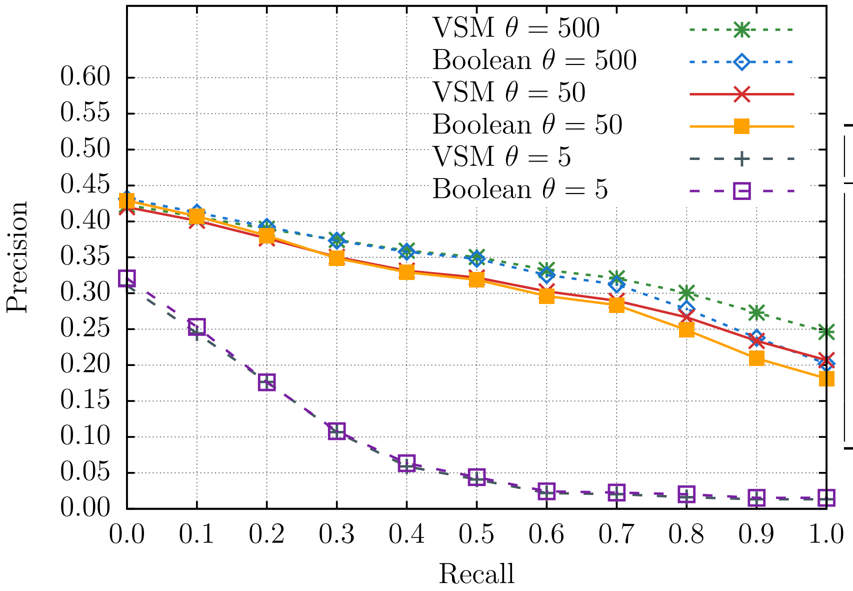
Parameter	Value
Q_L	$3-5$
D_L	53
I_Q	-
I_D	$50K$
DB	$3M$
Th	6

Figure 3.21: Comparing the filtering time efficiency of DocPar and RootPar, of all variations of StaR, for a query database of size $DB = 3M$, when varying the average query length Q_L.

less time respectively. Moreover, algorithms StaR-HF and StaR-LF reduce their filtering time by 94.49% and 97.33% respectively.

Comparing the two approaches, we can see that the DocPar is more efficient. Overall, in the DocPar approach all threads are continuously occupied by the stream of incoming documents that have to be filtered, as each incoming document is directed to the first available thread. Contrary, in the RootPar approach, due to the splitting of the trie roots to different threads the filtering tasks are unevenly distributed between the assigned threads as a result of the word distribution and word order in the incoming document. This leads to fully utilizing only a fraction of the available threads for each incoming document while the rest of the threads may stay inactive for long periods of time.

Comparing the two approaches, we can see that the first approach, DocPar, is the most efficient. Where, every incoming document is assigned to an unoccupied thread of the system. Thus, all threads of the system are continuously occupied by the stream of incoming documents that have to be filtered, as each incoming document is directed to the first available thread. On the other hand, the second approach RootPar, assigns all available threads on the filtering process of a single



Parameter	Value
Q_L	5
D_L	53
I_Q	$500K$
I_D	$50K$
DB	$3M$
Th	1

Figure 3.22: Comparing the effectiveness of Boolean and VSM models

document. This is achieved, by distributing the roots of the Forest to the available threads of the system. However, due to the splitting of the trie roots to different threads the filtering tasks are unevenly distributed between the assigned threads as a result of the word distribution and word order in the incoming document. This leads to fully utilizing only a fraction of the available threads for each incoming document while the rest of the threads may stay inactive for long periods of time.

3.6.7 Effectiveness Comparison

Figure 3.22 presents a proof-of-concept effectiveness evaluation and cross-comparison for the Boolean and VSM models. Our intention here is not to perform a full-scale study for the effectiveness of the two models, but rather to highlight that important publications are delivered to the users despite the crude nature of Boolean semantics.

To do so, we relied on the relevance judgments between queries and documents available at the TREC website, on official TREC tools (e.g., trec_eval), and the publicly available Lemur/Lucene libraries for parsing and preprocessing *ClueWeb09* WARC files. To derive the plots of Figure 3.22 we analyzed the notifications produced under each model, and set the cut-off threshold $\theta = 5$, $\theta = 50$ and $\theta = 500$ to consider the top-5, top-50 and top-500 most relevant notifications respectively. We used the typical (log normalization) tf.idf weighting scheme and vector normalization for VSM for both documents and queries (usually referred to as ltc.ltc [1]). For the Boolean model we utilized term frequencies (i.e., word counts) as a score function to make the notification lists of the two models comparable. Figure 3.22 presents the 11-point interpolated precision/recall graph, micro-averaged for the 686 TREC queries that had relevance judgments against 34013 different *ClueWeb09* documents.

Notice that the effectiveness between the two models is very similar when recall is low and comparable for high recall values, whereas an increase in θ results in better effectiveness. The reported Mean Average Precision values for $\theta = 5$ was 0.082, for $\theta = 50$ was 0.2763, and for $\theta = 500$ was 0.2972 for the Boolean model, and 0.0785, 0.2821, and 0.3055 for VSM respectively. Similarly, the reported NDCG values for $\theta = 5$, $\theta = 50$, and $\theta = 500$ were 0.1427, 0.4332, and 0.4695 (Boolean model) and 0.1380, 0.4508, and 0.4976 respectively (VSM). For more details on the effectiveness of the Boolean model and on how it compares against other alternatives the interested reader is referred to [115].

3.6.8 Summary of Results

Our extensive experimentation demonstrated the filtering efficiency of Algorithm StaR-HR when compared to the rest of the variants presented, as well as to other state-of-the-art algorithms. Algorithm StaR-HR achieves over 90% improvement in filtering time compared to state-of-the-art Algorithms ReTrie and Tree, while presenting low sensitivity to query database size, query length, and document size. Although Algorithm StaR-HR is designed for query databases that are unfocused and cover thematically a wide variety of topics, it performs well in terms of filtering time both for focused query databases with restricted vocabularies and real-life query logs. In addition, our experiments showed that Algorithm StaR-HR outperforms its competitors in terms of filtering time for various document sizes. Insertion and re-organisation times for StaR-HR are also efficient as it proves faster than its competitors due to the placement of rare words near trie roots. Finally, memory requirements for Algorithm StaR-HR are as much as 53% lower compared to all other examined algorithms, due to the delay in node creation; this strategy results in utilising each newly created node by as many queries as possible.

Overall, Algorithm StaR-HR is a versatile query reorganisation solution that outperforms competitors in demanding query clustering tasks, while presenting a steadily efficient performance in many versatile scenarios.

Limitations of the proposed family of algorithms include *(i)* reduced efficiency on limited query vocabularies and/or very short continuous queries, *(ii)* increased memory usage for indexing queries with disjunctions as the different disjuncts need to be split and indexed at different tries, and *(iii)* corpus-dependent parameter/algorithm setup.

3.7 Outlook and Future Directions

In this chapter, we identified different reorganization options and showed their effects on filtering efficiency. The presented variants of Algorithm StaR build upon the decoupling of query indexing and query insertion to (i) showcase the benefits and necessity of reorganization in the filtering performance and (ii) demonstrate the importance of word statistics incorporated appropriately in the reorganization process. As it was demonstrated, compact clustering is not always the best approach especially when words statistics are available. Thus, reorganization should be done in a careful and principled way, as false indexing objectives may lead in sub-standard filtering performance and lower than expected results. Finally, we investigated the usage of multi threaded solutions to enhance the filtering performance of the proposed Algorithm StaR. We identified and assessed two proof-of-concept parallelization scenarios, demonstrating the best approach to increase the filtering performance in our system.

Interesting directions for future research involve *(i)* the adaptation of automata/graph-based techniques as in [24, 50] to Boolean IF and their comparison against trie-based approaches, *(ii)* the extension of RDF-based data and SPARQL-based query models with text capabilities, and *(iii)* the construction of IF ontology systems that will be able to filter ontology data in a streaming fashion.

4
Efficient Continuous Multi-Query Processing over Textual and RDF Data

In the previous chapter, we dealt with the problem of efficiency in the context of *textual information filtering*. In this chapter, we focus on content-based information filtering systems, where we identify the importance of *efficient* and *effective information filtering*. To this end, we propose an *extension* of SPARQL with full-text operators, aiming at more expressive continuous queries that are able to support versatile user needs. Additionally, we propose efficient main-memory query indexing algorithms that support SPARQL queries with full-text constraints and are able to filter incoming publications in a few milliseconds. The results of this chapter have been published in [116].

The rest of this chapter is organized in six distinct sections. Section 4.1 provides the motivation for this work as well as the description of the research problems addressed. Section 4.2 presents the description of our data and query models employed to solve the problem at hand. Section 4.3 presents RTF, a family of algorithms developed to support the proposed data and query models, while focusing on efficiency. Section 4.4 presents Algorithm iBroker, a state-of-the-art solution, which serves as a baseline solution. Section 4.5 presents the experimental evaluation of the developed algorithms with a real-world data set. Finally, Section 4.6 concludes this chapter by discussing our findings, and presents future directions of our research.

4.1 Motivation

As the Web is growing continuously, a great amount of data is available to users, making it more difficult for them to discover interesting information by searching. For this reason, publish/subscribe (pub/sub) systems have emerged as a promising paradigm that enables the user to cope with the high rate of information production and avoid the cognitive overload of repeated searches. In a pub/sub system, users (or services that act on users' behalf) express their interests by submitting a continuous query and wait to be notified whenever a new event of interest occurs. The vast majority of modern pub/sub services and systems are typically content-based (contrary to previous decades, where they used to be topic/channel based); subscribers express their interest on the content of the publication (be it structure or data/text values) by appropriately specifying constraints in the submitted continuous queries.

In the early days of content-based pub/sub the structure of a publication was nothing more than a (usually static) collection of named attributes with values of different types (e.g., text) [4, 38]. As XML gained popularity and started becoming the standard for data/information representation and exchange on the web, various XML-based pub/sub systems have, naturally, arised [22, 23, 60–63, 117, 118]. In those systems, publications were expressed in XML and extensions of XPath/XQuery were used to express continuous queries. All research in the field focused mainly on the structural/value matching between (indexed) continuous queries and incoming publications, but has largely ignored semantics. This gave rise to ontology-based pub/sub systems [6–9] that typically used RDF [64] for representing publications and SPARQL [65] extensions/modifications for expressing user interests through continuous queries.

Ontology-based pub/sub systems research [6–9] has naturally focused more on semantics and has delivered interesting results. What it currently lacks, though, compared to the technological arsenal of the traditional pub/sub research is the support of a complete full-text retrieval mechanism, beyond existing regular expression and equality support, with sophisticated algorithms and data structures to minimize processing and memory requirements.

In this work, we initially propose an *extension* of SPARQL with full-text operators, aiming at more expressive continuous queries that are able to support versatile user needs in applications like digital libraries or news filtering. To preserve the expressiveness of SPARQL, we view the full-text operations as an additional filter of the query variables. In our setup, publications are ontology data that contain RDF literals in their property elements. A full-text expression is evaluated against a literal, and supported expressions involve the usual Boolean operators (i.e., conjunction, disjunction, negation), as well as word proximity and phrase matching as in Chang et al. [26]. To efficiently filter the incoming publications against the stored queries, we present RTF (acronym for RDF Text Filtering), a family of trie-based, main-memory, (continuous) query indexing algorithms that support SPARQL queries with full-text constraints and are able to filter incoming publications in a few milliseconds. We propose indexing methods that exploit the commonalities between continuous queries at indexing time and leverage on the natural properties of RDF during the filtering procedure. To the best of our knowledge, our family of algorithms is the first in the literature that is able to support SPARQL queries with full-text constraints. To demonstrate the efficiency of our approach we extend iBroker, developed by Park et al. [9], a state-of-the-art query indexing and RDF publication filtering algorithm, with full-text capabilities and compare it against our approach both on structural and full-text filtering tasks; our approach proves more than two orders of magnitude faster for the structural and more than one order of magnitude faster for the full-text filtering tasks.

In the light of the above, our contributions may be summarized as follows:

- We *extend* SPARQL with full-text operators and support Boolean, word proximity, and phrase matching operators.

- We *develop* a family of continuous query indexing algorithms that support full-text SPARQL queries and are able to filter the incoming RDF publications efficiently.

- We *extend* iBroker [9], a third party algorithm for ontology pub/sub, to offer full-text support and use it as a state-of-the-art competitor.

- We *identify* algorithmic alternatives for query indexing and assess their performance with a real-world data set against the extended version of iBroker.

In this section, we presented the motivation behind our work and gave a brief overview of the data model, our proposed SPARQL extension and our algorithmic solutions. In the following section, we present the data model and the proposed SPARQL extension we employed in our approach.

4.2 Data Model

In this section, we present the data model defined for the publication and profile model utilized in our system. Additionally, we present an *extension* of SPARQL with full-text operators, aiming at more expressive continuous queries that are able to support versatile user needs.

4.2.1 Profile and Data model

The Resource Description Framework (RDF) constitutes a conceptual model and a formal language for representing resources in the Semantic Web; it is the building block of a metadata layer on top of the current structured information layer of the *World Wide Web*, which enables interoperability between different systems and facilitate the exchange of machine-understandable information. Furthermore, the streaming fashion of RDF makes it a perfect candidate for modern pub/sub ontology systems, which demand sophisticated filtering mechanisms for matching massive ontology data against thousands of user profiles.

Constraint	::=	FTexpression
FTexpression	::=	ftcontains()
FTexpression	::=	FTmain ("ftand" FTmain)*
FTmain	::=	String

Table 4.1: The proposed SPARQL extension syntax.

The SPARQL query language is currently the W3C recommendation for querying the Semantic Web; the graph model, over which it operates, naturally joins data together and supports several query forms for querying RDF datasets. However, it still lacks the support of a complete full-text mechanism, which uses sophisticated algorithms and data structures to minimize processing load and memory requirements, for filtering purposes.

Since we focus our attention on full-text filtering of ontology data we are interested only in property elements with a plain RDF literal as their content. A literal in an RDF graph can be either plain or typed. Plain literals have a lexical form and an optional language tag, whereas typed literals have a lexical form and a datatype URI. In this context, the subject of an RDF triple is always a node element and the predicate denotes the relation to the literal. The object is the literal, which is expressed as a string.

The equivalent extension for XML publications [119] uses a concrete boolean model to capture the semantics of full text querying, where the atoms of the model are decomposed into basic queries representing both single word and context queries.

Profile Model

In the spirit of [119], we propose an extension to the SPARQL syntax to support full-text continuous queries in RDF datasets. To preserve SPARQL expressibility we view the full-text operations as an additional filter of the (continuous) query variables. Notice that SPARQL supports different query forms that affect only the form of the answers returned and not the graph matching process itself. In this context, we define a new binary operator *ftcontains* (full-text contains), that takes as input a *variable* of the continuous SPARQL query and a *full-text expression* that operates on the values of this variable. The query signature of the operator is expressed as the following function:

$$xsd : boolean : ftcontains(var, ft_{expression})$$

In this context, a full-text expression is evaluated only against a literal, so *var* is always the object of the SPARQL tuple pattern; the subject and/or predicate of the tuple pattern may be constants. The expressions supported involve the usual Boolean operators (denoted by ftAND, ftOR, etc.), as well as proximity (denoted by ftNEAR) and phrase matching as in [26]. The rules for the extended SPARQL syntax are listed in Table 4.1. To this end, we carefully designed a new set of full-text queries which currently can not be efficiently evaluated by existing pub/sub ontology systems.

In Figure 4.1, we present a set of SPARQL continuous queries that utilize the proposed extended syntax. SPARQL Query 1 will match all publications that are of type *article* and have an attribute *title* with a string literal. The title of the publications must contain the terms "olympic" and "games". Additionally, the publications that match must have an attribute *body* that contains the terms "olympic", "games" and "rio", and the term "rio" is at least 0 and at most 2 words after the term "games" (due to the word proximity constraint).

In the same spirit, SPARQL Query 2, presented in Figure 4.1, requests for publications that are of type *article*. The publication must have an attribute *title* that contains the term "olympic", an attribute *abstract* that contains the terms "olympic" and "rio". Additionally, the same publication must bare a *body* attribute that contains the terms "olympic" and "committee". Finally the publication must have a *publisher* with a *name* that includes the terms "the", "wall", "street" and "journal" at any given order.

Users may be unfamiliar with the structure of publications to be made available in a system, when submitting a continuous query. Thus, applying specific restrictions to the publications may result in missing notifications that may match a users' interest. The need for a more flexible subscription scheme seems necessary in order

SPARQL Query 1

```
SELECT ?publication
WHERE {?publication type article.
        ?publication title ?title.
        ?publication body ?body.
FILTER ftcontains(?title, "olympic" ftAND "games")
FILTER ftcontains(?body, "olympic" ftAND "games"
   ftNEAR[0,2] "rio")
}
```

SPARQL Query 2

```
SELECT ?publication
WHERE {?publication type article.
       ?publication title ?title.
       ?publication abstract ?abstract.
       ?publication body ?body.
       ?publication publisher ?publisher.
       ?publisher name ?name.
FILTER ftcontains(?title, "olympic")
FILTER ftcontains(?abstract, "olympic" ftAND "rio")
FILTER ftcontains(?body, "olympic" ftAND "committee")
FILTER ftcontains(?name, "the" ftAND "wall" ftAND "
   street" ftAND "journal")
}
```

SPARQL Query 3

```
SELECT ?publication
WHERE {?publication type *.
       ?publication title ?title.
       ?publication body ?body.
FILTER ftcontains(?title, "olympic" ftAND "committee"
   ftAND "president")
FILTER ftcontains(?body, "olympic" ftAND "rio" ftAND "
   stadium")
}
```

Figure 4.1: SPARQL Queries presenting the proposed extended syntax.

to better describe users' interests. Additionally when less restrictions are applied to the published data a better quality of information delivery can be achieved. This flexibility results to better content delivery as the description of interests is more expressive and effective. Thus, in addition to the full-text extension of SPARQL we also support the *wildcard (*)* operator applied in RDF triples, i.e., queries where the subject, predicate and/or object of a triple may match any value of the publication. Such a combination of full-text and wildcard operations allows us to offer the users a rich set of tools that allow them to specify expressive continuous queries that will match their information needs.

An example query, where the wildcard operator is utilized, is given in SPARQL Query 3 (Figure 4.1). In this example, publications can be of any type. While, the publication must have an attribute title that contains the terms "olympic" and "committee" and an attribute body that contains the terms "olympic", "rio" and "stadium".

4.2.2 The Publication Model

In the context of our system, a publication is defined as a set of triples that are expressed using RDF/XML [120]. Hence, the underlying model is a directed graph which contains a set of nodes that may serve as the subject or the object in a triple statement. The nodes are connected via properties that are expressed as the *predicate* in the triple statement.

In Figure 4.2 we present an example of an incoming publication that arrives in our system. Please observe, that the publication is represented in RDF form. The publication concerns the olympic games of 2016 in Rio, it bears three distinct attributes, namely *type*, *title* and *abstract*, while the title and abstract attributes bare a text field.

In our information filtering model, we utilize the RDF data language to describe the data publications. A publication *pub* is described in a structured manner using RDF-triples containing additional fields where needed to store the text parts. The usage of RDF data language renders our system flexible to publication input. The

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dbp="http://dbpedia.org/property/"
  xmlns:dbo="http://dbpedia.org/ontology/">
  <rdf:Description rdf:about="http://dbpedia.org/resource/2016_Summer_Olympics">
    <rdf:type rdf:resource="http://schema.org/Article"/>
    <dbp:title rdf:datatype="http://www.w3.org/1999/02/22-rdf-syntax-ns#langString">2016 Summer Olympic Games</dbp:title>
    <dbo:abstract xml:lang="en">The 2016 Summer Olympics (Portuguese: Jogos Olímpicos de Verão de 2016), officially known as the Games of the XXXI Olympiad and commonly known as Rio 2016, was a major international multi-sport event held in Rio de Janeiro, Brazil, from 5 August to 21 August 2016. More than...</dbo:abstract>
  </rdf:Description>
</rdf:RDF>

```

Figure 4.2: An example of an incoming publication.

information filtering algorithms, presented in the following section, can match an incoming publication expressed into a RDF-structured manner, against the profile database by translating each publication into a series of RDF-triples.

Definition 4.1 *We define a publication pub as a series of RDF-triples (also known as statements). Each triple has three attributes, namely* subject *($\mathcal{S}$),* predicate *($\mathcal{P}$) and* object *($\mathcal{O}$). In this context, the object of a triple $\mathcal{O}_i \in t_i$ may have a property value, and more specifically in our approach a text field. Alternatively, the value of $\mathcal{O}_i$ can be another resource. To this end, a publication in our system is represented as follows:*

$$pub = t_1(\mathcal{S}_1, \mathcal{P}_1, \mathcal{O}_1) \wedge t_2(\mathcal{S}_1, \mathcal{P}_2, \mathcal{O}_2) \wedge ... \wedge t_n(\mathcal{S}_1, \mathcal{P}_n, \mathcal{O}_n)$$

Example 4.1 *By applying Definition 4.1, on the publication presented in Figure 4.2 we receive the following set of three distinct triples:*

`pub =` t_1`(2016_Summer_Olympics, type, Article)` $\wedge$

t_2`(2016_Summer_Olympics, title, "2016 Summer Olympic Games")` $\wedge$

t_3`(2016_Summer_Olympics, abstract, "The 2016 Summer Olympics...")`

Please note that for presentation purposes, the URIs, properties and property values have been shortened.

In this section, we described the data model and a full-text extension on SPARQL adopted in our approach. In the following sections, we describe the algorithmic solutions developed to solve the information filtering problem.

4.3 Query Indexing Algorithms

In this section, we present RTF, a family of query indexing algorithms that utilise trie structures to exploit commonalities between continuous queries to achieve faster filtering times. Initially, we elaborate on the indexing algorithm RTFM[1](Section 4.3.1), discuss its variation RTFS (Section 4.3.2), and provide details for the common filtering procedure (Section 4.3.3).

4.3.1 Algorithm RTFM

Algorithm RTFM is indexing each continuous query by executing the following three steps:

1. Transforming the continuous query to conjunctions of tuples (quadruples or triples depending on the existence of a text constraint or not) and assigning a unique identifier to each tuple.

2. Registering all the discrete tuples produced from the previous step in a table that associates each continuous query with the tuple identifiers it contains.

3. Indexing of all the query tuples at the trie structure described below.

[1] No connection to the infamous initialism – https://en.wikipedia.org/wiki/RTFM

In the following, we analyze each step and provide details on the data structures and algorithms utilized. The pseudocode for the indexing phase of Algorithm RTFM is performed by Algorithm INDEX and is presented in Figure 4.4.

Step 1: Tuple Representation

Users express their interests in a defined manner by making use of the SPARQL query language and the full-text extension we supply. While Algorithm RTFM operates on tuples; in this section we define continuous queries as conjunctions of tuples, and, in the following sections, we illustrate how we exploit commonalities between those tuples to achieve better query indexing and thus faster filtering performance.

SPARQL is a standardized language that users (or service that act on behalf of users) can employ to define their interests. Nevertheless, SPARQL is a querying language that can be transformed in a series of conjuncts by making use of RDF triples. This transformation in RDF triples provides us with great advantages, as a SPARQL query can retain its restrictions, while being easier to manage and index each individual triple efficiently. Additionally, full-text constraints in SPARQL queries can be applied in every triple by employing an additional field that contains these constraints. To this end, we provide below the formal definition for a continuous query that is utilized in our approach.

Definition 4.2 *We define a continuous query q as a series of i, where $i \in 1, \ldots, n$ tuple conjuncts. Each tuple has three* mandatory *attributes, namely* subject *($\mathcal{S}_i$),* predicate *($\mathcal{P}_i$) and* object *($\mathcal{O}_i$). There is an additional, non-mandatory, attribute $\mathcal{F}_i$ that facilitates the representation of the full-text operators and their textual constraints. Thus, a continuous query may be represented as:*

$$q = t_1(\mathcal{S}_1, \mathcal{P}_1, \mathcal{O}_1\{, \mathcal{F}_1\}) \wedge \cdots \wedge t_n(\mathcal{S}_n, \mathcal{P}_n, \mathcal{O}_n\{, \mathcal{F}_n\})$$

Example 4.2 *By applying Definition 4.2 to the continuous SPARQL Query 1 (q_1) in the example of Figure 4.1 we receive the following set of tuples:*

q_1 = (?publication, type, article) $\wedge$

(?publication, title, ?title, ftcontains("olympic" ftAND "games")) $\wedge$
(?publication, body, ?body, ftcontains("olympic" ftAND "games" ftNEAR$_{[0,2]}$ *"rio"))*

Moving from a SPARQL query to a tuple-based representation is achieved by applying appropriate parsing on the continuous query with a tool like Sesame[2].

Example 4.3 *In the same manner as in Example 4.2, SPARQL Query 2 (*q_2*) (Figure 4.1) is processed into a set of tuple conjuncts:*
q_2 *= (?publication, type, article)* $\wedge$
(?publication, title, ?title, ftcontains("olympic")) $\wedge$
(?publication, abstract, ?abstract, ftcontains("olympic" ftAND "rio")) $\wedge$
(?publication, body, ?body, ftcontains("olympic" ftAND "committee")) $\wedge$
(?publication, publisher, ?publisher) $\wedge$
(?publisher, name, ?name,
ftcontains("the" ftAND "wall" ftAND "street" ftAND "journal"))

Example 4.4 *In the same manner as in Example 4.2, SPARQL Query 3 (*q_3*) (Figure 4.1) is processed into a set of tuple conjuncts:*
q_3 *= (?publication, type, *)* $\wedge$
(?publication, title, ?title,
ftcontains("olympic" ftAND "commitee" ftAND "president")) $\wedge$
(?publication, body, ?body, ftcontains("olympic" ftAND "rio" ftAND "stadium"))

In this step, we have demonstrated how we move from a SPARQL-based query representation to a tuple-based (Figure 4.4, Line 1). This query transformation enables us to have more flexibility in the way queries are handled, indexed and eventually answered during filtering time. In the next two steps, we present how Algorithm RTFm handles the tuples generated in this step.

[2] http://rdf4j.org

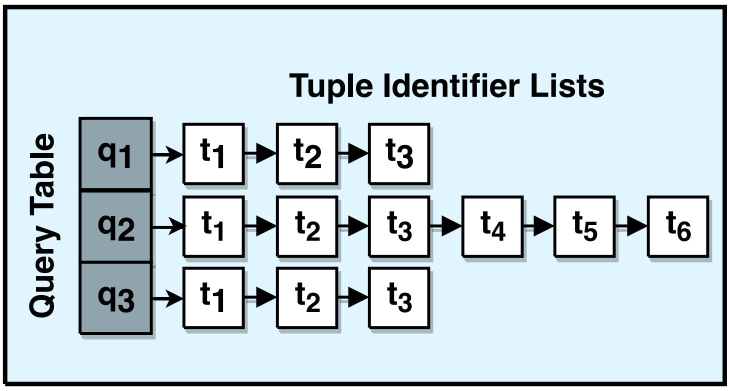


Figure 4.3: Query Table QT after during the indexing phase of RTFM.

Step 2: Associating Queries With Tuple Identifiers

Following Step 1, Algorithm RTFM receives a query q that consists of two fields, a unique *query identifier* and a set of tuples, that denote the constraints posed by the query. During the indexing phase of q, Algorithm RTF assigns every tuple that forms q a unique identifier $t_s...t_n$. The identifiers $t_s...t_n$ serve as a link between every query and its corresponding set of tuples. In continuation, Algorithm RTFM proceeds by storing each continuous query along with the tuple identifiers into the Query Table (QT). QT is comprised of two fields: the unique identifier of each query q and a linked list that stores the unique identifiers of the continuous query tuples (Figure 4.4, Lines 2-3). For instance, for the continuous SPARQL Query 1 (Figure 4.1), Algorithm RTFM will add three tuple identifiers into QT (as they are shown in the previous section). RTFM proceeds in a similar way to insert every new continuous query that is submitted in QT.

Figure 4.3 presents the query table QT indexing of all three continuous queries shown in Figure 4.1. The first cell of QT stores the identifier q_1, coupled with a linked list that contains the unique tuple identifiers $\{t_1, t_2, t_3\}$ generated in the second step of Algorithm RTFM. In the same manner, the rest of the cells of QT represent the indexing of queries q_2 and q_3 of Figure 4.1.

```
Algorithm: INDEX
Input: A query $q_i$
Result: Store $q_i$
// Extract a set of tuples from incoming query $q_i$
$\{t_1, \dots, t_n\} \leftarrow extractTuples(q_i)$;
// Assign a unique identifier to $t_i$ and index it to $QT$
foreach $t_i \in \{t_1, \dots, t_n\}$ do
    $QT[id(q_i)] \leftarrow QT[id(q_i)] \cup id(t_i)$;
$position \leftarrow$ Null;
// For all tuples in SPARQL Query $q$
foreach $t_i \in \{t_1, \dots, t_n\}$ do
    // For each field $f_l$ of $t_i$
    foreach $f_i \in t_i$ do
        $n_k \leftarrow DFS(forest)$;
        // If a node $n_k$ is found that can index $f_l$
        if $content(n_k) = f_i$ then
            // Store the node's position
            $position \leftarrow n_k$;
    // If no indexing position has been found
    if $position = NULL$ then
        // Create nodes $n_s$, $n_p$, $n_o$ for the subject, predicate and
            object of $t_i$ respectively
        $content(n_s) \leftarrow subj(t_i)$;
        $content(n_p) \leftarrow pred(t_i)$;
        $content(n_o) \leftarrow obj(t_i)$;
        // Create a new tree with children nodes $n_s$, $n_p$ and $n_o$
        create tree $T'$ with $root(T') = n_s$;
        $children(n_s) \leftarrow n_p$;
        $children(n_p) \leftarrow n_o$;
    else
        // For all remaining fields of $t_i$ create new nodes to index
            them
        foreach $f_r \in t_i$ do
            $content(n_j) \leftarrow f_r$;
            // Assign each new node to the previous one as child
            $children(position) \leftarrow n_j$;
            $position \leftarrow n_j$;
    // If tuple $t_i$ contains a full-text operator
    if $t_i$ has $FT$ then
        $ftIDs(n_o) \leftarrow id(t_i)$;
        indexText($t_i$);
    else // Else store $t_i$ in the tuples list
        $tIDs(n_o) \leftarrow t_i$;
```

Figure 4.4: Pseudocode for the query indexing phase (Algorithm INDEX) performed by Algorithm RTFM.

Step 3: Indexing Tuples in the Trie Forest

The *trie forest* is populated in order to store the tuples compactly by exploiting their common elements. Thus, every trie forest consists of a collection of tries, which in turn contain a number of trie nodes; in each node N the following information is stored:

- The node content, denoted by *content*(N), that may represent either an RDF attribute/variable or a word contained in a text constraint of a query.
- The list of children nodes of N, denoted *children*(N).
- The list of tuple identifiers, denoted by *tIDs*(N), that are indexed under N.

When a new query q_i arrives AlgorithmRTFm iterates through the set of all query tuples and indexes every tuple in the trie forest. During the indexing phase RTFm searches the trie forest for a suitable place to index each tuple as follows (Figure 4.4, Lines 6-9).

The first tuple of the first continuous query that is submitted will naturally arrive in an empty trie forest and will create a number of nodes that depend on the form of the tuple (Figure 4.4, Lines 10-21) . Specifically, for the structural constraints of the tuple, RTFm creates three new trie nodes one for each attribute specified (Figure 4.4, Lines 11-16). If the tuple contains also a full-text constraint with k distinct words, RTFm will create k more nodes (one for each distinct word) (Figure 4.4, Lines 22-26). For illustration purposes, we use a pseudo-node “FT” to separate the structural from the word constraints and highlight the difference between the different RTF variants (i.e., RTFm discussed here and RTFs discussed later in the section).

In general, when inserting a new tuple, RTFm considers storing it at an existing trie or creating a new trie. To insert a new tuple $t(\mathcal{S}, \mathcal{P}, \mathcal{O})$, RTFm examines the subject $\mathcal{S}$ of the tuple and utilizes the trie structure to find if there is a candidate trie which has a root node R such that *content*$(R) = \mathcal{S}$ (Figure 4.4, Lines 6-9). If such a trie is found, the indexing algorithm proceeds to examine *children*(R) in order to determine if there is a child C such that *content*$(C) = \mathcal{P}$. The same applies

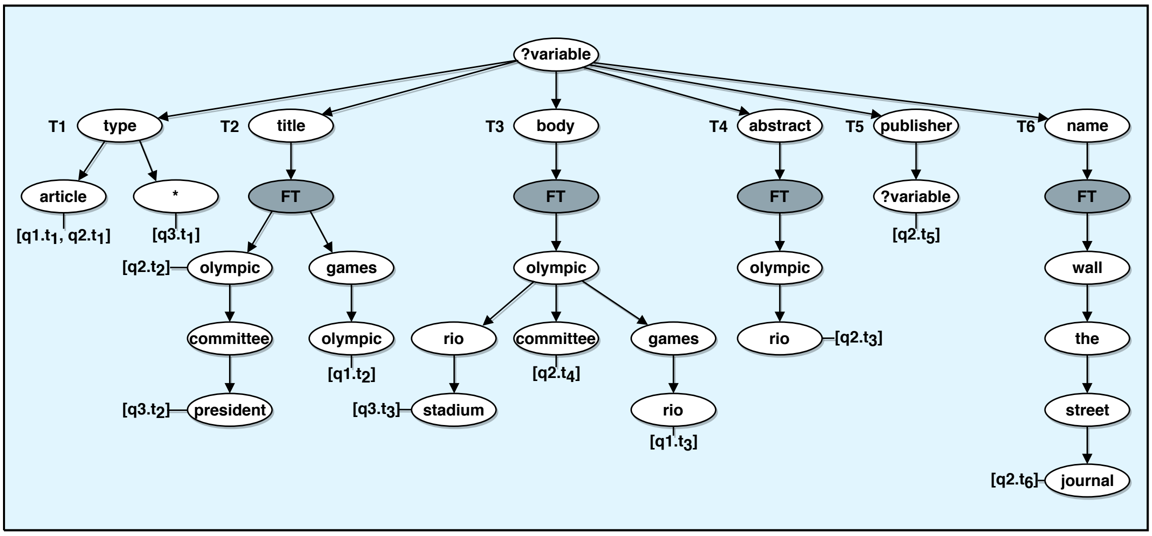


Figure 4.5: Trie forest during the indexing phase of RTFm.

for the object $\mathcal{O}$ of the tuple. Notice that variables (in subject/predicate/object) and wildcards in tuples are mapped onto the corresponding variable/wildcard nodes. If the new tuple contains full-text constraints, the trie is expanded with the distinct words contained in these tuple constraints in a similar manner.

If, during the indexing phase, RTFm fails to locate an appropriate trie position to store a new tuple, it proceeds in creating a new set of nodes that will index the remaining tuple fields (Figure 4.4, Lines 10-21). After locating (or creating) the appropriate trie that will store a tuple t, RTFm stores also the tuple id at $tIDs(N)$ of node N of this trie, so as to be able to identify the tuple at filtering time. Notice that different query insertion order will, naturally, give different tries, since query organization is greedy, and depends on the already stored queries.

Indexing of proximity formulas and phrases in the trie forest of RTFm is performed in the same way as described above, since proximity is a more constrained case of conjunction. To accommodate the word distance in the proximity/phrase expression, we use an extra data structure that stores the proximity constraints in the spirit of [4]. Disjunctions are handled by creating separate queries (that have the same user as the notification recipient) for the different word operands.

Figure 4.5 shows the resulting trie after inserting three continuous queries, q_1, q_2 and q_3 of Figure 4.1. Additionally, the three tuples of q_1 (shown in Example 4.2

above) are assigned ids $q1.t_1, q1.t_2, q1.t_3$ respectively. From the indexing performed by RTFM in these queries notice that:

- Query q_1 shares the same tuple (`(?publication, type, article)`) with query q_2, as two different tuple identifiers (namely $q1.t_1$ and $q2.t_1$) are stored in the same leaf node. Moreover, this tuple contains only structural constraints.
- Query q_1 contains tuple $q1.t_3$ that specifies both structural and full-text constraints.
- Query q_1 (with tuple $q1.t_3$) shares the same structural constraints and also has the word `"olympic"` in common for the textual constraints part with query $q2$ (with tuple $q2.t_4$).
- Query q_3 contain a tuples with a wildcard operator ($q3.t_1$).

Finally, note that Figure 4.5 shows just one of the tries that would be created; typically, because of different query structure the resulting indexing structure is a *forest of tries*. Thus, a hash table (not shown in Figure 4.5 to avoid cluttering) is used to provide fast access to trie roots.

4.3.2 Algorithm RTFs

Indexing of word constraints in the context of RTF may be performed in two different ways: (i) using *multiple* tries (hence the name RTFM) for indexing the word constraints depending on the structural part of the continuous queries as described in the previous section (and shown in Figure 4.5), and (ii) using a *single* trie forest (hence the name RTFS) that is dedicated to all text components, regardless of the structural part of the continuous queries shown in Figure 4.6.

In the former case, namely Algorithm RTFM, the textual constraints are considered as a natural expansion of the structural ones, but there exist fewer clustering opportunities for words. Contrary, in the latter case, namely Algorithm RTFS, the word constraints are considered as a different type of constraint and are clustered together regardless of the structural constraints of the query.

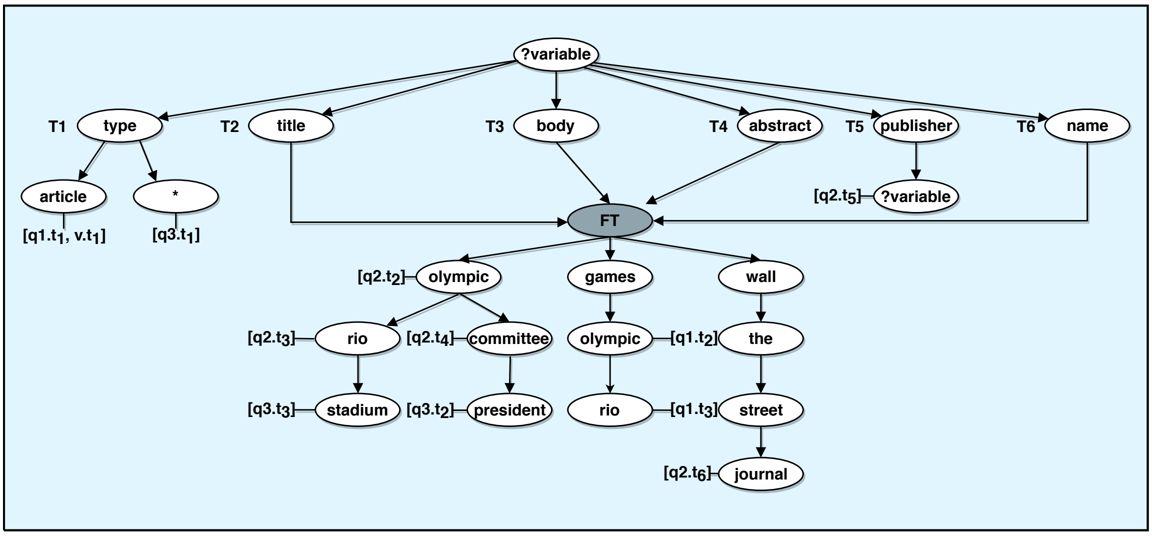


Figure 4.6: Trie forest during the indexing phase of RTFs.

Algorithm RTFs is a variation that allows us to construct a more compact forest of tries since this organization creates more clustering opportunities for the words. Notice that the trie of Figure 4.6 has less nodes compared to that of Figure 4.5 for the same queries and the same query insertion order. As we will demonstrate in Section 4.5, Algorithm RTFs is better suited for cases where queries with text constraints are relatively sparse, whereas Algorithm RTFm is better suited for cases where many queries contain text constraints.

4.3.3 Filtering algorithm

The filtering algorithm is common for the two variants (RTFm and RTFs) of Algorithm RTF. In this section, we present the filtering algorithm that allows RTF to filter incoming RDF publications and issue notifications to subscribed users. The pseudocode for the filtering phase of Algorithm RTFm is performed by Algorithm Filter and is presented in Figure 4.7.

The filtering process operates on triples; new RDF publications are parsed and transformed to a set of triples that are subsequently used to guide the traversal of the trie forest in search for matching continuous queries.

Definition 4.3 *We define a publication p as a series of i, where $i \in 1, \ldots, n$ conjuncts of RDF triples. Each triple has three attributes, namely* subject *$(\mathcal{S}_i)$,*

```
   Algorithm: Filter
   Input: A publication $p$
   Output: A list of queries $match = \{q_i, \ldots, q_j\}$
1  Procedure filterPublication(p)
2      matchedTuples ← Null
3      foreach tuple t in publication p do
4          foreach trie root R do
5              traverseTrie(R,t)
6      isSatisfied ← TRUE
7      foreach query q in QT do
8          foreach tuple $t \in q$ do
9              if t is not marked as matched then
10                 isSatisfied ← FALSE
11                 break
12         if isSatisfied==TRUE then
13             notify subscriber
14 Procedure traverseTrie(node N, tuple t)
15     if content(N) is satisfied by t then
16         matchedTuples ← matchedTuples ∪ tIDs(N)
17         traverseTrie(C,t), where $C \in children(N)$
```

Figure 4.7: Pseudocode for the publication filtering (Algorithm Filter) performed by Algorithms RTFm and RTFs.

predicate *($\mathcal{P}_i$) and* object *($\mathcal{O}_i$) or* text field *($\mathcal{T}_i$) that represents the textual content of an attribute. Thus, a publication may be represented as:*

$$p = t_1(\mathcal{S}_1, \mathcal{P}_1, \mathcal{O}_1|\mathcal{T}_1) \wedge \cdots \wedge t_n(\mathcal{S}_n, \mathcal{P}_n, \mathcal{O}_n|\mathcal{T}_n)$$

The filtering process proceeds as follows. For every triple $t(\mathcal{S}, \mathcal{P}, \mathcal{O})$, in the newly arrived publication p, the trie forest is examined and the root R for which *content*$(R) = \mathcal{S}$ is visited (Figure 4.7, Lines 3 - 5). Thereafter, RTF begins traversing the trie in a depth first manner and examines the nodes *children*(R) in order to determine if there are matching tuples. In order to reach from the root node R to a leaf node, every node N in the path must fulfill the following requirements: *content*$(N) = \mathcal{P}$ and *content*$(N) = \mathcal{O}$, or *content*$(N) = \$variable$. If, at any point of the trie traversal a node N with a wildcard field ($content(N) = *$) is visited the traversal continues to $children(N)$, as this is considered a match for N (Figure 4.7, Lines 14 - 17). The traversal of the trie finishes when a leaf is reached.

For every triple $t(\mathcal{S}, \mathcal{P}, \mathcal{T})$, in the newly arrived publication p, the trie forest is examined as above. When a node N that represents a word constraint is visited, the traversal continues as follows. For every node $C, C \in children(N)$, for which $content(C)$ is contained in $\mathcal{T}$ the sub-trie that has C as a root is examined in a depth-first manner. The traversal of the trie continues recursively for as long as common words between the children of a visited node and $\mathcal{T}$ exist.

Notice that, independently of the structural or full-text constraints, the $tIDs(N)$ list at each node N gives implicitly all query tuples that match the incoming publication tuple. Thus, all $tIDs(N)$ of all traversed trie nodes are marked as matched in QT. Word distance constraints in phrase/proximity operations are checked for satisfaction after the trie traversal. In the end of the processing of publication p (i.e., after processing all its tuples), a scan of QT allows us to determine the queries that have matched the incoming publication.

In this section, we presented a family of query indexing algorithms that utilize trie structures to capture common elements between continuous queries, namely Algorithms RTFm and RTFs. In the following section, we present Algorithm iBroker, a state-of-the-art competitor that uses an inverted index to store submitted SPARQL queries.

4.4 Algorithm iBroker

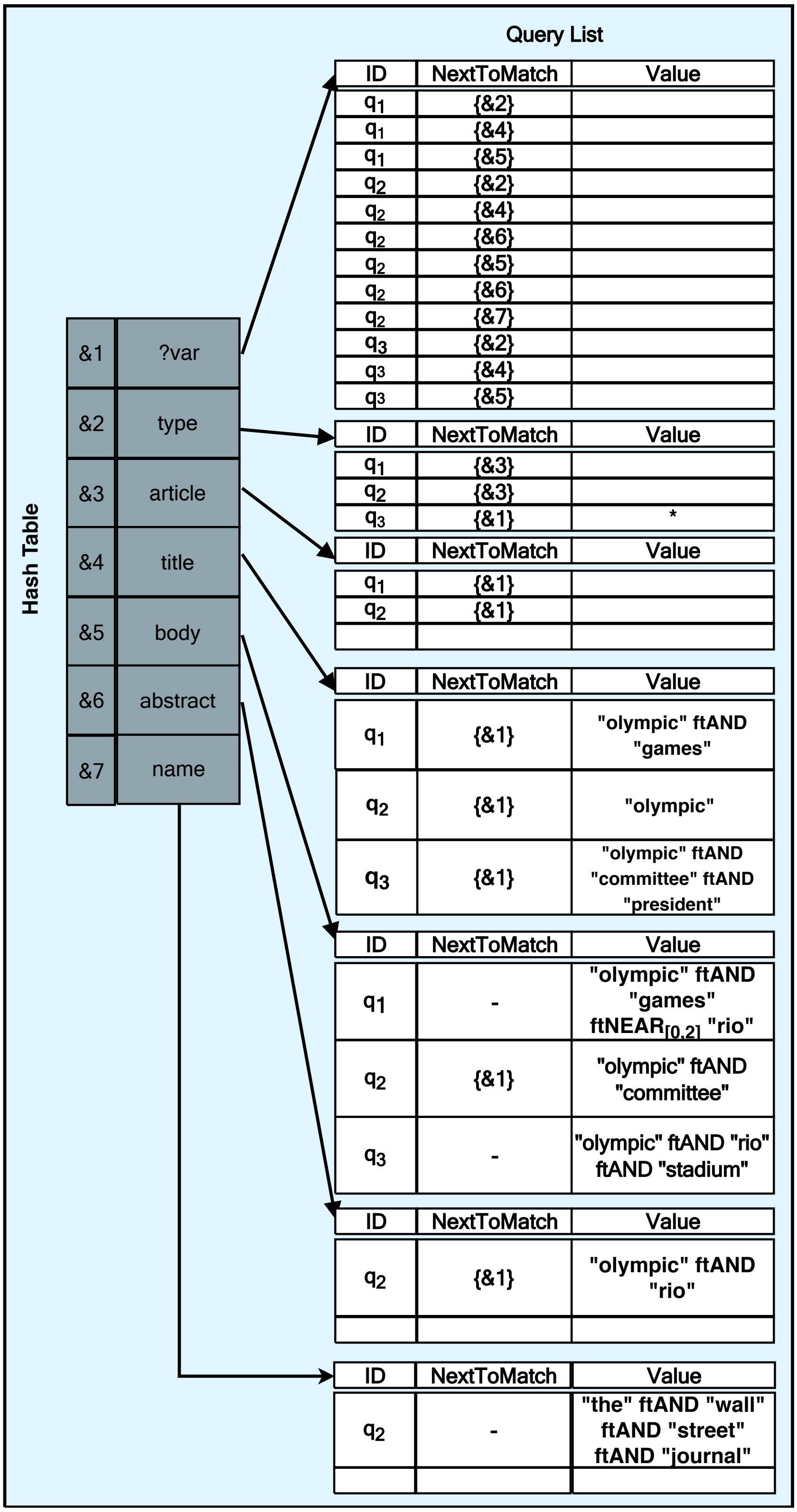
In this section, we present Algorithm iBroker [9], a state-of-the-art competitor that uses an inverted index to store submitted SPARQL queries. Additionally, we describe its extension with full-text capabilities.

In order to evaluate the efficiency of RTF we have implemented iBroker [9] as a *baseline competitor*. Algorithm iBroker is a continuous query indexing algorithm that supports SPARQL queries with structural and string matching constraints, and is currently *the only* state-of-the-art algorithm that is able to handle RDF queries with both structure and (some form of) text. In this section, we present the basic idea behind iBroker and the data structures upon which it operates and show how we extended its functionality to support full-text constraints.

Algorithm iBroker utilizes an inverted index to store the continuous queries. Its indexing structure consists of a hash table that is used to index the unique attributes of all triples that correspond to the submitted SPARQL queries. iBroker uses the unique attribute names as hash keys to access the corresponding hash buckets, and each hash bucket contains references to lists of stored queries. These lists store: (i) the unique identifier ID of query q, (ii) a reference to a hash bucket, named *NextToMatch*, that contains the next attribute in q, (iii) the string that might be present in q named *Value*, and (iv) any possible variables in q. Figure 4.8 presents the inverted index structure of iBroker after indexing queries q_1, q_2 and q_3 presented in Figure 4.1.

This inverted index stores the queries in a chain-like manner. Every query may be recomposed by following the *NextToMatch* references to hash buckets until an empty *NextToMatch* field is visited. This procedure is applied by the algorithm iBroker during the filtering of a publication event. As there is no defined hierarchy that outlines the filtering sequence, an incoming publication may need to examine many hash buckets looking for the beginning of a query. The result is that iBroker must, in this case, examine all the continuous queries in the corresponding bucket list, and then proceed to examine their *NextToMatch* entries until there is none left to match.

In the case that iBroker starts from fields that belong to a profile p, but, are not part of the first triples that form it, we can not be sure if it's a complete or partial match for the profile. Subsequently, without additional matching techniques the query q can not be determined as positive, resulting into false positives for iBroker. This problem can be solved by utilizing the query table QT as described in Section 4.3.1. As the RDF triples that form a query q are assigned unique identifiers and indexed in QT create a streamlined sequence for every triple identifier, thus, each triple identifier must match sequentially in order to satisfy a query. By enhancing Algorithms iBroker with this technique we have achieved to eliminate



&1	?var
&2	type
&3	article
&4	title
&5	body
&6	abstract
&7	name

ID	NextToMatch	Value
q_1	{&2}	
q_1	{&4}	
q_1	{&5}	
q_2	{&2}	
q_2	{&4}	
q_2	{&6}	
q_2	{&5}	
q_2	{&6}	
q_2	{&7}	
q_3	{&2}	
q_3	{&4}	
q_3	{&5}	

ID	NextToMatch	Value
q_1	{&3}	
q_2	{&3}	
q_3	{&1}	*

ID	NextToMatch	Value
q_1	{&1}	
q_2	{&1}	

ID	NextToMatch	Value
q_1	{&1}	"olympic" ftAND "games"
q_2	{&1}	"olympic"
q_3	{&1}	"olympic" ftAND "committee" ftAND "president"

ID	NextToMatch	Value
q_1	-	"olympic" ftAND "games" ftNEAR$_{[0,2]}$ "rio"
q_2	{&1}	"olympic" ftAND "committee"
q_3	-	"olympic" ftAND "rio" ftAND "stadium"

ID	NextToMatch	Value
q_2	{&1}	"olympic" ftAND "rio"

ID	NextToMatch	Value
q_2	-	"the" ftAND "wall" ftAND "street" ftAND "journal"

Figure 4.8: Inverted index structure during the indexing phase of Algorithm iBroker.

the majority of false positives, but, extreme cases where the first triple of a profile matches partially can not be solved efficiently and are out of the scope of our research.

Finally, Algorithm iBroker implements string matching, but has no support for full-text operators. To evaluate it against RTF we have extended iBroker with full-text subscriptions by replacing the *Value* field with the list of words that appear in a full-text constraint. This modification enables iBroker to support both string and full-text constraints. Finally, for comparison purposes, we have also extended the functionality of iBroker to index and filter SPARQL queries that contain wildcard operators. For a more detailed description of iBroker and the specifics of its algorithms the interested reader is referred to [9].

In this section, we presented Algorithm iBroker, a solution in the literature that supports continuous query answering. To this end, we have extended iBroker to support our proposed full-text extension, while we aimed at addressing its design problem of false positive generation. In the following section, we present the experimental evaluation results when comparing iBroker against Algorithms RTFm and RTFs described in Section 4.3.1.

4.5 Experimental Evaluation

In this section, we present a series of experiments that compare RTF against iBroker under a series of different scenarios, since in a pub/sub scenario we investigate the behavior of algorithms when varying the profile set. By capturing and studying the most common scenarios that emerge in the pub/sub paradigm, we achieve a better understanding of the algorithms' performance. To this end, we designed and simulated three distinct experimental scenarios, which we present in the following sections.

4.5.1 Experimental Setup

In this section, we present the data sets we choose for the experimental evaluation as well as we elaborate on the generations of query sets. Additionally, we present the underlying algorithmic and technical configuration. Finally, we present the metrics under which, the experimental evaluation was designed and executed.

Data and Query Sets

For the experimental evaluation we utilized, the *DBpedia* corpus[3] extracted from the *Wikipedia* domain that forms a structured knowledge database of more than 4 million items. A major part, namely 3.22 millions publications, of the *DBpedia* corpus has been classified into an ontology resulting to 529 different classes which are described by 2.3 thousand properties. Additionally, publications bare textual information that originates from human generated content published at the *Wikipedia* domain. These characteristics give an additional advantage to the *DBpedia* corpus as it provides a solid source of textual information.

The vocabulary extracted from the *DBpedia* publications consists of 3.14 millions unique words. The maximum textual information present in a publication is $14,254$ words, while the average is 53 words. The diversity in content of the *DBpedia* corpus, accompanied with the information on structural and textual level, renders it as the perfect candidate for evaluating our algorithms in terms of indexing and filtering efficiency.

Table 4.2 summarizes some key characteristics of the *DBpedia* corpus. The data collected in the DBpedia corpus cover a wide variety of topics. This diversity in topics and structural data is providing a plethora of publications that are represented in a RDF manner making DBpedia corpus the perfect candidate for evaluating our algorithms. As we need to evaluate our algorithms both on their structural and text matching capabilities, we utilise textual information provided by DBpedia.

Query Set

The queries were constructed by utilizing classes and properties extracted from the *DBpedia* corpus. Each query, contains at most 4 tuples. The query set is formed by sets of tuples containing full-text operators with probabilities of FT_{pr} = 0%, 50% or 100%. The full-text operators contain conjunctive terms

[3] `http://dbpedia.org/Downloads2015-04`

Description	Value
Items	$4M$
Classes	529
Properties	$2,333$
Average publication size (words)	53
Maximum publication size (words)	$14,425$
Minimum publication size (words)	1

Table 4.2: Characteristics of the DBpedia corpus.

that are selected equiprobably among the multi-set of words from the *DBpedia* vocabulary. Additionally, the full-text operators contain at most 3 terms. Queries with $FT_{pr} = 0\%$ examine the performance of the algorithms for structural matching only. For queries with $FT_{pr} = 50\%$, we examine a mixed filtering scenario where half of the queries contain also textual constraints apart from the structural ones. Finally, for queries with $FT_{pr} = 100\%$, we demonstrate the scaling capabilities of the algorithms as all the indexed queries contain full-text constraints.

The Main Query Collection. More specifically, the *main query collection*, where $FT_{pr} = 50\%$, aims at evaluating the algorithms under an average filtering scenario, where the database is filled with profiles consisting of approximately the same percentage of structural and textual restrictions. A subscription usually bears more structural constraints in order to focus on specific types of publications, while the textual constraints refine the publication selection by utilizing a small amount of keywords.

The Second Query Collection. The *second query collection* is formed by sets of RDF triples baring no full-text operators, i.e. $FT_{pr} = 0\%$. In this filtering scenario, triples match at a higher percentage with an incoming publication as there is no full-text restriction to purge out irrelevant publications based on their textual information. This experimental setup gives a good overview at the structural matching performance of the algorithms under investigation. Intuitively, the

Parameter	Description	Baseline value
I_q	Number of incoming queries	$20K$
I_{pub}	Number of incoming publications	$5K$
DB	Number of queries indexed in the database	$100K$
FT_{pr}	Percent of tuples that contain full-text restrictions in a query	50%

Table 4.3: Parameters' description and their baseline values.

algorithms should be able to perform equally or better compared to the main profile collection.

The Third Query Collection. The *third query collection* aims at examining the algorithms on their textual filtering capabilities and demonstrates the importance of solutions that support natively full-text filtering. In this filtering scenario, each tuple present in a query, contains a full-text operator with a probability of $FT_{pr} = 100\%$ (when this is applicable). As all tuples contain full-text operators, the algorithms that we evaluate will demonstrate their scaling capabilities when the indexing structures contains the maximum percentage of full-text restrictions. To this end, this experimental setup serves as a stress-test for the full-text filtering capabilities of the proposed algorithms.

Publication Set

In order to evaluate the query collections, described above, we selected $I_{pub} = 5K$ publications from the *DBpedia* corpus. The set selected had structural and textual information as extracted and processed from the *Wikipedia* domain. The publications contain human-generated, real-life data, thus providing a realistic overview of the performance of the algorithm. The selected publications are used in every filtering event during the algorithm evaluation. We maintain the same publication set through the evaluation process against different query collections. Thus, it is asserted that the algorithms are evaluated based on their indexing capabilities and the nature of queries they index.

Metrics Employed

In our evaluation, we present and discuss the filtering time and throughput of each algorithm, i.e., the amount of time needed to locate all continuous queries satisfied by an incoming publication. We present and compare the memory requirements of the algorithm. As all algorithms index the same query databases, a lower memory requirement indicates a more compact clustering of data while a higher memory footprint indicates a less compact database. Additionally, we present the algorithms differences in filtering time and demonstrate how they differentiated during the size increase of the profile database.

Finally, we present the insertion time of each algorithm, i.e., the amount of time needed to index a set of queries $I_q = 20K$ into the database. Table 4.3 summarizes the parameters examined in our experimental evaluation with their baseline values.

Technical Configuration

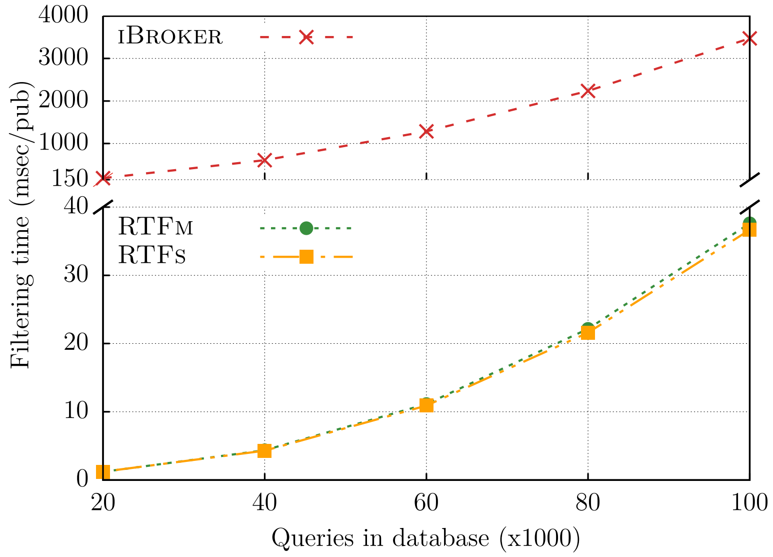
All algorithms shown, were implemented in C++, and an of-the-shelf PC with a Core $i7$ $3.6GHz$ processor and $8GB$ RAM running Ubuntu Linux 14.04 was used. The time shown in the graphs is wall-clock time and the results of each experiment are averaged over 10 runs to eliminate any fluctuations in time measurements.

4.5.2 Results when Varying the Query Database Size

In this section, we present the most significant findings for the proposed algorithms, when varying the query database size DB for queries of $FT_{pr} = 50\%$. The time reported in the graphs represents the average time needed to filter a collection of $I_{pub} = 5K$ publications.

Comparing Filtering Time

This section presents results concerning the filtering time required to match an incoming publication, when the DB size is increasing. Figure 4.9 presents the time in milliseconds needed to filter an incoming publication for $I_{pub} = 5K$ publications. Please notice that the y-axis is split into two parts due to high differences in the



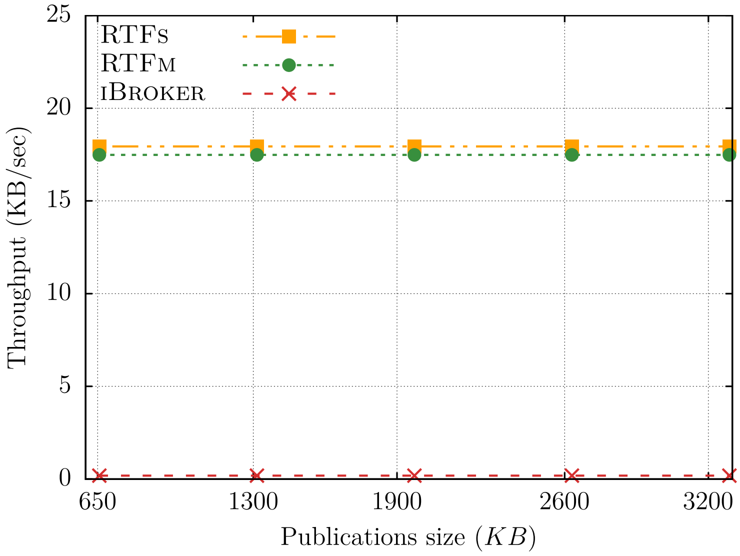
Parameter	Value
I_q	$20K$
I_{pub}	$5K$
DB	$20K - 100K$
FT_{pr}	50%

Figure 4.9: Comparing the filtering time, when varying the query database size DB, for a query set of $FT_{pr} = 50\%$.

performance of RTF variants (RTFm and RTFs) and iBroker. We observe that filtering time increases for all algorithms as the DB size grows. Algorithms RTFs and RTFm achieve the lowest filtering times, suggesting better performance. Please note that both Algorithms RTFs and RTFm exhibit relatively the same performance, with Algorithm RTFm being the fastest. Algorithm iBroker is more sensitive to DB size changes compared to RTF due to the data structures it utilizes to index the queries, e.g., an inverted index that does not implement any clustering techniques. More specifically, the results indicate that Algorithm RTFm filters incoming publications 92 times faster compared to iBroker, for a query database size of $DB = 100K$. Algorithm RTFs achieves the lowest filtering time, namely 2.5% faster than RTFm and 94 times faster than iBroker, and more specifically it achieves 36 msec/publication of filtering time, for a query database size of $DB = 100K$.

Comparing Filtering Throughput

This section presents the results concerning the algorithms' filtering throughput, when the query database size remains constant $DB = 100K$, for $I_{pub} = 5K$ incoming publications, while the average publication size (KB) is increasing.



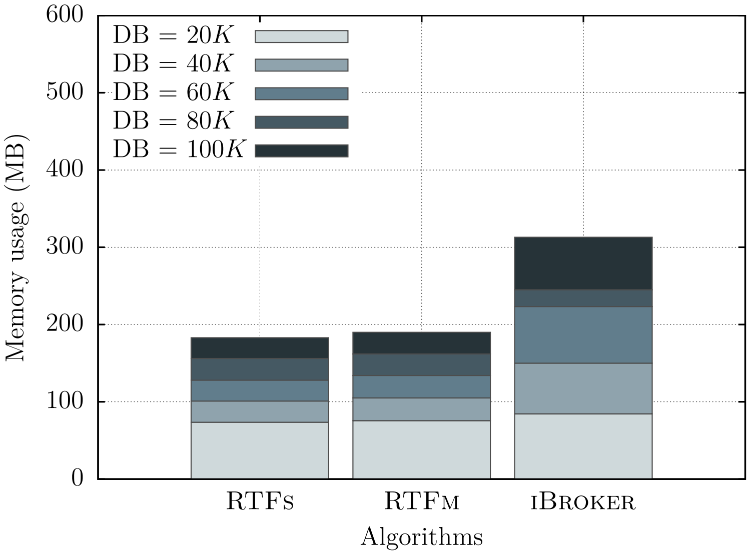
Parameter	Value
I_q	$20K$
I_{pub}	$5K$
DB	$100K$
FT_{pr}	50%

Figure 4.10: Comparing the filtering throughput, for a query database size $DB = 100K$, for a query set of $FT_{pr} = 50\%$, when varying the average publication size (KB).

Figure 4.10 presents the throughput that all algorithms achieve during the filtering of $I_{pub} = 5K$ incoming publications. We observe that the throughput remains steady throughout the publication events. This is attributed to the nature of the algorithms, as their filtering capability is not affected by the publications' size but from the indexing structures that store the queries. Algorithms RTFs and RTFm achieve the highest throughput, thus the best performance, compared to Algorithm iBroker. More specifically algorithms RTFs and RTFm achieve a throughput of more than 17 KB/sec that corresponds to more than 27 pubs/sec. On the other hand, iBroker accomplishes a throughput of 0.18 KB/sec that corresponds to 0.28 pubs/sec. To this end, both Algorithms RTFs and RTFm achieve a higher throughput rate of two orders of magnitude, when compared to Algorithm iBroker.

Comparing Memory Usage

In Figure 4.11, we present the results for the memory requirements of each algorithm when increasing the query database DB by $I_q = 20K$ new continuous queries in each iteration. We observe that both RTF's variations exhibit similar memory requirements, where Algorithm RTFs has the lowest memory requirements using 183

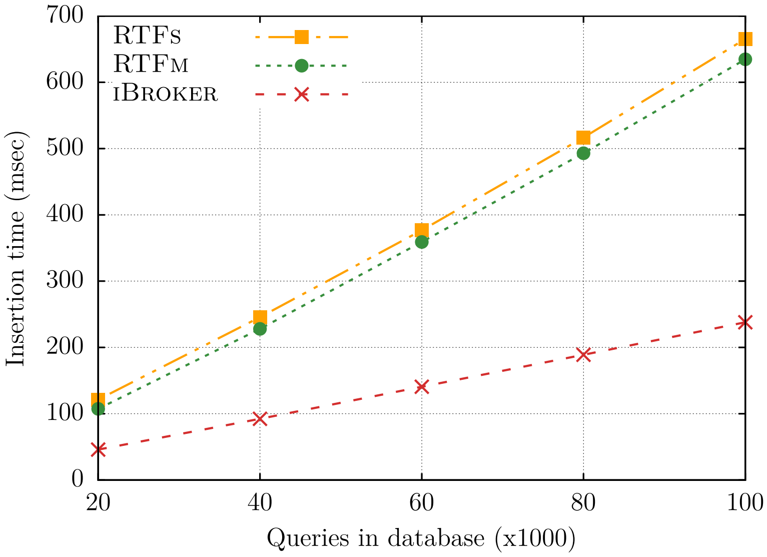


Parameter	Value
I_q	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	50%

Figure 4.11: Comparing the memory usage, when varying the query database size DB, for a query set of $FT_{pr} = 50\%$.

MB for storing the whole query database, while Algorithm RTFm occupies 190 MB for the same $DB = 100K$. RTFm's additional memory requirements is attributed to the fact that, RTFm maintains multiple forests of trees for the indexing of textual constraints. We observe that RTF's variations reserve the majority of their memory when indexing the first $I_q = 20K$ to an empty database. This behavior is attributed to the creation of many new tries due to the index structure initialization. Namely, RTFs reserves 73 MB when indexing the first $I_q = 20K$ queries and RTFm reserves 75 MB. For every new $I_q = 20K$ inserted into the database RTFs and RTFm do not require more than $28MB$ to facilitate the indexing of new queries due to the accommodation of new queries mostly in existing tries. Finally, algorithm iBroker occupies 313 MB of memory to index a database of $DB = 100K$ queries. iBroker reserves 84 MB for the first $I_q = 20K$ queries, while it requires more than 60 MB of memory to index every set of $I_q = 20K$ new queries.

The variations of RTF namely Algorithms RTFs and RTFm, exhibit low memory requirements as they utilize clustering techniques to capture the common elements of queries (both on a structural and textual level). Algorithm RTFs has lower memory requirements, when compared against RTFm, as it utilizes a single text indexing structure, thus, achieving higher clustering and generating a



Parameter	Value
I_q	$20K$
I_{pub}	$5K$
DB	$20K - 100K$
FT_{pr}	50%

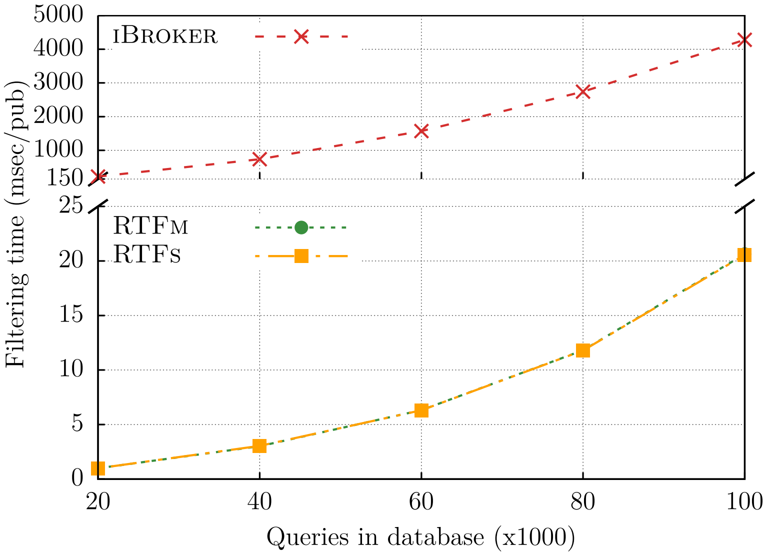
Figure 4.12: Comparing the insertion time, when varying the query database size DB, for a query set of $FT_{pr} = 50\%$.

highly compact forest. On the other hand, Algorithm RTFm employs multiple text indexing structures and thus, misses clustering opportunities as displayed by the higher memory requirements. Finally, Algorithm iBroker does not employ any text clustering techniques, thus, requiring more memory to index the same query set.

Comparing Insertion Time

In this section, we discuss the query indexing time required of all algorithms, when the query database size DB is increasing.

Figure 4.12 shows the insertion time in milliseconds required to insert $I_q = 20K$ queries when the DB size increases. We observe that the algorithms require more time to index new queries as the database size increases. Algorithms RTFs and RTFm need more time to index the same number of queries $I_q = 20K$ compared to iBroker. This can be explained as follows. The variations of RTF utilise trie-based data structures to capture and index the common structural and textual constraints of the queries. Trie traversal results to high insertion time during the indexing phase. On the other hand, insertion in an inverted index (as done by Algorithm iBroker) is faster. Notice that insertion time is not critical in



Parameter	Value
I_q	$20K$
I_{pub}	$5K$
DB	$20K - 100K$
FT_{pr}	0%

Figure 4.13: Comparing the filtering time, when varying the query database size DB, for a query set of $FT_{pr} = 0\%$.

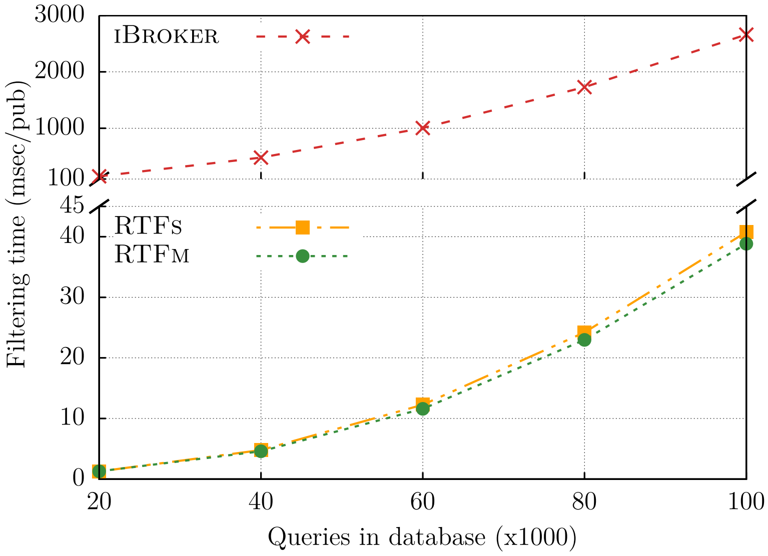
a pub/sub scenario; the most important dimension is filtering time/throughput that defines the processing rate of publications.

4.5.3 Results when Varying the Full-Text Percentage

In this section, we present the most interesting results concerning the RTF variants when varying the percentage of full-text constraints in the tuples. We evaluate the structural matching performance of RTFm and RTFs when $FT_{pr} = 0\%$, and stress-test the algorithms when the query database contains the highest number of full-text constraints possible, i.e., when $FT_{pr} = 100\%$.

Comparing Filtering Time

Figure 4.13 presents the time in milliseconds needed to filter an incoming publication, for $I_{pub} = 5K$ publications, against a query database of $FT_{pr} = 0\%$ (*second query collection*). Please notice, that the y-axis is split into two parts due to high differences in the performance of RTF variants (RTFm and RTFs) and iBroker. We observe that the filtering time increases for all algorithms as the query database size increases. Algorithms RTFs and RTFm achieve the lowest filtering times, suggesting better performance, while Algorithm iBroker exhibits

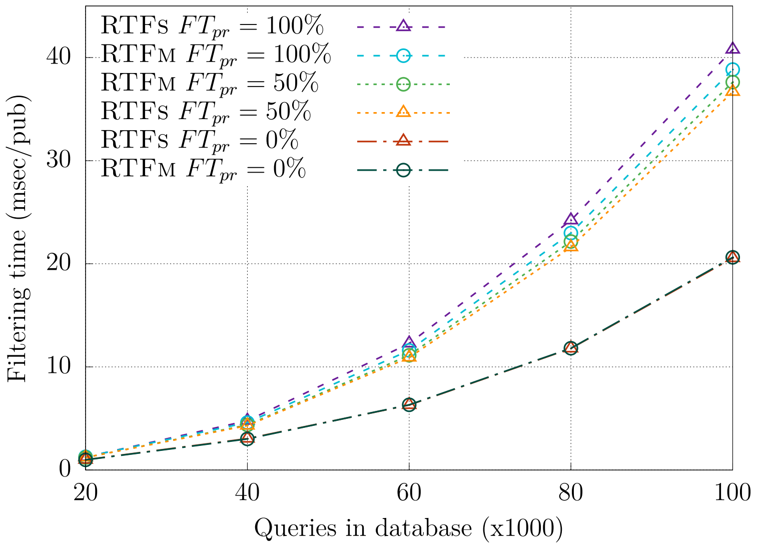


Parameter	Value
I_q	$20K$
I_{pub}	$5K$
DB	$20K - 100K$
FT_{pr}	100%

Figure 4.14: Comparing the filtering time, when varying the query database size DB, for a query set of $FT_{pr} = 100\%$.

higher increases. Algorithm iBroker is more sensitive to DB size changes when compared to RTF due to the data structures it utilizes to index the queries i.e. an inverted index that does not employ any clustering techniques. Additionally, notice that both RTF variants exhibit identical filtering time, this is attributed to the fact that in this experimental setup (where $FT_{pr} = 0\%$) we evaluate the structural matching performance of the algorithms. As both RTF variants share the same underlying mechanism for structural matching their performance is identical. More specifically, the results indicate that Algorithms RTFm and RTFs filter an incoming publication 99.55% faster when indexing a database size of $DB = 20K$ and $FT_{pr} = 0$, compared to Algorithm iBroker. Finally, the difference in performance is consistent for all database sizes DB.

Figure 4.14 presents the time in milliseconds needed to filter an incoming publication, for $I_{pub} = 5K$ publications, against a query database of $FT_{pr} = 100\%$ (*third query collection*). Please notice, that the y-axis is split into two parts due to to high differences in performance of RTF variants (RTFm and RTFs) and Algorithm iBroker. We observe that the filtering time increases for all algorithm as the query database size DB increases. Algorithms RTFs and RTFm achieve the lowest filtering times, suggesting better performance, while Algorithms iBroker

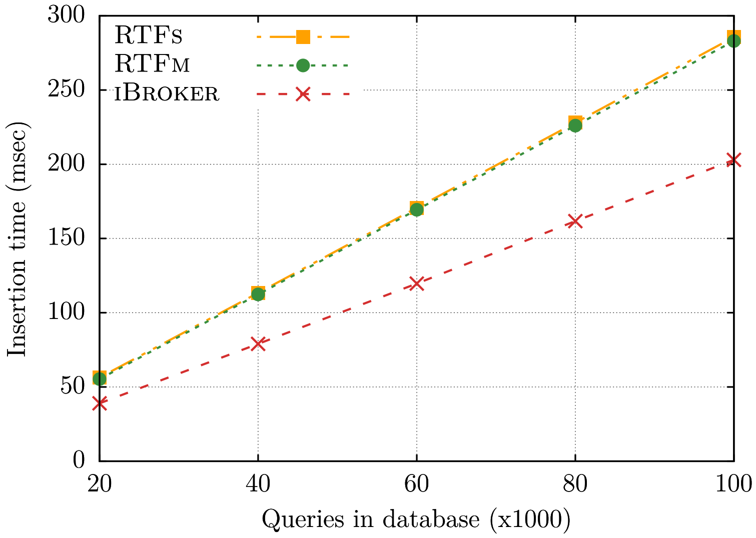


Parameter	Value
I_q	$20K$
I_{pub}	$5K$
DB	$20K - 100K$
FT_{pr}	0%, 50%, 100%

Figure 4.15: Comparing RTF's filtering time, when varying the query database size DB and the percent of full-text constraints FT_{pr}.

exhibits higher filtering times. As discussed in Section 4.5.2, this performance difference can be attributed to the nature of data structures employed by each algorithmic solution. More specifically, Algorithms RTFs and RTFm employ trie-based data structures that aim at clustering together common query parts, while Algorithm iBroker employs an inverted index that does not employ any clustering techniques. Finally, the results indicate that Algorithms RTFs and RTFm filter an incoming publication 98.4% and 98.5% respectively compared to iBroker , when $DB = 100K$ and $FT_{pr} = 100K$.

Finally, Figure 4.15 presents a comprehensive presentation of the filtering results for the three distinct query collections (Section 4.5.1). To this end, Figure 4.15 shows the time needed to filter an incoming publication when increasing the query database size DB and the percent of full-text constraints with $FT_{pr} = 0\%, 50\%$ and 100%. As expected, RTFs and RTFm achieve the lowest filtering times when $FT_{pr} = 0\%$, and exhibit the same performance as they utilise the same indexing structure for the structural constraints of the queries. Finally, RTFs and RTFm show an increase in their filtering times when $FT_{pr} = 100\%$ with RTFm achieving better performance (i.e., lowest filtering time) when all queries contain full-text constraints.

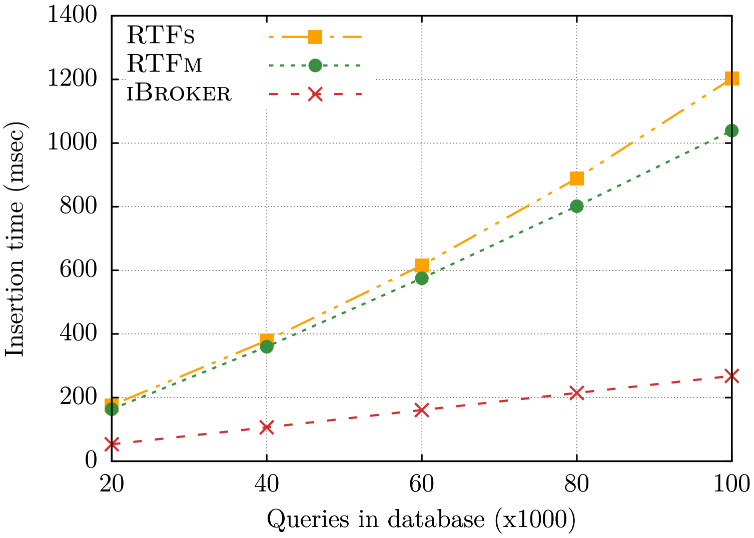


Parameter	Value
I_q	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	0%

Figure 4.16: Comparing the insertion time, when varying the query database size DB, for a query set of $FT_{pr} = 0\%$.

Comparing Insertion Time

Figure 4.16 presents the time in milliseconds that Algorithms RTFm, RTFs and iBroker require to insert $I_q = 20K$ queries when DB size is increasing and $FT_{pr} = 0\%$ (*second query collection*). We observe that the algorithms increase the time needed to index new queries as the database size DB increases. Algorithms RTFs and RTFm need more time to index the same number of queries $I_q = 20K$ compared against iBroker. Additionally, the two variations of RTF increase their time requirements faster compared to iBroker. The performance exhibited by the algorithms under study can be explained as follows: Algorithms RTFs and RTFm utilize trie-based data structures in order to index the structural restrictions of the queries and capture their common elements. The traversal of the trie-structures and search for existing tries to cluster an incoming query results in slower indexing performance as opposed to iBroker. Comparing the results of Figure 4.12 that demonstrate the insertion times for the *main query collection* against Figure 4.16, we observe that Algorithms RTFm and RTFs reduce in half their time requirements when there are no full-text restrictions to be indexed. While iBroker exhibits little time deviations when there are no full-text restrictions to be indexed.

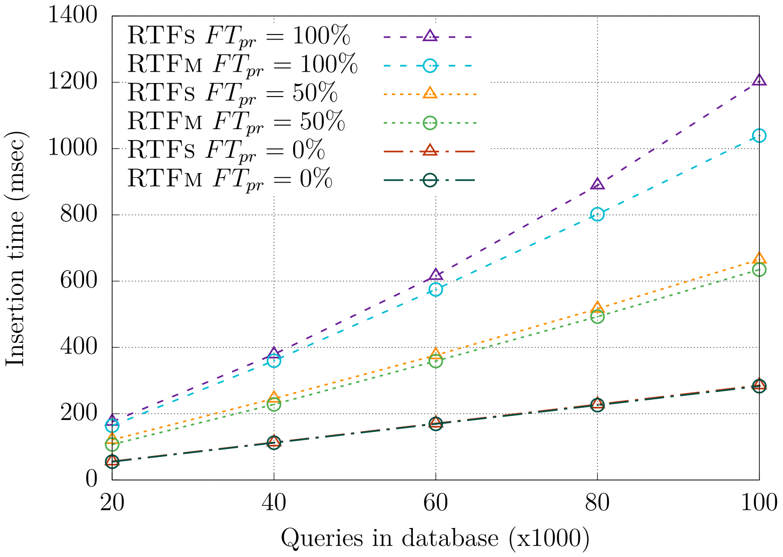


Parameter	Value
I_q	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	100%

Figure 4.17: Comparing the insertion time, when varying the query database size DB, for a query set of $FT_{pr} = 100\%$.

Figure 4.17 presents the time in milliseconds that Algorithms RTFs, RTFm and iBroker require to insert $I_q = 20K$ new queries when DB size is increasing and $FT_{pr} = 100\%$ (*third query collection*). We observe that the algorithms increase the time needed to index new queries as the database size DB increases. Algorithms RTFs and RTFm need more time to index the same number of queries $I_q = 20K$ compared against iBroker. Additionally, the two variations of RTF increase their time requirements faster compared to iBroker. The performance exhibited by the algorithms under study can be explained as follows: Algorithms RTFs and RTFm (as discussed in previous sections) utilize trie-based data structures in order to index the structural and full-text restrictions of the queries and capture their common elements. The traversal of the trie-structures and search for existing tries to cluster an incoming query results in slower indexing performance, as opposed to Algorithm iBroker that does not employ any clustering technique.

Finally, Figure 4.18 presents a comprehensive presentation of the insertion time for the three distinct query collections. To this end, Figure 4.18 shows the time required to insert $I_q = 20K$ queries when the query database size DB is increasing and when varying the percent of full-text constraints $FT_{pr} = 0\%$, 50% and 100%. As expected, Algorithms RTFs and RTFm increase their time



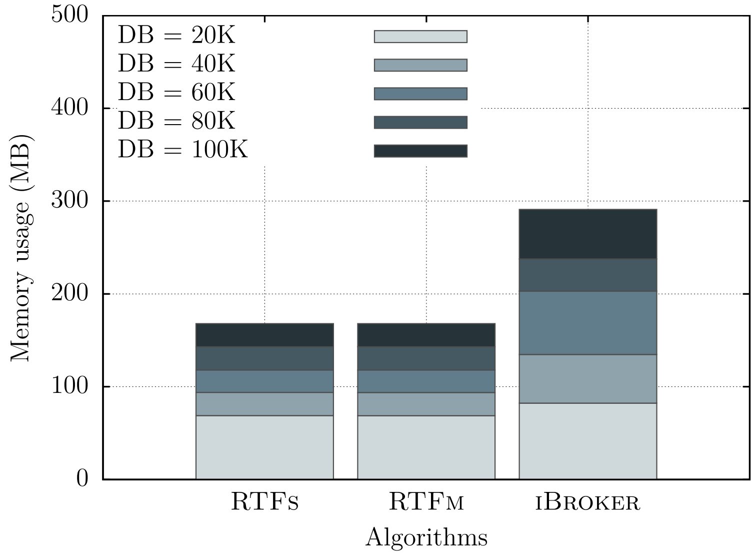
Parameter	Value
I_q	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	50%

Figure 4.18: Comparing RTF's insertion time, when varying the query database size DB and the percent of full-text constraints FT_{pr}.

requirements, when more textual constraints (i.e. $FT_{pr} = 100\%$) are included in the continues queries (*third query collection*). Finally, Algorithms RTFs and RTFm reduce their time requirements, when no textual constraints ($FT_{pr} = 0\%$) are present (*second query collection*).

Comparing Memory Usage

Figure 4.19 presents the memory requirements in megabytes that Algorithms RTFs, RTFm and iBroker require to insert $I_q = 20K$ queries when the query database size DB is increasing and $FT_{pr} = 0\%$ (*second query collection*). We observe that Algorithms RTFs and RTFm have the lowest and identical memory requirements and more specifically $168MB$ of main memory. Algorithm iBroker occupies the highest amount of main memory, requiring $291MB$ to index a database of $DB = 100K$ queries. RTF's variations exhibit low memory requirements as they utilize clustering techniques to capture the common structural elements of the profiles. As in this experimental setup there are no full-text constraint present in the query set (*second query collection*), both algorithms present the same memory usage when indexing the same query collection. Finally, we observe that the RTF variations reserve the majority of their memory when indexing the first incoming

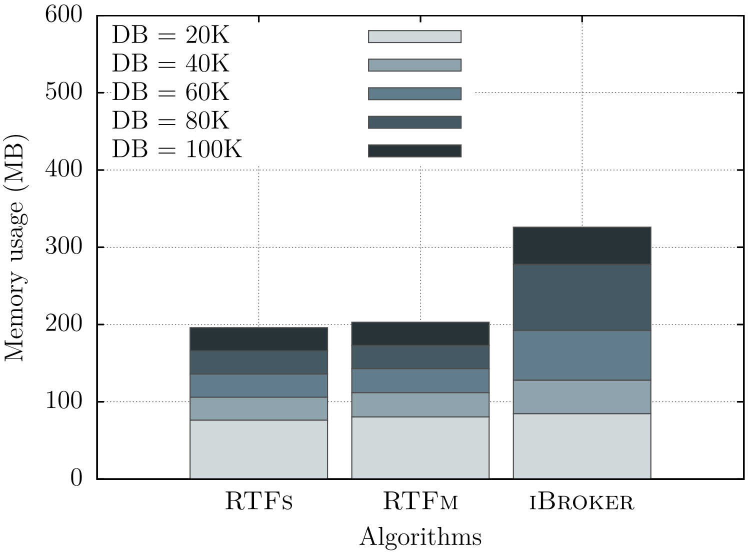


Parameter	Value
I_q	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	0%

Figure 4.19: Comparing the memory usage, when varying the query database size DB, for a query set of $FT_{pr} = 0\%$.

$I_q = 20K$ queries (i.e. when $DB = 0 - 20K$). This behavior can be attributed to the initialization of the indexing structures that the algorithms use. Namely both RTFs and RTFm reserve $68MB$ of memory during the first insertion of $I_q = 20K$, while the algorithms do not require more than $25MB$ to facilitate the indexing of the rest of new queries. On the other hand, Algorithm iBroker reserves $82MB$ of memory to index the first $I_q = 20K$ profiles into an empty database while it requires at most $68MB$ of memory to index every set of $I_q = 20K$ new queries. Comparing the results against the finding for *first query collection* where $FT_{pr} = 50\%$ (Figure 4.11) we observe that algorithms RTFs and RTFm decrease significantly their memory requirements, thus demonstrating high scalability.

Figure 4.20 presents the memory requirements in megabytes that Algorithms RTFs, RTFm and iBroker require to insert $I_q = 20K$ queries when the query database size DB is increasing and $FT_{pr} = 100\%$ (*third query collection*). We observe that Algorithm RTFs has the lowest memory requirements using $196MB$ for indexing the entirety of the query set $DB = 100K$. Algorithm RTFm's memory usage is at $203MB$ for indexing the same query set. Algorithm iBroker occupies the highest amount of main memory, i.e. $326MB$, to index a database of $DB = 100K$ queries. The variations of RTF exhibit low memory requirements



Parameter	Value
I_q	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	100%

Figure 4.20: Comparing the memory usage, when varying the query database size DB, for a query set of $FT_{pr} = 100\%$.

as they utilize clustering techniques to capture the common elements of the queries both on structural and full-text constraints. Algorithm RTFs exhibits the lowest requirements as it utilizes a single full-text indexing structure, thus, capturing more common elements and generating a compacter forest. On the other hand, Algorithm RTFm by utilizing multiple forests misses important clustering opportunities, thus, requiring more memory to index the same query set.

4.5.4 Summary of Results

Our experimental evaluation demonstrated the filtering effectiveness of algorithm RTFs for cases where queries with text constraints are not very often, whereas algorithm RTFm is better suited for cases where a high percentage or queries contain text constraints. Both algorithms RTFs and RTFm are over two orders of magnitude faster than iBroker on average.

4.6 Outlook and Future Directions

In this chapter, we have studied the problem of full-text support on ontology-based pub/sub systems. In this context, we proposed a full-text extension for SPARQL continuous queries and a family of query indexing algorithms that are two orders of magnitude faster at filtering tasks than a state-of-the-art competitor.

Future directions of this work include: (i) extending our indexing solution to other Boolean operators, (ii) supporting vector space queries and text representation in SPARQL, and (iii) adapting our algorithms to multi-processor environments.

5

Efficient Continuous Multi-Query Processing over Evolving Graph Data

In the previous chapter, we dealt with the problem of efficiency in the domain of *context-based information filtering.* In this chapter, we focus on evolving graphs and graph streams, we introduce the notion of *continuous multi-query processing over graph streams* and discuss its applications to a number of use cases. Capturing the continuous evolution of a graph can be achieved by long-standing sub-graph queries and can yield important insights about the nature and activities of the underlying network. To this end, we designed and developed a novel algorithmic solution for efficient multi-query evaluation against a stream of graph updates and experimentally demonstrated its applicability. Our results against two baseline approaches using either real-world or synthetic datasets confirm a two orders of magnitude improvement of the proposed solution. The initial steps of this research have been published in [121], while the complete results and extended research have been submitted to a journal and are currently under review.

The rest of this chapter is formed of seven sections. Section 5.1 provides the motivation for this work, as well as the description of the research problem addressed. Section 5.2 presents the adopted data and query model. Section 5.3 presents our proposed solution Algorithm TriC and an extension that employs a caching strategy. Section 5.4 presents Algorithms Inv and Inc, two advanced baselines that utilize inverted index data structures, as well as a third baseline that is based on the well-established graph database Neo4j [28]. Subsequently, Section 5.5 presents our experimental results using real-world and synthetic datasets. Finally, Section 5.6 discusses several application scenarios for our solution, and Section 5.7 concludes the chapter by discussing the results, provides the outlook, and presents future directions of this research.

5.1 Motivation

In recent years, graphs have emerged as prevalent data structures to model information networks in several domains such as social networks, knowledge bases, communication networks, biological networks and the World Wide Web. These graphs are *massive* in scale and *evolve* constantly due to frequent updates. For example, Facebook has over 1.52 billion daily active users who generate over 500K posts/comments and 4 million likes every minute resulting in massive updates to the Facebook social graph [122].

To gain meaningful and up-to-date insights in such frequently updated graphs, it is essential to be able to monitor and detect continuous patterns of interest. There are several applications from a variety of domains that may benefit from such monitoring. In social networks, such applications may involve targeted advertising, spam detection [10, 11], and fake news propagation monitoring based on specific patterns [12, 13]. Similarly, other applications like (i) protein-to-protein interaction patterns in biological networks [14, 15], (ii) traffic monitoring in transportation networks, (iii) attack detection (e.g., distributed denial of service attacks in computer networks), (iv) question answering in knowledge graphs [16, 17], and (v) reasoning over RDF graphs may also benefit from such pattern detection.

For the applications mentioned above it is necessary to express the required patterns as *continuous sub-graph queries* over (one or many) streams of graph updates and appropriately *notify* the subscribed users for any patterns that match their subscription. Detecting these query patterns is fundamentally a sub-graph

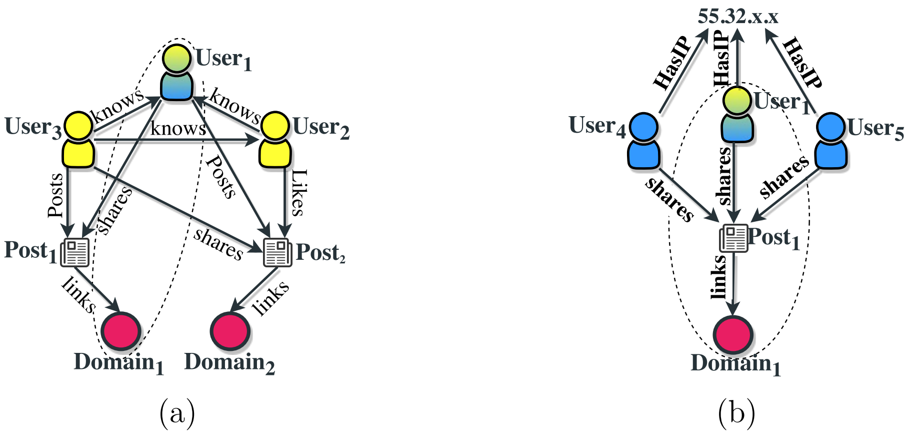


Figure 5.1: Spam detection: Users sharing and liking content with links to flagged domains. (a) A clique of users who know each other, and (b) Users sharing the same IP address.

isomorphism problem which is known to be NP-complete due to the exponential search space resulting from all possible sub-graphs [18, 19]. The typical solution to address this issue is to pre-materialize the necessary sub-graph views for the queries and perform exploratory joins [20]; an expensive operation even for a single query in a static setting.

Nevertheless, the applications described above deal with graph streams in such a setup that is often essential to be able to support hundreds or thousands of continuous queries simultaneously. This leads to several challenges that require:

- Quickly detecting the affected queries for each update.
- Maintaining a large number of materialized views.
- Avoiding the expensive join and explore approach for large sets of queries.

To better illustrate the remarks above, consider the application of spam detection in social networks. Figure 5.1 shows an example of two graph patterns that may emerge from malicious user activities, i.e., users posting links to domains that have been flagged as fraudulent. Notice that malicious behavior could be caused either because a group of users that know each other share and like each other's posts containing content from a flagged domain (Figure 5.1(a)), or because the group of users shared the same flagged post several times from the same IP

(Figure 5.1(b)). Even though these two queries are fundamentally different and produce different matching patterns, they share a common sub-graph pattern, i.e., "User1$\xrightarrow{\text{shares}}$Post1$\xrightarrow{\text{links}}$Domain1". If these two queries are evaluated independently, all the computations for processing the common pattern have to be executed twice. However, by identifying common patterns in query sets (shared by multiple continuous queries), we can amortize the costs of processing and answering them.

One simple approach to avoid processing all the (continuous) queries upon receiving a graph update is to index the query graphs using an inverted-index at the granularity of edges. While this approach may help us quickly detect all the affected queries for a given graph update, we still need to perform several exploratory joins to answer the affected queries. For example, in Figure 5.1, we would need to join and explore the edges matching the pattern "User1$\xrightarrow{\text{Shares}}$Post1 and Post1$\xrightarrow{\text{Links}}$Domain1" upon each update to process the two queries. On the contrary, if we first identify the maximal sub-graph patterns shared among the queries, we can minimize the number of operations necessary to answer the queries and this will consequently, reduce the query answering time. Therefore, a solution which groups queries based on their shared patterns would be expected to deliver significant performance gains. To the best of our knowledge, none of the existing works provide a solution that exploits common patterns for continuous multi-query answering.

In this paper, we address this gap by proposing a novel *algorithmic solution*, coined TRIC (TRIe-based Clustering) to index and cluster continuous graph queries. In TRIC, we first decompose queries into a set of directed paths such that each vertex in the query graph pattern belongs to at least one path (path covering problem [27]). However, obtaining such paths leads to redundant query edges and vertices in the paths; this is undesirable since it affects the performance of the query processing. Therefore, we are interested in finding paths which are shared among different queries, with minimal duplication of vertices. The paths obtained are then indexed using 'tries' that allow us to minimize query answering time by (i) quickly identifying the affected queries, (ii) sharing materialized views between common patterns, and (iii) efficiently ordering the joins between materialized views affected from the update.

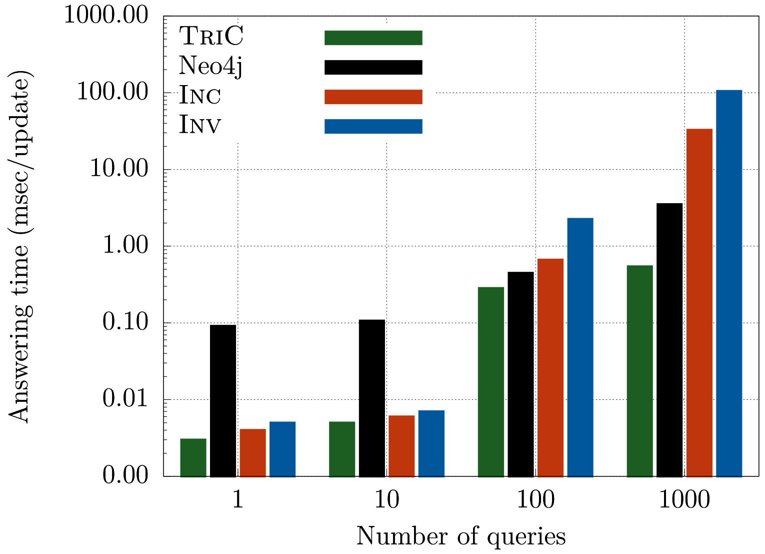


Figure 5.2: Log-log plot comparing query answering time for the *SNB* dataset from the LDBC benchmark, when varying the number of queries exponentially.

Figure 5.2 shows the potential for improvement in query answering time with our query clustering solution TRIC, for the LDBC graph benchmark [123]. We can observe that TRIC provides a speedup of two orders of magnitude in query answering time, compared against the two advanced baselines using the “inverted indexing technique" (INV, INC) and the graph database Neo4j that do not exploit the common sub-graph patterns in the queries. The overall evaluation and comparison of the algorithms is presented in detail in Section 5.5.

In the light of the above, our *contributions* can be summarized as follows:

- We formalize the problem of continuous multi-query answering over graph streams (Section 5.2).

- We propose a novel query graph clustering algorithm that is able to efficiently handle large numbers of continuous graph queries by resorting on (i) the decomposition of continuous query graphs to minimum covering paths and (ii) the utilization of tries for capturing the common parts of those paths. We identify different variations of the main algorithmic solution and the baseline approaches, which utilize caching strategies and demonstrate the effect of such solutions on the problem at hand. (Section 5.3).

- Since no prior work in the literature has considered continuous multi-query answering, we designed and developed two algorithmic solutions that utilize inverted indexes for the query answering. Additionally, we deploy and extend Neo4j [28], a well-established graph database solution, to support our proposed paradigm. To this end, the proposed solutions will serve as baselines approaches during the experimental evaluation. (Section 5.4).

- We experimentally evaluate the proposed solution using three different datasets from social networks, transportation, and biology domains, and compare the performance against the three baselines. In this context, we show that our solution can achieve up to two orders of magnitude improvement in query processing time (Section 5.5).

- Finally, we identify and present *application scenarios* from various domains, that could benefit from the proposed continuous multi-query answering paradigm (Section 5.6).

In this section, we presented the motivation behind our work and gave a brief overview of the algorithmic solutions we developed in order to solve the continuous multi-query processing over graph streams problem. In the following section, we present the data model we employed in our approach.

5.2 Data Model and Problem Definition

In this section, we present the data model that defines the evolving graph, as well as the stream of changes that arrive at the evolving graph (Section 5.2.1). Additionally, we present the query model that users can utilize to capture certain patterns that emerge in an evolving graph (Section 5.2.2).

5.2.1 Graph Model

In our approach, we employ attribute graphs [124] (Definition 5.1) as our data model, as they are used natively in a wide variety of applications, such as social network graphs, traffic network graphs, and citation graphs. Datasets in other data models can be mapped to attribute graphs in a straightforward manner so that our approach can be applied to them as well.

Definition 5.1 *An* attribute graph *G is defined as a directed labeled multigraph:*

$$G = (V, E, l_V, l_E, \Sigma_V, \Sigma_E)$$

where V is the set of vertices and E the set of edges. An edge $e \in E$ is an ordered pair of vertices $e : (s, t)$, where $s, t \in V$ represent source and target vertices. $l_V : V \to \Sigma_V$ and $l_E : E \to \Sigma_E$ are labeling functions assigning labels to vertices and edges from the label sets Σ_V and Σ_E.

For ease of presentation we adopt the following notation: We denote an edge e as $e = (s, t)$, where e, s and t are the labels of the edge($l_E(e)$), source vertex ($l_V(s)$) and target vertex ($l_V(t)$) respectively.

As our goal is to facilitate efficient continuous multi-query processing over graph streams, we also provide formal definitions for updates (Definition 5.2) and graph streams (Definition 5.3).

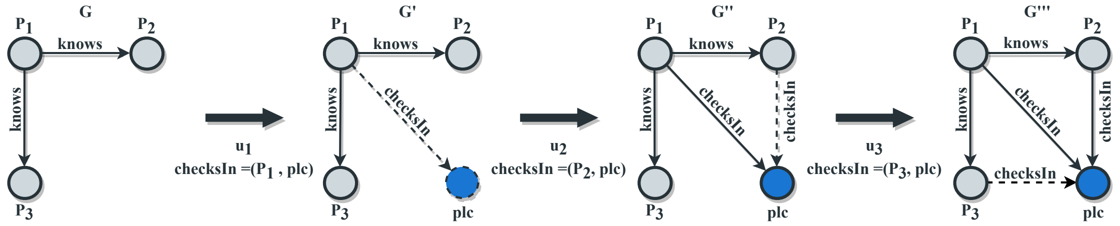
Definition 5.2 *An update u_t on graph G is defined as an* addition (e) *of an edge e at time t. An addition leads to new edges between vertices and possibly the creation of new vertices.*

Definition 5.3 *A* graph stream *$S = (u_1, u_2, \ldots, u_t)$ of graph G is an* ordered sequence *of updates.*

Figure 5.3(a) presents an update stream S consisting of three graph updates u_1, u_2, and u_3 generated from social network events. While, Figure 5.3 (b) shows the initial state of graph G and its evolution after inserting sequentially the three updates.

Update stream S
$u_1 = (checksIn = (P_1, \text{plc}))$
$u_2 = (checksIn = (P_2, \text{plc}))$
$u_3 = (checksIn = (P_3, \text{plc}))$

(a)



(b)

Figure 5.3: (a) An update stream S and (b) the evolution of graph G after inserting $u_i \in S$.

5.2.2 Query Model

In the proposed model, a user (or service that operates on behalf of the user) subscribes into subgraphs or motifs (i.e., subgraphs with a fixed number of vertices commonly found in a graph) that emerge through the evolution of the graph and match a given set of attribute restrictions. These structural and attribute constraints are expressed by the user (or service), while their native structure is similar to the evolving graph. During the graph evolution, subgraphs emerge that match the restrictions (i.e., query graphs are satisfied), while the respective users are notified. In order to enable users with this functionality we define *query graph patterns* that are used to capture the subgraph and attribute restrictions of an evolving graph.

For our query model we assume that users (or services operating on their behalf) are interested to learn when certain patterns emerge in an evolving graph. Definition 5.4 formalizes *query graph patterns* that define structural and attribute constraints on graphs.

Definition 5.4 *A* query graph pattern Q_i *is defined as a directed labeled multigraph:*

$$Q_i = (V_{Q_i}, E_{Q_i}, vars, l_V, l_E, \Sigma_V, \Sigma_E)$$

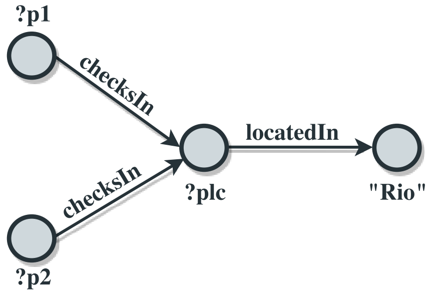


Figure 5.4: An example query graph pattern capturing activities that occur, within a social network.

where V_{Q_i} *is a set of vertices,* E_{Q_i} *a set of edges, and vars a set of variables.* $l_V : V \rightarrow \{\Sigma_V \cup vars\}$ *and* $l_E : E \rightarrow \Sigma_E$ *are labeling functions assigning labels (and variables) to vertices and edges.*

Example 5.1 *Let us consider an example where a user wants to be notified when his friends visit places nearby. Figure 5.4 shows the corresponding query graph pattern that will result in a user notification when two people check in at the same place/location in Rio.*

Based on the above definitions, let us now define the problem of *multi-query processing over graph streams*, which is the main focus of this work.

Definition 5.5 ***Problem Definition.*** *Given a set of query graph patterns* $Q_{DB} = \{Q_1, Q_2, \ldots, Q_k\}$*, an initial attribute graph* G*, and a graph stream* S *with continuous updates* $u_t \in S$*, the problem of* multi-query processing over graph streams *consists of continuously identifying all satisfied query graph patterns* $Q_i \in Q_{DB}$ *when applying incoming updates.*

The query answering problem is of high importance as servers are expected to handle hundreds of thousands of query graph patterns, and high rates of incoming updates through graph streams.

Query Set and Graph Modifications

A set of query graph patterns Q_{DB} is subject to modifications (i.e., additions and deletions). In this work, we focus on streamlining the query indexing phase, while developing techniques that allow processing each incoming query graph pattern separately, thus supporting continuous additions in Q_{DB}. In the same manner, a graph G is subject to edge additions and deletions, our main objective is to efficiently determine the queries satisfied by an edge addition. The proposed model does not require indexing the entire graph G and retains solely the necessary parts of G for the query answering. To this end, we do not further discuss deletions on Q_{DB} and G, as we focus on providing high performance query answering algorithms. Nonetheless, we provide concise information on how graph deletions can be handled by our proposed solutions.

In this section, we described the graph and stream model, as well as presented the query model that employs query graph patterns. In the following sections, we describe the algorithmic solutions developed to provide efficient multi-query evaluation over graph streams.

5.3 Trie-Based Clustering

In order to solve the problem defined in the previous section, we propose Algorithm TriC (TRIe-based Clustering), which uses clustering techniques to capture common parts in a given set of continuous query subgraph patterns. We aim at efficiently solving the multi-query evaluation problem by utilizing clustering techniques in order to capture the common parts of the continuous query subgraph patterns. In this section we present Algorithm TriC, a solution we designed and implemented to efficiently solve the problem.

As motivated in Section 5.1, the *key idea* behind Algorithm TriC lies in the fact that query graph patterns overlap in their structural and attribute restrictions. After identifying and indexing these shared characteristics (Section 5.3.1), they can be exploited to batch-answer the indexed query set and in this way reduce query response time (Section 5.3.2).

5.3.1 Query Indexing Phase

Algorithm TriC indexes each query graph pattern Q_i by applying the following two steps:

1. Transforming the original query graph pattern Q_i into a set of path conjuncts, that cover all vertices and edges of Q_i, and when combined can effectively re-compose Q_i.

2. Indexing all paths in a trie-based structure along with unique query identifiers, while clustering all paths of all indexed queries by exploiting commonalities among them.

In the following, we present each step of the query indexing phase of Algorithm TriC and give details about the data structures utilized. The pseudocode of the query indexing phase of TriC is provided in Figure 5.5.

Extracting the Covering Paths (Step 1)

In the first step of the query indexing process, Algorithm TriC decomposes a query graph pattern Q_i and extracts a set of paths $CP(Q_i)$ (Figure 5.5, line 1). This set of paths, covers all vertices $V \in Q_i$ and edges $E \in Q_i$. At first, we give the definition of a path and subsequently define and discuss the covering path set problem.

Definition 5.6 *A* path $P_i \in Q_i$ *is defined as a list of vertices* $P_i = \{v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \dots v_k \xrightarrow{e_k} v_{k+1}\}$ *where* $v_i \in Q_i$*, such that two sequential vertices* $v_i, v_{i+1} \in P_i$ *have exactly one edge* $e_i \in Q_i$ *connecting them, i.e.,* $e_k = (v_k, v_{k+1})$*.*

Definition 5.7 *The* covering path problem *(CP) of a query graph* Q_i *is defined as a set of paths* $CP(Q_i) = \{P_1, P_2, \dots, P_k\}$ *that cover all* vertices *and* edges *of* Q_i*. In more detail, we are interested in the least number of paths while ensuring*

```
Algorithm: INDEX
Input: Query $Q_i = (V_{Q_i}, E_{Q_i}, vars, l_V, l_E, \Sigma_V, \Sigma_E)$
Result: Store $Q_i$ in $Q_{DB}$ $(Q_{DB} \leftarrow Q_{DB} \cup\ Q_i)$
Step 1 :
// Obtain the set of covering paths
$Paths \leftarrow CP(Q_i)$
Step 2 :
// For each covering path $P_i$ of query graph $Q_i$
foreach $P_i \in Paths$ do
    foreach trie $T_i$ with $root(T_i) = e_1 : e_1 \in P_i$ do
        // Get reference to trie $T_i$
        $\&T_i \leftarrow rootInd(T_i)$
        // Traverse trie in DFS
        $depthFirstSearch(\&T_i)$
        // If there exists a trie that can store $P_i$
        if $\exists \{n_0 \rightarrow \ldots n_i \rightarrow \ldots n_k\} \subseteq P_i$ then
            // Store the trie path positions
            $positions \leftarrow \{n_0 \rightarrow \ldots n_i \rightarrow \ldots n_k\}$
    // If all edges $e_i \in P_i$ cannot be indexed, create additional trie
       nodes to index them
    if $positions \cap P_i \neq \emptyset$ then
        $create\_nodes(P_i \setminus positions)$
    // Store the query identifier in the last trie node
    $last(positions) \leftarrow id(Q_i)$
    // Keep a reference to the last trie node
    $pathPositions \rightarrow pathPositions \cup last(positions)$
    // Store tries $T_i$ under which, edge $e_i$ is indexed
    foreach $e_i \in P_i$ do
        $edgeInd[e_i] \leftarrow T_i$
// Store the nodes that $Q_i$ was indexed under
$queryInd[id(Q_i)] \leftarrow pathPositions$
```

Figure 5.5: Pseudocode for the query indexing phase (Algorithm INDEX), performed by Algorithm TRIC.

that for every vertex $v_i \in Q_i$ there is at least one path P_j that contains v_i, i.e., $\forall i \exists j : v_i \in P_j$, $v_i \in Q_i$. In the same manner, for every edge $e_i \in Q_i$ there is at least one path P_j that contains e_i, i.e., $\forall i \exists j : e_i \in P_j$.

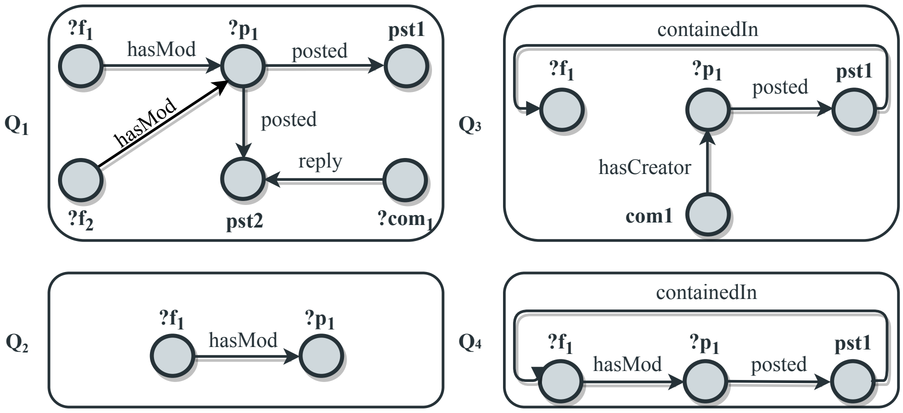
Obtaining the Set of Covering Paths

The problem of obtaining a set of paths that covers all vertices and edges is a graph optimization problem that has been extensively studied, in the domains of *software* and *circuit testing* [125]. In the domain of software testing it is used to model all possible execution states of source code, where, the execution flow is modeled as a graph. In this context, all possible execution paths of the graph should be covered, i.e., visiting all vertices and edges of the graph and obtaining the minimum set of paths that cover the graph, while a path can not be a sub-path of an already existing path. The set of covering path set is commonly known, in the software testing domain, as *prime paths*.

In our approach, we choose to solve the problem by applying a greedy algorithm, as follows: For all vertices v_i in the query graph Q_i execute a depth-first walk until a leaf vertex (no outgoing edge) of the graph is reached, or there is no new vertex to visit. Subsequently, repeat this step until all vertices and edges of the query graph Q_i have been visited at least once and a list of paths has been obtained. Finally, for each path in the obtained list, check if it is a sub-path of an already discovered path, and remove it from the list of covering paths. Please note that in queries that contain a cycle, a random vertex is chosen as the initial vertex of the dept-first walk. The end result of this procedure yields the *set of covering paths*.

Example 5.2 *In Figure 5.6(a) we present four query graph patterns, namely Q_1, Q_2, Q_3 and Q_4. These query graph patterns capture activities of users inside a social network. Those activities include users posting new content and receiving comments on their posts (Q_1, Q_3), users creating new forums (Q_2), and posting content inside the forums they moderate (Q_4).*

By applying Definition 5.7 on the four query graph patterns presented, Algorithm TriC *extracts four sets of covering paths, presented in Figure 5.6(b).*



(a)

Query ID	Set of Covering Paths
Q_1	$P_1 = \{?var \xrightarrow{hasMod} ?var \xrightarrow{posted} “pst1”\}$ $P_2 = \{?var \xrightarrow{hasMod} ?var \xrightarrow{posted} “pst2”\}$ $P_3 = \{?var \xrightarrow{reply} “pst2”\}$
Q_2	$P_1 = \{?var \xrightarrow{hasMod} ?var\}$
Q_3	$P_1 = \{“com1” \xrightarrow{hasCreator} ?var \xrightarrow{posted} “pst1” \xrightarrow{containedIn} ?var\}$
Q_4	$P_1 = \{?var \xrightarrow{hasMod} ?var \xrightarrow{posted} “pst1” \xrightarrow{containedIn} ?var\}$

(b)

Figure 5.6: (a) Four query graph patterns that capture events generated inside a social network and (b) their covering paths.

Obtaining a set of paths serves two purposes: (a) it gives a less complex representation of the query graph that is easier to manage, index and cluster, as well as (b) it provides a streamlined approach on how to perform the materialization of the subgraphs that match a query graph pattern, i.e., the query answering during the evolution of the graph.

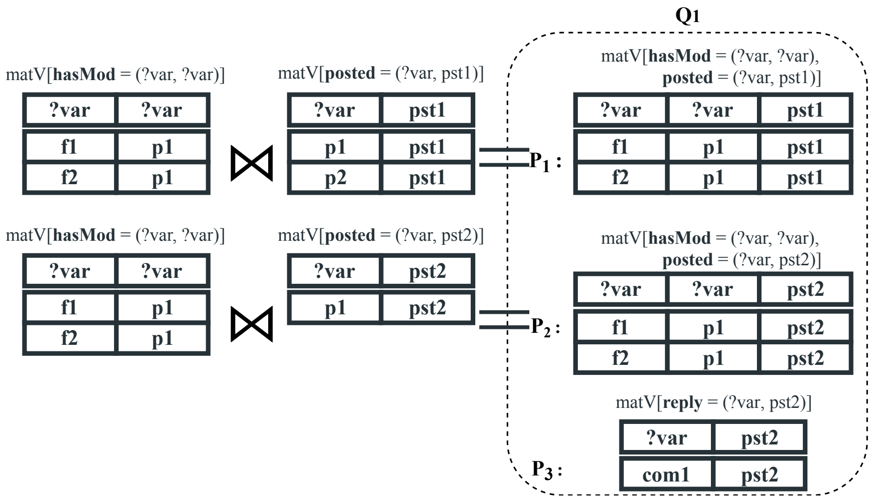
The time complexity, of obtaining the minimum set of covering paths of a query graph pattern Q_i is calculated as: (i) The complexity of obtaining a set of paths that can cover Q_i, is $\mathcal{O}(|V_{Q_i}| + |E_{Q_i}|)$, where $|V_{Q_i}|$ and $|E_{Q_i}|$ denote the cardinality of the vertex and edge sets respectively. (ii) The complexity of obtaining the minimum set of paths out of a path set P, where each path is not a subset of an other path, is $\mathcal{O}(|P|^2 * |P_{max}|)$, where $|P|$ is the cardinality of set P and $|P_{max}|$ is the cardinality of the longest path P_{max}. Thus, the total time complexity is $\mathcal{O}((|V_{Q_i}| + |E_{Q_i}|) + (|P|^2 * |P_{max}|))$.

Materialization

Each edge e_i that is present in the query set has a materialized view that corresponds to its $matV[e_i]$. The materialized view of e_i stores all the updates u_i that contain e_i. In order to obtain the subgraphs that satisfy a query graph pattern Q_i, all edges $e_i \in Q_i$ must have a non-empty materialized view (i.e., $matV \neq \emptyset$) and the materialized views should be joined as defined by the query graph pattern.

In essence, the query graph pattern determines the *execution plan* of the query. However, given that a query pattern itself is a graph, there is a high number of possible execution plans available. A path $P_i = \{v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \dots v_k\}$ serves as a model that defines the order in which the materialization should be performed. Thus, starting from the source vertex $v_1 \in P_i$ and joining all the materialized views from v_1 to the leaf vertex $v_k \in P_i : |P| = k$ yields all the subgraphs that satisfy the path P_i. After all paths P_i that belong in Q_i have been satisfied, a final join operation must be performed between all the paths. This join operation will produce the subgraphs that satisfy the query graph Q_i. To achieve this path joining set, additional information is kept about the intersection of the paths $P_i \in Q_i$. The intersection of two paths P_i and P_j are their common vertices, defined as the set of vertices that are common between them.

Example 5.3 *Figure 5.7 presents some possible materialized views that correspond to the covering paths of query graph Q_1 (Figure 5.6 (b)). In order to locate all subgraphs that satisfy the structural and attribute restrictions posed by paths P_1, P_2 and P_3 their materialized views should be calculated. More specifically, path $P_1 = \{?var \xrightarrow{hasMod} ?var \xrightarrow{posted}$ "$pst1$"$\}$, is formulated by two edges, edges $hasMod = (?var, ?var)$ and $posted = (?var, pst1)$, thus, their materialized views $matV[hasMod = (?var, ?var)]$ and $matV[posted = (?var, pst1)]$ must be joined.*



matV[**hasMod** = (?var, ?var)]

?var	?var
f1	p1
f2	p1

matV[**posted** = (?var, pst1)]

?var	pst1
p1	pst1
p2	pst1

matV[**hasMod** = (?var, ?var), **posted** = (?var, pst1)]

?var	?var	pst1
f1	p1	pst1
f2	p1	pst1

matV[**hasMod** = (?var, ?var)]

?var	?var
f1	p1
f2	p1

matV[**posted** = (?var, pst2)]

?var	pst2
p1	pst2

matV[**hasMod** = (?var, ?var), **posted** = (?var, pst2)]

?var	?var	pst2
f1	p1	pst2
f2	p1	pst2

matV[**reply** = (?var, pst2)]

?var	pst2
com1	pst2

Figure 5.7: Materialized views of query graph pattern Q_1, as generated by Algorithm TriC.

These two views contains all updates u_i that correspond to them, while the result of their join operation will be a new materialized view $matV[hasMod = (?var, ?var), posted = (?var, pst1)]$ as shown in Figure 5.7. In a similar manner, the subgraphs that satisfy path P_2 are calculated, while P_3 that is formulated by a single edge does not require any join operations. Finally, in order to calculate the subgraphs that match Q_1 all materialized views that correspond to paths P_1, P_2 and P_3 must be joined.

Finally, *the time complexity*, of joining two materialized views $matV_1$ and $matV_2$, where N and M denote their respective cardinalities, is $\mathcal{O}(N * M)$, since each tuple of $matV_1$ must be compared against each tuple of $matV_2$.

Indexing the Covering Paths (Step 2)

Subsequently, Algorithm TriC proceeds into indexing all the paths, extracted in *Step* 1, into a trie-based data structure. For each path $P_i \in CP(Q_i)$, TriC examines the forest for trie roots that can index the first edge $e_1 \in P_i$ (Figure 5.5, lines $3 - 7$). To access the trie roots, TriC utilizes a hash table (namely *rootInd*) that indexes the values of the root-nodes (keys) and the references to the root nodes (values), Figure 5.5, line 4. If such trie T_i is located, T_i is traversed in a *DFS* manner to determine in which sub-trie path P_i can be indexed (Figure 5.5,

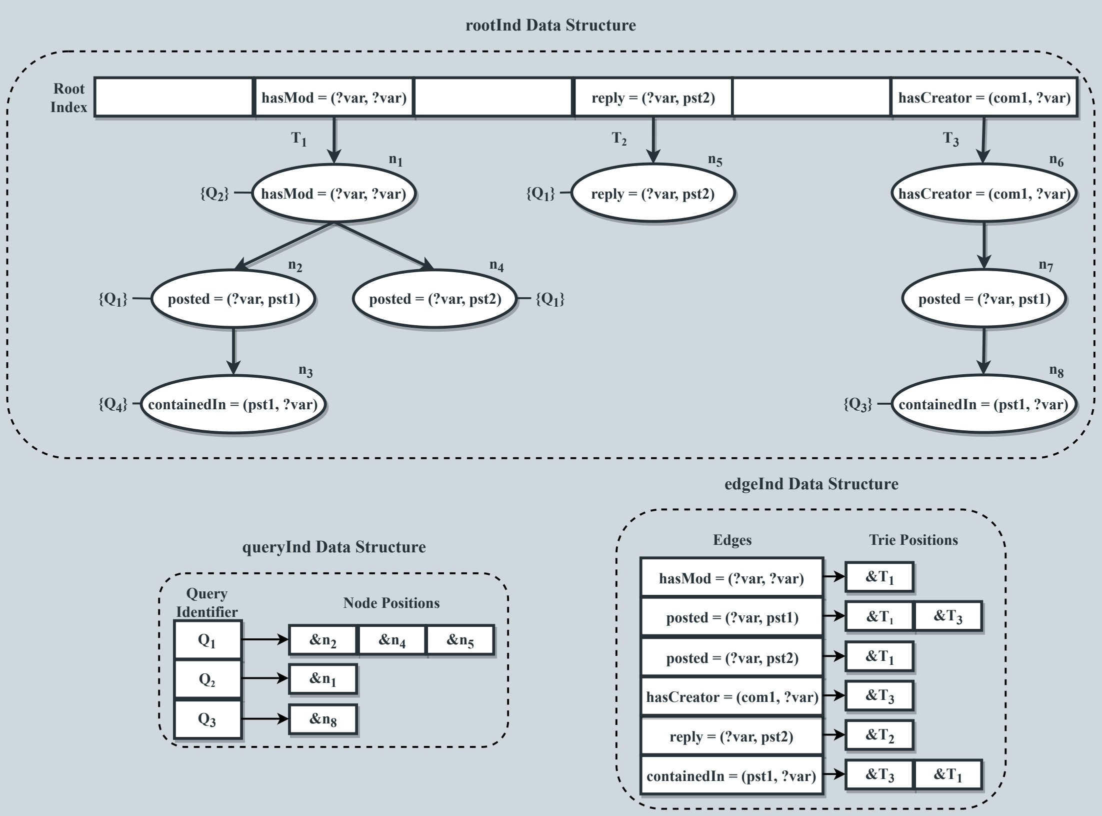


Figure 5.8: The data structures utilized by Algorithm TriC to index the set of query graph patterns.

line 5). Thus, Algorithm TriC traverses the forest to locate an existing trie-path $\{n_1 \rightarrow \ldots n_i \rightarrow \ldots n_k\}$ that can index the ordered set of edges $\{e_1, \ldots, e_k\} \in P_i$. If the discovered trie-path can index P_i partially (Figure 5.5, line 8), TriC proceeds into creating a set of new nodes under n_k that can index the remaining edges (Figure 5.5, line 9). Finally, the algorithm stores the identifier of Q_i at the last node of the trie path (Figure 5.5, line 10).

Algorithm TriC makes use of two additional data structures, namely *edgeInd* and *queryInd*. The former data structure is a hash table that stores each edge $e_i \in P_i$ (key) and a collection of trie roots T_i which index e_i as the hash table's value (Figure 5.5, lines $12 - 13$). Finally, TriC utilizes a matrix *queryInd* that indexes the query identifier along side the set of nodes under which its covering paths $P_i \in CP(Q_i)$ were indexed (Figure 5.5, line 14).

Example 5.4 *Figure 5.8 presents an example of rootInd, queryInd and edgeInd of*

Algorithm TriC *when indexing the set of covering paths of Figure 5.6 (b). Notice, that* TriC *indexes paths $P_1, P_2 \in Q_1$, path $P_1 \in Q_2$ and path $P_1 \in Q_4$ under the same trie T_1, thus, clustering together their common structural restrictions (all the aforementioned paths) and their attribute restrictions. Additionally, note that the queryInd data structure keeps references to the last node where each path $P_i \in Q_i$ is stored, e.g. for Q_1 it keeps a set of node positions $\{\&n_2, \&n_4, \&n_5\}$ that correspond to its original paths P_1, P_2 and P_3 respectively. Finally, edgeInd stores all the unique edges present in the path set of Figure 5.6 (b), with references to the trie roots under which they are indexed, e.g. edge posted $= (?var, pst1)$ that is present in $P_1 \in Q_1$, $P_1 \in Q_3$ and $P_1 \in Q_4$, is indexed under both tries T_1 and T_3, thus this information is stored in set $\{\&T_1, \&T_3\}$.*

The *time complexity* of Algorithm TriC when indexing a path P_i, where $|P_i| = M$ is the number of edges and B is the branching factor of the forest, is $\mathcal{O}(M * B)$, since TriC uses a DFS strategy, with the maximum depth bound by the number of edges. Thus, for a new query graph pattern Q_i with N covering paths, the total time complexity is $\mathcal{O}(N * M * B)$. Finally, the space complexity of Algorithm TriC when indexing a query Q_i is $\mathcal{O}(N * M)$, where M is the number of edges in a path and N is the cardinality of Q_i's covering paths.

Variable Handling

A query graph pattern Q_i contains vertices that can either be literals (specific entities in the graph) identified by their label, or variables denoted as "?var". This approach alleviates restrictions posed by naming conventions and thus leverages on the common structural constraints of paths.

However, by substituting the variable vertices with the generic "?var" requires to keep information about the joining order of each edge $e_i \in P_i$, as well as, how each $P_i \in CP(Q_i)$ intersects with the rest of the paths in $CP(Q_i)$. In order to calculate the subgraphs that satisfy each covering path $P_i \in CP(Q_i)$, each $matV[e_i] : e_i \in P_i$ must be joined. Each path P_i that is indexed under a trie path $\{n_1 \rightarrow \ldots n_i \rightarrow \ldots n_k\}$ maintains the original ordering of its edges and vertices, while the order under which each edge of a node n_i is connected with its children nodes ($chn(n_i)$) is determined as follows: the target vertex $t \in e_i$ (where e_i is indexed under n_i) is connected with the source node $s \in e_{i+1} : e_{i+1} \in chn(n_i)$ of the parent node n_i. Finally, for each covering path $P_i \in CP(Q_i)$ TRIC maintains information about the vertices that intersected in the original query graph pattern Q_i, while this information is utilized during the query answering phase.

5.3.2 Query Answering Phase

During the evolution of the graph, a constant stream of updates $S = (u_1, u_2, \ldots, u_k)$ arrives at the system. Each update u_i can affect a query graph pattern Q_i and potentially answer it. The query answering process of Algorithm TRIC commences by locating all the tries that are affected by update u_i and updates them by indexing u_i under them. Updating the tries, generated during the query indexing phase (Section 5.3.1), and locating which ones are affected, indicates the covering paths that have been affected by update u_i. Subsequently, each affected covering path is then utilized in the final phase of query answering, where it is determined if Q_i has matched. To this end, for each update $u_i \in S$ Algorithm TRIC performs the following distinct steps:

1. Determines which tries are affected by update u_i and proceeds in examining them.

2. While traversing the affected tries, performs the materialization and prunes sub-tries that are not affected by u_i.

In the following, we describe each step of the query answering phase of Algorithm TRIC. While, the pseudocode for each step of the query answering phase is provided in Figures 5.9 and 5.11.

Locate and Traverse Affected Tries (Step 1)

When an update u_i arrives at the system, Algorithm TriC utilizes the edge $e_i \in u_i$ to locate the tries that are affected by u_i. To achieve this, TriC uses the hash table *edgeInd* to obtain the list of tries that contain e_i in their children set. Thus, Algorithm TriC receives a list ($affectedTries$) that contains all the tries that were affected by u_i and must be examined (Figure 5.9, line 1). Subsequently, Algorithm TriC proceeds into examining each trie $T_i \in affectedTries$ by traversing each T_i in order to locate the node n_i that indexes edge $e_i \in u_i$. When node n_i is located, the algorithm stores all the corresponding positions of the indexed edges (Figure 5.9, lines $3-6$) and proceeds in $\mathcal{S}tep2$ of the query answering process described below (Figure 5.9, line 8).

Example 5.5 *Let us consider the data structures presented in Figure 5.8, the materialized views in Figure 5.10, and an update $u_1 = (posted = (p2, pst1))$ that arrives into the evolving graph (Figure 5.10(a)). Algorithm* TriC *prompts hash table edgeInd and obtains list $\{\&T_1, \&T_3\}$. Subsequently,* TriC *will traverse tries T_1 and T_3. When traversing trie T_1* TriC *locates node n_2 that matches update $e_1 \in u_1$ and proceeds in $\mathcal{S}tep2$ (described below). Finally, when traversing T_3* TriC *will stop the traversal at root node n_6 as its materialized view is empty $matV[hasCreator = (pst1, ?var)] = \emptyset$ (Figure 5.10 (b)), thus all sub-tries will yield empty materialized views.*

Trie Traversal and Materialization (Step 2)

Intuitively, a trie path $\{n_0 \rightarrow \ldots n_i \rightarrow \ldots n_k\}$ represents a series of joined materialized views $matVs = \{matV_1, matV_2\ , \ldots, matV_k\}$. Each materialized view $matV_i \in matVs$ corresponds to a node n_i that stores edge e_i and the materialized view $matV_i$. The materialized view contains the results of the join operation between the $matV[e_i]$ and the materialized view of the parent node n_i ($matV(prnt(n_i)]$), i.e., $matV_i = matV[prnt(n_i)] \bowtie matV[e_i]$. Thus, when an update u_i affects a node n_i in this "chain" of joins, n_i's and its children's ($chn(n_i)$) materialized views must

```
    Algorithm: ANSWER
    Input: Update $u_i = (e_i) : e_i = (s, t)$
    Output: Locate matched queries
    Step 1 :
    // Get affected tries
 1  $affectedTries \leftarrow edgeInd[e_i]$
 2  foreach $T_i \in affectedTries$ do
        // Traverse trie $T_i$ in a DFS manner
 3      foreach node $n_i \in T_i$ do
            // If the current node indexes edge $e_i$
 4          if $edge(n_c) = e_i$ then
                // Store the position
 5              $fndPos \leftarrow n$
                // Terminate the traversal
 6              break
        // Update $matV$s of $fndPos$ and its children
 7      $affectedQueries \leftarrow$ Trie Traversal & Materialization ($fndPos$)
 8  foreach query $Q_i \in affectedQueries$ do
 9      $results \leftarrow \emptyset$
        // For the covering paths of $Q_i$
10      foreach $P_i \in Q_i$ do
11          $results \leftarrow results \bowtie matV[P_i]$
12      if $results \neq \emptyset$ then
13          $mark\_Matched(Q_i)$
```

Figure 5.9: Pseudocode of the query answering phase (*Step* 1), performed by Algorithm TRIC.

be updated with u_i. Based on this, TRIC searches for and locates node n_i inside T_i that is affected by u_i and updates n_i's sub-trie.

After locating node $n_i \in T_i$ that is affected by u_i, Algorithm TRIC continues the traversal of n_i's sub-trie and prunes the remaining sub-tries of T_i (Figure 5.9, line 7). Subsequently, TRIC updates the materialized view of n_i by performing a join operation between its parent's node materialized view $matV[prnt(n_i)]$ and the update u_i, i.e., $results = matV[prnt(n_i)] \bowtie u_i$. Notice, that Algorithm TRIC calculates the subgraphs formulated by the current update solely based on the

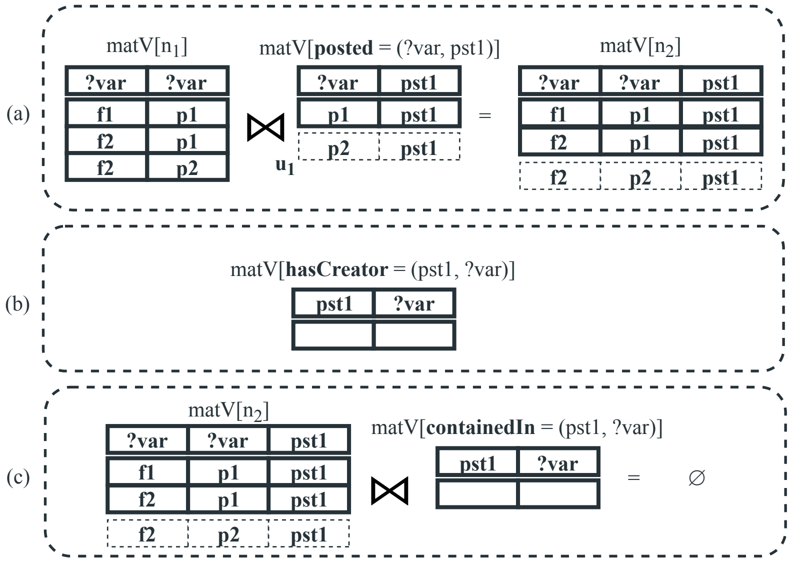


Figure 5.10: Updating materialized views.

update u_1 and does not perform a full join operation between $matV[prnt(n_i)]$ and $matV[edge(n_i)]$, the updated results are then stored in the corresponding $matV[n_i]$.

For each child node $n_j \in chn(n_i)$, TriC updates its corresponding materialized view by joining its view $matV[n_j]$ that corresponds to the edge that it stores (given by $matV[edge(n_j)]$) with its parent node materialized view $matV[n_i]$ (Figure 5.11, lines $1-7$). If at any point the process of joining the materialized views returns an empty result set the specific sub-trie is pruned, while the traversal continues in a different sub-trie of T_i (Figure 5.11, lines $5-6$). Subsequently, for each trie node n_j in the trie traversal, when there is a successful join operation among $matV[e_j] : e_j \in n_j$ and $matV[n_i]$, the query identifiers indexed under n_j are stored in $affectedQueries$ list (Figure 5.11, lines 4 and 7). Note that similarly to before, only the updated part of a materialized view is utilized as the parent's materialized view, an approach applied on database-management system [126].

Example 5.6 *Let us consider the data structures presented in Figures 5.8 and 5.10, and an update $u_1 = (posted = (p2, pst1))$ that arrives into the evolving graph. After locating the affected trie node n_2 (described in Example 5.5),* TriC *proceeds in updating the materialized view of n_2, i.e., $matV[n_2]$, by calculating the join operation between its parents materialized view, i.e., $matV[n_1]$ and the update*

```
Function: Trie Traversal & Materialization
Input: Node $n_i$
Output: Locate matched queries
// Update the current materialized view by joining the parent
   materialized view with the materialized view of the edge in
   node $n_i$
$result \leftarrow matV[prnt(n_i)] \bowtie matV[edge(n_i)]$;
if $result = \emptyset$ then
    return;
// Store the query identifiers of node $n_i$
$affectedQueries \leftarrow affectedQueries \cup qIDs(n_i)$;
// Recursively update the $matV$s of $n_i$'s children
foreach $n_c \in chn(n_i)$ do
    Trie Traversal & Materialization ($n_c$);
// Return the affected $qIDs$
return $affectedQueries$;
```

Figure 5.11: Pseudocode of the query answering phase (*Step* 2), performed by Algorithm TriC.

u_1. *Figure 5.10, demonstrates the operations of joining* $matV[n_2]$ *with update* u_1*, the result of the operation is tuple* $(f2, p2, pst1)$*, which is added into* $matV[n2]$*, presented in Figure 5.10(a). While the query identifiers of* n_2 *(i.e.,* Q_1*) are indexed in* $affectedQueries$*. Finally,* TriC *proceeds by updating the sub-trie of* n_2*, that is node* n_3*, where the updated tuple* $(f2, p2, pst1)$ *is joined with* $matV[edge(n_3)]$ *(i.e.,* $matV[containedIn = (pst1, ?var)]$*). This operation yields an empty result (Figure 5.10(c)), thus terminating the traversal.*

Finally, to complete the answering phase Algorithm TriC iterates through the affected list of queries and performs the join operations among the paths that form a query, thus, yielding the final answer (Figure 5.9, lines $8-13$).

The *time complexity* of Algorithm TriC when filtering an update u_i is calculated as follows: The traversal complexity is $\mathcal{O}(T*(P_m*B))$, where T denotes the number of tries that contain $e_i \in u_i$, P_m denotes the size of the longest trie path, and B is the branching factor. The time complexity of joining two materialized views $matV_1$

and $matV_2$, where $|matV_1| = N$ and $|matV_2| = M$, is $\mathcal{O}(N * M)$. Finally, the total time complexity is calculated as $\mathcal{O}((T * (P_m * B)) * (N * M))$.

5.3.3 Query Set and Graph Modifications

In this section, we briefly discuss how query set and graph modifications can potentially be handled by Algorithm TriC.

Algorithm TriC indexes a set of query graph patterns denoted as Q_{DB}, where $|Q_{DB}| = k$ (i.e., the number of query graph patterns indexed). Set Q_{DB} is subject to modifications, more specifically, query additions, deletions and updates. When a new query graph pattern Q_i is added in Q_{DB} the same indexing process is followed as described previously in Steps 1 and 2 , since the process of indexing a query graph patterns is independent of the existing query set.

When a query graph pattern Q_i is removed from the query set Q_{DB}, each covering path $P_i \in Q_i$ must be removed from the trie forest. The removal of the covering paths is achieved by gaining access to the corresponding nodes through the *queryInd* data structure. Subsequently, all nodes that correspond to path P_i are traversed and examined for removal. In addition, Algorithm TriC examines all affected nodes that do not facilitate the indexing of other paths (nodes rendered redundant) and thus removes them from forest, while it updates all edge entries $e_i \in Q_i$ in the *edgeInd* data structure. Furthermore, the *queryInd* table is updated and Q_i is removed. Finally, query updates can be handled in straightforward fashion, in the following two stages: (i) the removal of the original query Q_i from Q_{DB}, and (ii) the re-insertion of the updated query Q_i.

Algorithm TriC can support edge deletions in the same fashion as edge additions are handled during the query answering phase (as discussed above). More specifically, when an update u_d arrives at the system, Algorithm TriC can utilize the edge $e_d \in u_d$ and probe the hash table *edgeInd* to locate the tries that are affected by u_d and thus, receiving a list of affected tries ($affectedTries$). Subsequently, TriC can examine each affected trie (as described in $Step1$, Figure 5.9) and locate the node n_i that indexes $e_d \in u_d$. After locating node n_i that is affected by u_d, Algorithm TriC continues the traversal of the remaining sub-tries (similarly to *Step*2, Figure 5.11), and updates the affected materialized views, by removing all the tuples that contain u_d.

5.3.4 Implementing a Caching Approach

During the trie traversal and materialization (*Step*2, Figure 5.11), of the query answering phase, two materialized views are joined using a typical hash join operation with a build and a probe phase. In the build phase, a hash table for the smallest (in the number of tuples) table is constructed, while in the probe phase the largest table is scanned and the hash table is probed to perform the join. Algorithm TriC discards all the data structures and intermediate results after the join operation commences. In order to enhance this resource intensive operation, we cache and reuse the data structures generated during the build and probe phases as well as the intermediate results whenever possible. This approach constitutes an extension of our proposed solution, Algorithm TriC, and it is coined Algorithm TriC+.

In this section, we presented two novel query graph pattern indexing algorithms that utilize trie structures to cluster continuous graph pattern queries, namely Algorithms TriC and TriC+. In the following section, we present two advanced baseline solutions that use inverted index data structures, and a baseline solution that employs the well-established Neo4j graph database.

5.4 Advanced Baselines

Since no prior work in the literature considers the problem of continuous multi-query evaluation, we designed and implemented Algorithms Inv and Inc, two advanced baselines that utilize inverted index data structures. Finally, we provide a third baseline that is based on the well-established graph database Neo4j [28].

5.4.1 Algorithm INV

Algorithm Inv (INVerted Index), utilizes inverted index data structures to index the query graph patterns. The inverted index data structure is able to capture and index common elements of the graph patterns at the edge level during indexing time, as opposed to the more sophisticated clustering imposed by the use of tries. Subsequently, the inverted index is utilized during filtering time to determine which queries have been satisfied. In the following, we describe the query indexing and answering phase of Inv.

The Query Indexing Phase

The Query Indexing Phase of Algorithm Inv, for each query graph pattern Q_i, is performed in two steps:

1. Transforming the original query graph pattern Q_i into a set of path conjuncts, that cover all vertices and edges of Q_i and when combined can effectively re-compose Q_i, and indexing those covering paths in a matrix along the unique query identifier.

2. Indexing all edges $e_i \in Q_i$ into an inverted index data structure, as well as indexing all covering paths in a matrix, thus, maintaining a close representation of the original query.

In the following, we present each step of the query indexing phase of Algorithm Inv and give details about the data structures utilized. The complete pseudocode of the query indexing phase is provided in Figure 5.12.

Extracting the Covering Paths (Step 1)

In the first step of the query indexing phase, Algorithm Inv decomposes a query graph Q_i into a set of paths CP, a process described in detail in Section 5.3.1, (Figure 5.12, line 1). Thus, given the query set presented in Figure 5.6 (a), Inv yields the same set of covering paths CP (Figure 5.6 (b)). Finally, the covering path set CP is indexed into an array (*queryInd*) with the query identifier of Q_i (Figure 5.12, line 2).

```
  Algorithm: Index
  Input: Query $Q_i = (V_{Q_i}, E_{Q_i}, vars, l_V, l_E, \Sigma_V, \Sigma_E)$
  Result: Store $Q_i$ in $Q_{DB}$ ($Q_{DB} \leftarrow Q_{DB} \cup Q_i$)
  Step 1 :
  // Extract all paths that cover the query graph
1 $Paths \leftarrow CP(Q_i)$
  // Store all paths that cover the query graph
2 $queryInd[id(Q_i)] \leftarrow CP(Q_i)$
  Step 2 :
  // For all edges of $Q_i$
3 foreach edge $e_i \in Q_i$ do
      // Keep an entry
4     $edgeInd[e_i] \leftarrow id(Q_i)$
5     $sourceInd[src(e_i)] \leftarrow e_i$
6     $targetInd[trg(e_i)] \leftarrow e_i$
```

Figure 5.12: Pseudocode for the query indexing phase (Algorithm Index), performed by Algorithm Inv.

Indexing the Query Graph (Step 2)

Algorithm Inv builds three inverted indexes, where it stores all edges and vertices that formulate the query graph patterns, thus storing the structural and attribute constrains of the query graph pattern Q_i. Hash table *edgeInd* indexes all edges $e_i \in Q_{DB}$ (keys) and the respective query identifiers as values, (Figure 5.12, line 4), hash table *sourceInd* indexes the source vertices of each edge (key), where the edges are indexed as values (Figure 5.12, line 5), and hash table *targetInd* indexes the target vertices of each edge (key), where the edges are indexed as values (Figure 5.12, line 6).

In Figure 5.6(a), we present four query graph patterns and in Figure 5.13 we show the data structures of Inv when indexing those queries. Finally, Inv applies the same techniques of handling variables as Algorithm TriC, as described in Section 5.3.1.

The Query Answering Phase

During the evolution of the graph a constant stream of updates $S = (u_1, u_2, \ldots, u_k)$ arrives at the system. For each update $u_i \in S$ Algorithm Inv performs the

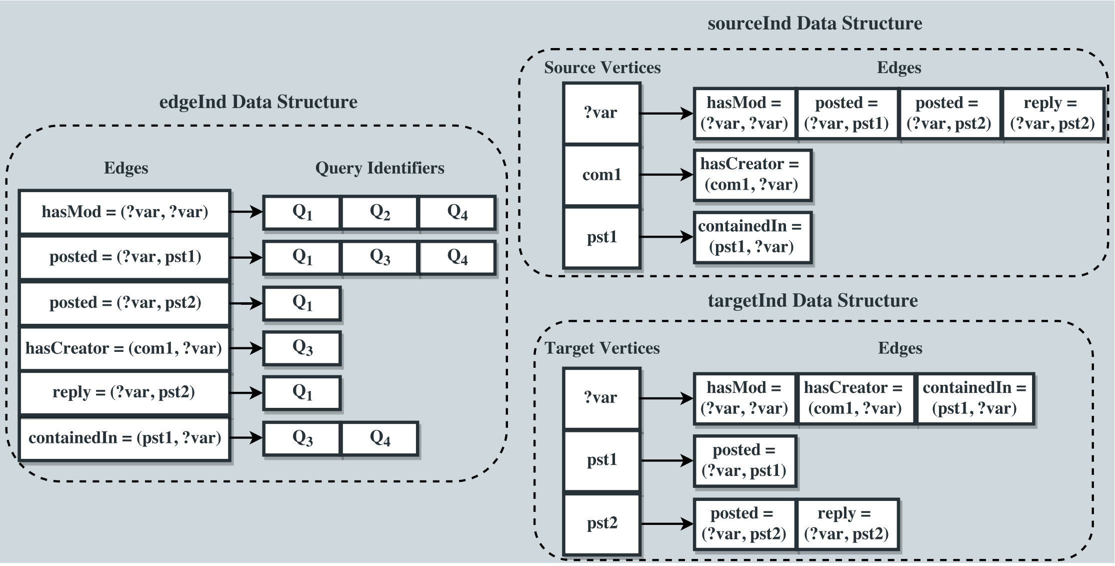


Figure 5.13: Index structures utilized by Algorithm Inv.

following three steps:

1. Determines which queries are affected by update u_i.

2. Prompts the inverted index data structures *sourceInd* and *targetInd* and determines which paths have been affected by update u_i.

3. Performs the materialization while querying the inverted index data structures.

In the following, we describe each step of the query answering phase of Algorithm Inv. The pseudocode for each step of the query answering phases is provided in Figures 5.14 (*Steps*1&2) and 5.15 (*Steps*2&3).

```
Algorithm: ANSWER
Input: Update $u_i = (e_i) : e_i = (s, t)$
Output: Locate matched queries
Step 1 :
// Get affected queries
1  $affectedQueries \leftarrow edgeInd[e_i]$
// For all affected queries
2  foreach $Q_i \in affectedQueries$ do
       // If there exists at least one edge that does not have a
          matterialised view
3      if $\exists e_j \in Q_i$ that $matV[e_j] = \emptyset$ then
           // Remove $Q_i$ from the candidate list
4          $affectedQueries \leftarrow affectedQueries \setminus \{Q_i\}$
Step 2 :
// Visit all edges $e_i$ that are affected by update $u_i$
5  foreach $e_i \in sourceInd[s] \cup targetInd[t]$ do
6      ExaminePath($e_i$)
7  foreach query $Q_i \in affectedQueries$ do
8      $results \leftarrow \emptyset$
       // For each covering path $p_i \in Q_i$
9      foreach $p_i \in Q_i$ do
10         $results \leftarrow results \bowtie matV[p_i]$
11     if $results \neq \emptyset$ then
12         $mark\_Matched(Q_i)$
```

Figure 5.14: Pseudocode of the query answering phase (*Steps* 1 & 2), performed by Algorithm INV.

Locate the Affected Queries (Step 1)

When a new update u_i arrives at the system, Algorithm INV utilizes the edge $e_i \in u_i$ to locate the queries that are affected, by querying the hash table *edgeInd* to obtain the query identifier *qIDs* that contain e_i (Figure 5.14, line 1). Subsequently, the algorithm iterates through the list of *affectedQueries* and checks each query $Q_i \in qIDs$. For each query Q_i the algorithm checks $\forall e_i \in Q_i$ if $matV[e_i] \neq \emptyset$, i.e., each e_i should have a *non empty* materialized view. The check is performed

```
Function: ExaminePath
Input: Edge $e_c$
Output: Locate matched queries
Steps 2 & 3 :
// Mark edge as visited
$visited \leftarrow visited \cup e_c$
// Keep the current path
$currentPath$.push($e_c$)
// If the current edge $e_c$ is not part of the affected queries
   set, terminate the traversal
if $qIDs \cap edgeInd[e_c] = \emptyset$ then
    return
// Join the result with materiliased view of the current edge
$result \leftarrow result \bowtie matV[e_c]$
// If the results are not empty
if $(result \neq \emptyset)$ then
    // For each query that contains the current edge $e_c$ mark the
       path that this edge is contained as matched
    $\forall Q_i$ contains $currentPath$, $mark\_Matched(Q_i, currentPath)$
// Visit all edges $e_i$ adjacent to $e_c$
foreach $e_i \in sourceInd[src(e_c)] \cup targetInd[trg(e_c)]$ do
    ExaminePath($e_i$)
// Update the current path
$currentPath$.pop($e_c$)
```

Figure 5.15: Pseudocode of the query answering phase (***Steps*** 2 & 3), performed by Algorithm Inv.

by iterating through the edge list that is provided by *queryInd* and a hash table that keeps all materialized views present in the system. Intuitively, a query Q_i is candidate to match, as long as, all materialized views that correspond to its edges can be used in the query answering process (Figure 5.14, lines 2 – 4).

Locate the Affected Paths (Step 2)

Algorithm Inv proceeds to examine the inverted index structures *sourceInd* and *targetInd* by making use of $e_i \in u_i$ to determine which edges are affected by the update, by utilizing the source and target vertices of update u_i. Subsequently, Algorithm Inv examines each edge of the affected edge set (Figure 5.14, lines $5-6$).

Algorithm Inv examines each current edge e_c of the affected edge set and recursively visits all edges connected to e_c, which are determined by querying the *sourceInd* and *targetInd* (Figure 5.15, lines $8-9$). While examining the current edge e_c, Inv checks if e_c is part of $affectedQueries$, if not, the examination of the specific path is pruned (Figure 5.15, lines $3-4$). For efficiency reasons, the examination is bound by the maximum length of a path present in $affectedQueries$, which is calculated by utilizing the *queryInd* data structure.

Path Examination and Materialization (Step 3)

While Algorithm Inv examines the paths affected by update u_i ($\mathcal{S}tep$ 2), it performs the materialization on the currently examined path, (Figure 5.15, line 5). More specifically, while Inv searches through the paths formulated by the visits of edge sets determined by *targetInd* and *sourceInd*, it maintains a path $P_c = \{v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \dots v_k \xrightarrow{e_k} v_{k+1}\}$ that corresponds to the edges already visited (Figure 5.15, lines $2, 10$).

While, visiting each edge e_c, Inv accesses the materialized view that corresponds to it (i.e., $matV[e_c]$) and updates the set of materialized views $matVs = \{matV_1, matV_2, \dots, matV_k\}$ that correspond to the current path. For example, given an already visited path $P = \{v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} v_3\}$ its materialized view $matV[P]$ will be generated by $matV[P] = matV[e_1] \bowtie matV[e_2]$. When visiting the next edge e_n, a new path P' is generated and its materialized view $matV[P'] = matV[P] \bowtie matV[e_n]$ will be generated. If at any point, the process of joining the materialized views yields an empty result set the examination of the edge is terminated (pruning), while the examination continues in different directions (Figure 5.15, lines $6-7$). This allows us to prune paths that are not going to satisfy any $Q_i \in affectedQueries$ and avoid unnecessary edge visits. If a path P_i yields a successful series of join operations (i.e., $matV[P_i] \neq \emptyset$), it is marked as matched (Figure 5.15, line 7).

Finally, to produce the final answer subgraphs Algorithm Inv iterates through the affected list of queries $qIDs \in affectedQueries$ and performs the final join operation among all the paths that comprise the query, thus producing the final answer subgraphs that satisfied the query (Figure 5.14, lines $7 - 12$).

The *time complexity* of Algorithm Inv, when filtering an update u_i, is calculated as follows: The path examination complexity is $\mathcal{O}(V * B^D)$, where E denotes the number of vertices connected by $e_i \in u_i$, B is the size of the edge set reachable when initiating the path examination, and D denotes the size of the longest path in the graph. Finally, taking into consideration the time complexity of the materialization process, the total time complexity is calculated as $\mathcal{O}((V * B^D) * (N * M))$, since the materialization occurs in each edge visit.

Query Set and Graph Modifications

Similarly to Algorithm TriC (described in detail in Section 5.3.1), the indexed query set Q_{DB} can be subject to modifications, i.e., insertions, deletions and updates. In the case of inserting a new query Q_i in Q_{DB}, the same indexing process is followed as described in *Steps* 1 and 2. When a query Q_i is removed from Q_{DB} the covering paths are removed from the *queryInd* data structure, while these paths are utilized to locate the edge and vertex entries of Q_i present in *edgeInd*, *sourceInd* and *targetInd* that are updated accordingly. Finally, when updating a query graph Q_i, the operation is handled in two stages: (i) the removal of Q_i from Inv's data structures, and (ii) the insertion of a new Q_i.

Algorithm Inv can support edge deletions, in the same fashion as edge additions are handled during the query answering phase. More specifically, when an update u_d arrives at the system, Algorithm Inv can utilize the *sourceInd* and *targetInd* indexes to determine which edges are affected by the update, as described in ***Step*** 2. Subsequently, Inv can traverse each affected path (***Step*** 3) and locate all the paths that contain $e_d \in u_d$. While visiting each edge e_c Algorithm Inv accesses the materialized view that corresponds to e_c and removes the tuples affected by update u_d.

Implementing a Caching Approach

In the spirit of Algorithm TRIC+ (described in Section 5.3.4), we developed an extension of Algorithm INV, namely Algorithm INV+, that caches and reuses the calculated data structures of the hash join phase.

5.4.2 Algorithm INC

Based on Algorithm INV we developed an algorithmic extension, namely Algorithm INC. Algorithm INC utilizes the same inverted index data structures to index the covering paths, edges, source and target vertices as Algorithm INV, while the examination of a path affected during query answering remains similar. The key difference lies in executing the joining operations between the materialized views that correspond to edges belonging to a path. More specifically, when Algorithm INV executes a series of joins between the materialized views (that formulate a path) to determine which subgraphs match a path, it utilizes all tuples of each materialized view that participate in the joining process. On the other hand, Algorithm INC makes use only of the update u_i and thus, reduces the number of tuples examined throughout the joining process of the paths.

Implementing a Caching Approach

In the spirit of Algorithm TRIC+ (described in Section 5.3.2), we developed an extension of Algorithm INC, namely Algorithm INC+, that caches and reuses the calculated data structures of the hash join phase.

5.4.3 Neo4j

To evaluate the efficiency of the proposed algorithm against a real-world approach, we implemented a solution based on the well-established graph database Neo4j [28]. In this approach, we extend Neo4j's native functionality with auxiliary data structures to efficiently store the query set. These data structures are used during the answering phase to efficiently locate affected queries and execute them on the Neo4j's query execution engine.

The Query Indexing Phase

To address the continuous multi-query evaluation scenario, we designed main-memory data structures to facilitate indexing of query graph patterns. Initially, in the preprocessing phase, we convert each incoming query Q_i into Neo4j's native query language Cypher [127]. Subsequently, the query indexing phase of Neo4j commences as follows: (1) indexing each Cypher query in the *queryInd* data structure and (2) indexing all edges $e_i \in Q_i$ in the *edgeInd* data structure, where e_i is used as key and a collection of query identifiers as values. The *queryInd* structure is defined as matrix, while the *edgeInd* is an inverted index, similarly to the data structures described in Section 5.4.1.

The Query Answering Phase

Each update that is received as part of an incoming stream of updates $S = (u_1, u_2, \ldots, u_k)$ is processed in the following steps: (1) an incoming update u_i is applied to Neo4j through the graph database service, (2) the inverted index *edgeInd* is queried with $e_i \in u_i$, to determine which queries are affected, (3) all affected queries are retrieved from matrix *queryInd*, (4) the affected queries are executed.

To enhance the query answering performance, the following configurations are applied: (1) the graph database builds indexes on all labels of the schema allowing for faster look up times of nodes, (2) the execution of Cypher queries employs the *parameters syntax* [128] as it enables the execution planner of Neo4j to cache the query plans for future use, (3) the number of writes per transaction [129] in the database and the allocated memory were optimized based on the hardware configuration (see Section 5.5.1).

In this section, we presented three advanced baseline solution, two of which employ inverted index data structures (Algorithms Inv and Inc) and one based on the graph database Neo4j. In the following section, we present the experimental evaluation results when comparing Algorithm TriC against Algorithms Inv, Inc and Neo4j.

5.5 Experimental Evaluation

In this section, we present a series of experiments that compare Algorithm TriC against Algorithms Inv, Inc and the Neo4j approach. To this end, we present the data and query sets, the algorithmic and technical configuration, and the metrics employed and finally, we present and extensively discuss the experimental evaluation results.

5.5.1 Experimental Setup

In this section, we present the data and query sets, the algorithmic and technical configuration, and the metrics employed in our experimental evaluation.

Data and Query Sets

For the experimental evaluation we used a synthetic and two real-world datasets, which we proceed to describe below.

The SNB Dataset

The first dataset we utilized is the LDBC Social Network Benchmark (SNB) [123]. SNB is a synthetic benchmark designed to accurately simulate the evolution of a social network through time (i.e., vertex and edge sets labels, event distribution etc). This evolution is modeled using activities that occur inside a social network (i.e., user account creation, friendship linking, content creation, user interactions etc). In this synthetic benchmark, chosen as it fits the proposed paradigm, the network evolves naturally through time with the addition of new information generated from activities of the users, while the changes occur in a streaming fashion. Based on the SNB generator, we simulated the evolution of a graph consisting of user activities over a time period of 2 years. From this dataset, we derived 3 query loads and configurations: (i) a set with a graph size of $|G_E| = 100K$ edges and $|G_V| = 57K$ vertices, (ii) a set with a graph size of $|G_E| = 1M$ edges and $|G_V| = 463K$ vertices, and (iii) a set with a graph size of $|G_E| = 10M$ edges and $|G_V| = 3.5M$ vertices.

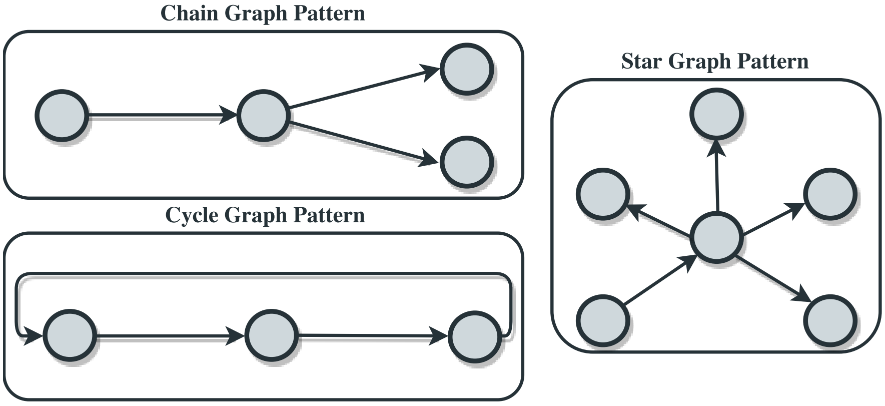


Figure 5.16: Examples of the query graph pattern classes.

The NYC Dataset

The second dataset we utilized is a real world set of taxi rides performed in New York City[130] (*NYC*) in 2013 utilized in *DEBS 2015 Grand Challenge* [**debs2015**]. *NYC* contains more that $160M$ entries of taxi rides with information about the license, pickup and drop-off location, the trip distance, the date and duration of the trip, and the fare. We utilized the available data to generate a stream of updates that result in a graph of $|G_E| = 1M$ edges and $|G_V| = 280K$, accompanied by a set of $5K$ query graph patterns.

The BioGRID Dataset

The third dataset we utilized is BioGRID [131], a real world dataset that represents protein to protein interactions. The BioGRID repository is a dataset that contains entries that describe interaction between protein pairs. In this scenario, proteins are modeled as vertices while the relations model the interactions observed. As only a single type of relation between vertices is present (interacts), and the type of each vertex is solely one (a protein), this dataset serves as a stress test for the algorithms developed, as each update affects the whole query database, thus triggering a search of query answered in the entire database. To this end, we used BioGRID to generate a stream of updates that result in a graph size of $|G_E| = 1M$ edges and $|G_V| = 63K$ vertices, with a set of $5K$ query graph patterns.

Query Set Configuration

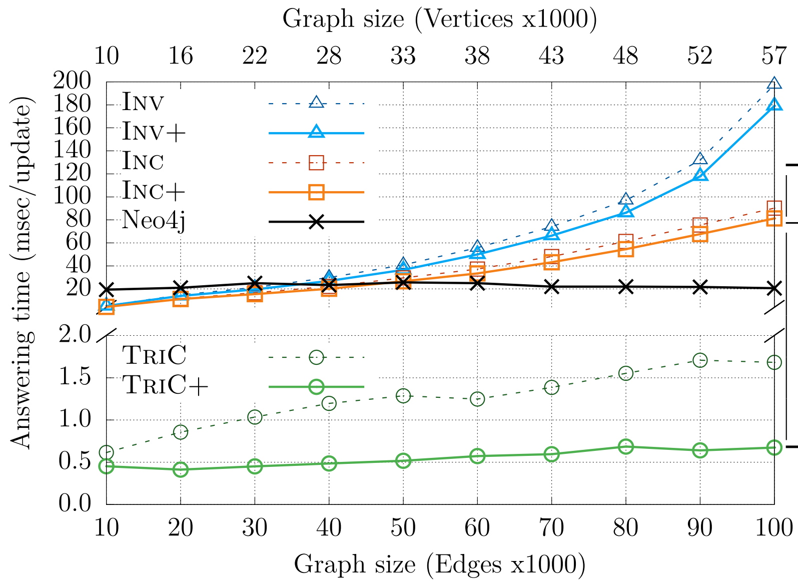
In order to construct the set of query graph patterns Q_{DB} we identified three distinct query classes that are typical in the relevant literature: chains, stars, and cycles [107, 132], presented in Figure 5.16. Each type of query graph pattern was chosen equiprobably during the generation of the query set. The baseline values for the query set are: (i) an average size ℓ of 5 edges/query graph pattern, a value derived from the query workloads presented in SNB [123], (ii) a query database $|Q_{DB}|$ size of $5K$ graph patterns, (iii) a factor that denotes the percentage of the query set Q_{DB} that will ultimately be satisfied, denoted as selectivity $\sigma = 25\%$, and (iv) a factor $o = 35\%$ that denotes the percentage of overlap between the queries in the set.

Metrics

In our evaluation, we present and discuss the filtering time of each algorithm, that is the amount of time needed to locate all continuous graph patterns that are satisfied by an incoming update u_i. We also, present and compare the memory requirements of the algorithms. Finally, we present the indexing time of each algorithm, i.e., the amount of time needed to index a set of query graph patterns into the database.

Technical Configuration

All algorithms were implemented in Java 8 while for the materialization implementation the Stream API was employed. The Neo4j-based approach was implemented using the embedded version of Neo4j 3.4.7. Extensive experimentation evaluation concluded that a transaction [129] can perform up to $20K$ writes in the database without degrading Neo4j's performance, while in order to guarantee indexes are cached in main memory $55GB$ of main memory were allocated. A machine with Intel(R) Xeon(R) Processor E5-2650 at $2.00GHz$, $64GB$ RAM, and Ubuntu Linux 14.04 was used. The time shown in the graphs is wall-clock time and the results of each experiment are averaged over 10 runs to eliminate fluctuations in measurements.



Parameter	Value
$\lvert Q_{DB} \rvert$	$5K$
$\lvert G_E \rvert$	$10K - 100K$
ℓ	5
σ	25%
o	35%

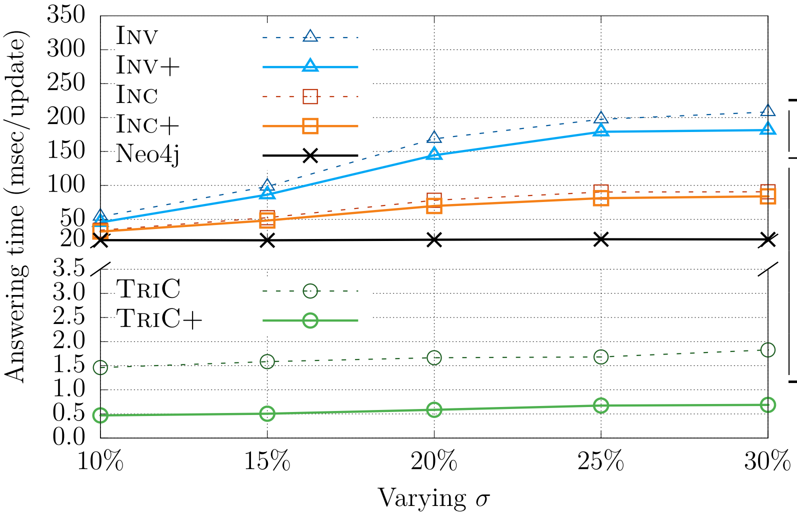
Figure 5.17: Examining the influence of graph size, while comparing the query answering time, when varying the graph size from $|G_E| = 10K$ to $|G_E| = 100K$ edges.

5.5.2 Results for the SNB Dataset

In this section, we present the evaluation for the *SNB* benchmark and highlight the most significant findings.

Query Answering Time.

Figure 5.17 presents the results regarding the query answering time, i.e., the average time in milliseconds needed to determine which queries are satisfied by an incoming update, against a query set of $Q_{DB} = 5K$. Please notice that the y-axis is split due to the high differences in the performance of Algorithms TRIC/TRIC+ and its competitors. We observe that the answering time increases for all algorithms as the graph size increases. Algorithms TRIC and TRIC+ achieve the lowest answering times suggesting better performance. Contrary, the competitors are more sensitive in graph size changes, with Algorithm INV performing the worst (highest query answering time). When comparing Algorithm TRIC to Algorithms INV, INC and Neo4j, the query answering time is improved by 99.15%, 98.14% and 91.86% respectively, while the improvement between INC and INV is 54.33%. Finally, comparing Algorithm TRIC+ to INV+, INC+ and Neo4j, TRIC+ demonstrates a



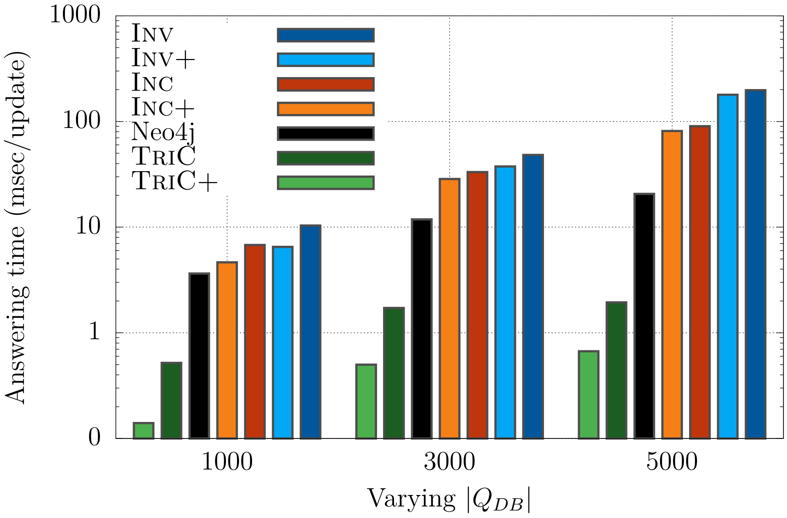
Parameter	Value
$\|Q_{DB}\|$	$5K$
$\|G_E\|$	$100K$
ℓ	5
σ	$10\% - 30\%$
o	35%

Figure 5.18: Examining the influence of σ, while comparing the query answering time, for a graph size $|G_E| = 100K$ edges.

performance improvement of 99.62%, 99.17% and 96.74% respectively, while the difference of Inc+ and Inv+ is 54.6%.

The results (Figure 5.17) suggest that all solutions that implement caching are faster compared to the versions without it. In more detail, Algorithms TriC+, Inv+, and Inc+ are consistently faster than their non-caching counterparts, by 59.95%, 9.36% and 9.91% respectively. This is attributed to the fact that Algorithms TriC, Inv, and Inc, have to recalculate the probe and build structures required for the joining process, in contrast to Algorithms TriC+, Inv+, and Inc+ that store these structures and incrementally update them, thus providing better performance.

In Figure 5.18 we present the results when varying the selectivity parameter σ, for 10%, 15%, 20%, 25% and 30% of a query set for $|Q_{DB}| = 5K$ and $|G_E| = 100K$. Varying σ affects the percentage of queries that match during the evolution of the graph; a higher number of queries satisfied, increases the number of calculations performed by each algorithm. The results show that all algorithms behave in a similar manner as previously described. In more detail, Algorithm TriC+ is the most efficient of all, and thus the fastest among the extensions that utilize caching, while TriC is the most efficient solution among the solutions that do not employ



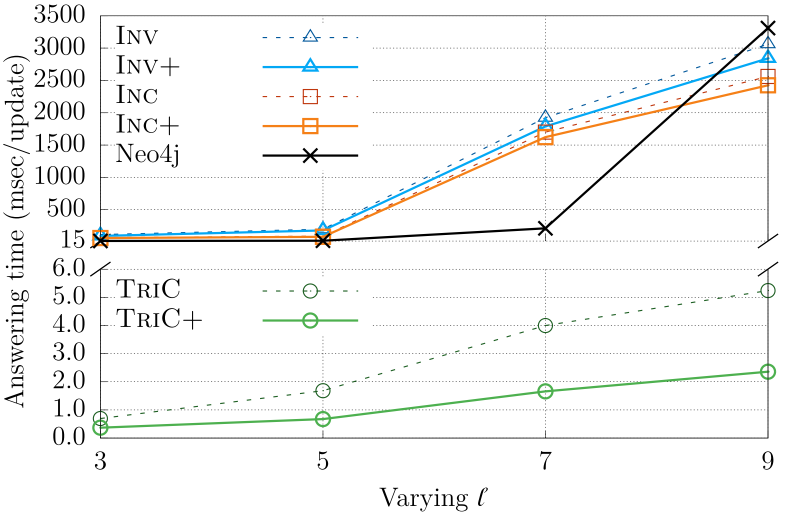
Parameter	Value
$\|Q_{DB}\|$	$1K - 5K$
$\|G_E\|$	$100K$
ℓ	5
σ	25%
o	35%

Figure 5.19: Examining the influence of the query database size, while comparing the query answering time, for a graph size $|G_E| = 100K$ edges.

a caching strategy. Finally, the percentage differences between the algorithmic solutions remain the same as before in most cases.

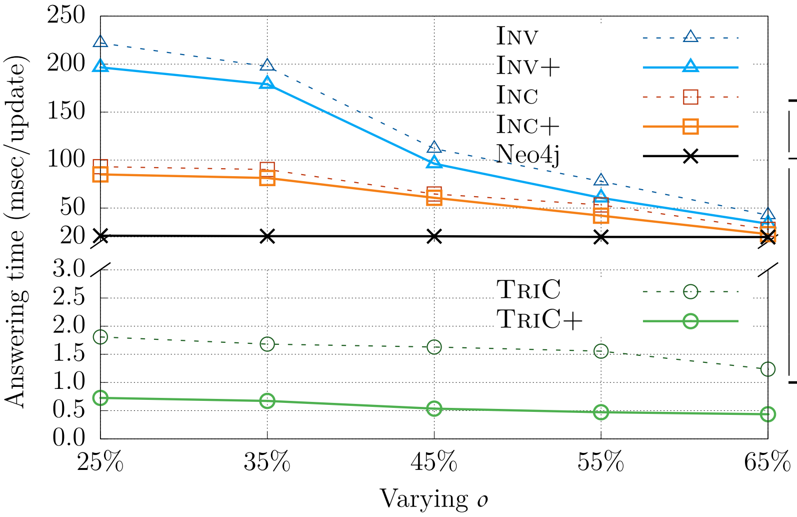
In Figure 5.19 we give the results of the experimental evaluation when varying the size of the query database $|Q_{DB}|$. More specifically, we present the answering time per triple when $|Q_{DB}| = 1K$, $3K$ and $5K$, and $|G_E| = 100K$. Please notice the y-axis is in logarithmic scale. The results demonstrate that the behavior of all algorithms is aligned with our previous observations. More specifically, Algorithms TriC+ and TriC exhibit the highest performance (i.e., lowest answering time), throughout the increase of $|Q_{DB}|$, and thus determine faster which queries of $|Q_{DB}|$ have matched given an update u_i. Similarly to the previous setups, the competitors have lower performance, while Algorithms Inc and Inc+ perform better compared to Algorithms Inv and Inv+.

In Figure 5.20 we give the results of the experimental evaluation when varying the average query size ℓ. More specifically, we present the answering time per triple when ℓ = 3, 5, 7 and 9 of a query set for $|Q_{DB}| = 5K$ and $|G_E| = 100K$. We observe that the answering time increases for all algorithms as the average query length increases. More specifically, Algorithms TriC+ and TriC exhibit the highest performance (i.e., lowest answering time), throughout the increase of ℓs, and



Parameter	Value
$\|Q_{DB}\|$	$5K$
$\|G_E\|$	$100K$
ℓ	$3-9$
σ	25%
o	35%

Figure 5.20: Examining the influence of ℓ, while comparing the query answering time, for a graph size $|G_E| = 100K$ edges.



Parameter	Value
$\|Q_{DB}\|$	$5K$
$\|G_E\|$	$100K$
ℓ	5
σ	25%
o	25% – 65%

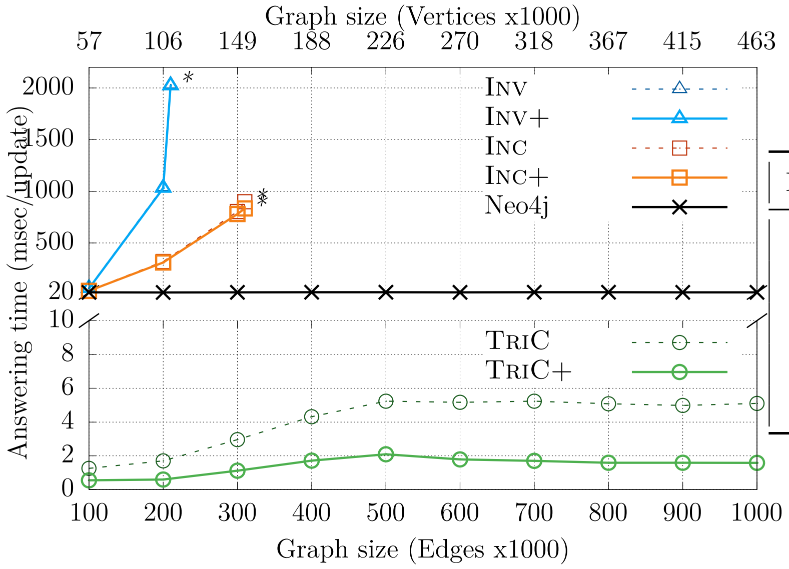
Figure 5.21: Examining the influence of o, while comparing the query answering time, for a graph size $|G_E| = 100K$ edges.

thus determine faster which queries have been satisfied. Similarly to the previous evaluation setups, the Algorithms INV, INV+, INC, INC+, and Neo4j have the lowest performance, and increase significantly their answering time when ℓ increases, while Algorithms INC and INC+ perform better compared to Algorithms INV, INV+ and Neo4j when $\ell = 9$.

In Figure 5.21 we give the results of the experimental evaluation when varying the overlapping factor o, for 25%, 35%, 45%, 55% and 65% of a query set for $|Q_{DB}| = 5K$ and $|G_E| = 100K$ edges. A higher number of query overlap should decrease the number of calculations performed by algorithms designed to exploit commonalities among the query set. The results show that all algorithm behave in a similar manner as previously described, while Algorithms Inv, Inv+, Inc, and Inc+ observe higher performance gains. Algorithm TriC+ is the most efficient of all, and thus the fastest among the extensions that utilize caching techniques, while TriC is the most efficient solution among the solutions that do not employ caching.

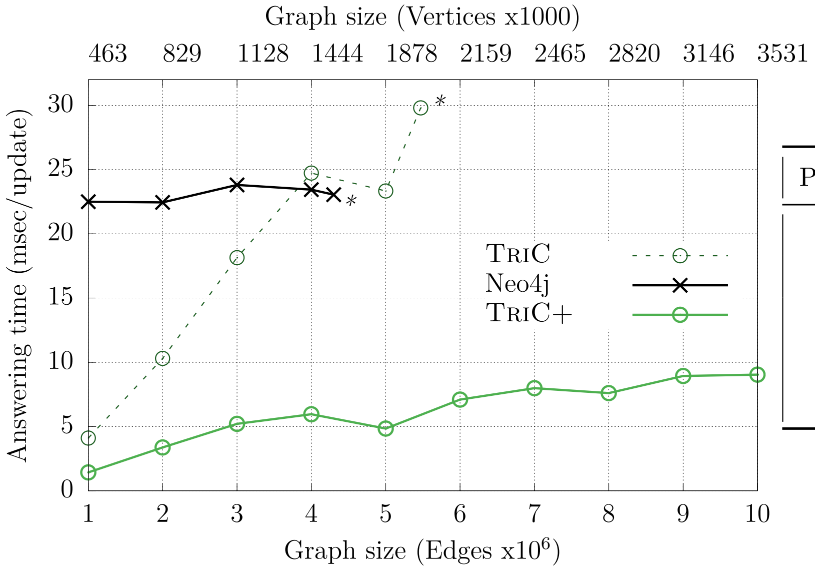
Figure 5.22 presents the results regarding the query answering time for all algorithms when indexing a query set of $|Q_{DB}| = 5K$ and a final graph of $|G_E| = 1M$ and $|G_V| = 463K$. Given the extremely slow performance of some algorithms we have set an *execution time threshold* of 24 hours for all algorithms under evaluation, thus, when the threshold was crossed the evaluation was terminated. We again observe that the answering time increases for all algorithms as the graph size increases. Algorithms TriC and TriC+ achieve the lowest answering times, suggesting better performance, while Algorithms Inv, Inv+, Inc, and Inc+ are more sensitive in graph size changes and thus fail to terminate within the time threshold. More specifically, Algorithms Inv and Inv+ *time out* at $|G_E| = 210K$, while Algorithms Inc and Inc+ *time out* at $|G_E| = 310K$ as denoted by the asterisks in the plot. When comparing Algorithms TriC and TriC+ to Neo4j the query answering is improved by 77.01% and 92.86% respectively.

Figure 5.23 presents the results regarding the query answering time for Algorithms TriC, TriC+ and Neo4j when indexing a query set of $Q_{DB} = 5K$ and a final graph size of $|G_E| = 10M$ and $|G_V| = 3.5M$. Again, we have set an *execution time threshold* of 24 hours for all the algorithms under evaluation. Algorithm TriC+ achieves the lowest answering times, suggesting better performance, while Algorithms TriC and Neo4j fail to terminate within the given time threshold. More specifically, Algorithm TriC *times out* at $|G_E| = 5.47M$, while Algorithm Neo4j times out at $|G_E| = 4.3M$ as denoted by the asterisks in the plot.



Parameter	Value
$\lvert Q_{DB}\rvert$	$5K$
$\lvert G_E\rvert$	$100K - 1M$
ℓ	5
σ	25%
o	35%

Figure 5.22: Examining the influence of graph size, while comparing the query answering time, when varying the graph size from $|G_E| = 100K$ to $|G_E| = 1M$ edges.



Parameter	Value
$\lvert Q_{DB}\rvert$	$5K$
$\lvert G_E\rvert$	$1M - 10M$
ℓ	5
σ	25%
o	35%

Figure 5.23: Examining the influence of graph size, while comparing the query answering time, when varying the graph size from $|G_E| = 1M$ to $|G_E| = 10M$ edges.

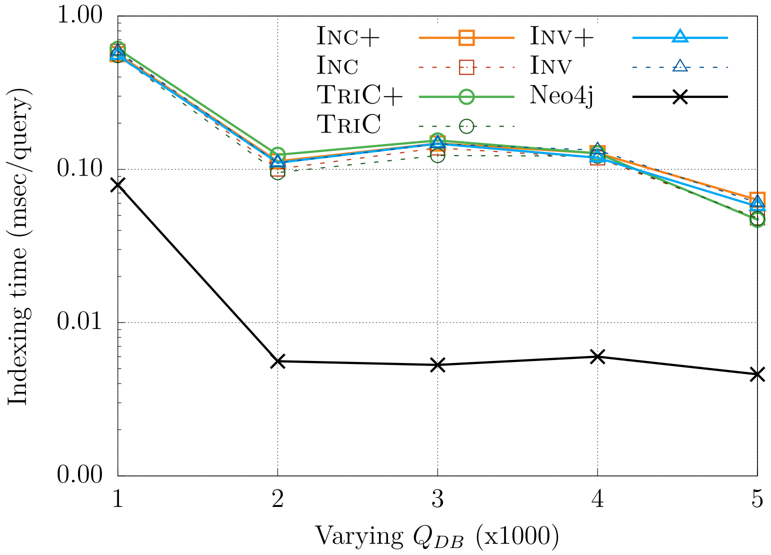
Overall, Algorithms TriC+ and TriC, the two solutions that utilize trie structures to capture and index the common structural and attribute restrictions of query graphs, achieve the lowest query answering times compared to Algorithms Inv, Inv+, Inc, and Inc+ that employ no clustering techniques, as well as when compared with commercial solutions such as Neo4j. Adopting the incremental joining techniques (found in Algorithm TriC) into Algorithm Inc does not seem to significantly improve its performance when compared to Algorithm Inv. Additionally, it is evident that the usage of only the updated results during the joining operations of algorithms, a technique designed originally for Algorithm TriC and applied on Algorithm Inv through its extension Algorithm Inc, changes significantly the performance of Algorithm TriC+. Taking all the above into consideration, we conclude that the algorithms that utilize trie-based indexing capture the common elements of query graph patterns during the indexing phase, and exploit this overlap during the query answering phase, thus resulting in extremely low query answering times compared to the baseline solutions, i.e., Algorithms Inv, Inv+, Inc, and Inc+.

Indexing Time

Figure 5.24 presents the indexing time in milliseconds required to insert $1K$ query graph patterns when the query database size increases. We observe that the time required to go from an empty query database to a query database of size $1K$ is higher compared to the time required for the next iterations. Please notice the y-axis is in logarithmic scale. This can be explained as follows: All algorithms utilize data structures that need to be initialized during the initial stages of query indexing phase, i.e. when inserting queries in an empty database, while, as the queries share common restrictions, less time is required for creating new entries in the existent data structures. Additionally, the time required to index a query graph pattern in the database does not vary significantly for all algorithms. Notice that query indexing time is not a critical performance parameter in the proposed paradigm, since the most important dimension is query answering time.

5.5.3 Results for the NYC and BioGRID Dataset

In this section, we present the evaluation for the *NYC* and *BioGRID* dataset and highlight the most significant findings.



Parameter	Value
$\lvert Q_{DB} \rvert$	$1K - 5K$
$\lvert G_E \rvert$	–
ℓ	5
σ	25%
o	35%

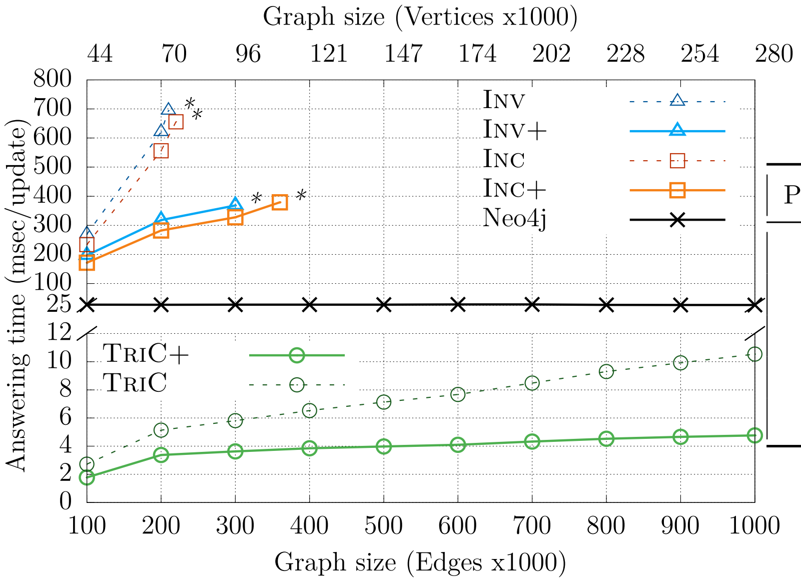
Figure 5.24: Comparing the insertion time, when varying the query database size Q_{DB}, for a query graph pattern set of $\ell = 5$, $\sigma = 25\%$, and $o = 35\%$.

The *NYC* Dataset

Figure 5.25 presents the results from the evaluation of the algorithms for the *NYC* dataset. More specifically, we present the results regarding the query answering performance of all algorithms when $Q_{DB} = 5K$, $\ell = 5$, $o = 35\%$, $\sigma = 25\%$ and an execution time threshold of 24 hours. Please notice that the y-axis is split due to high differences in the performance of the algorithms. Algorithms Inv and Inv+ fail to terminate within the time threshold and *time out* at $|G_E| = 210K$ and $|G_E| = 300K$ respectively. Similarly, Algorithms Inc and Inc+ *time out* at $|G_E| = 220K$ and $360K$ respectively. When comparing Algorithms TriC and TriC+ to Neo4j the query answering is improved by 59.68% and 81.76% respectively. These results indicate again that an algorithmic solution that exploits and indexes together the common parts of query graphs (i.e., Algorithms TriC and TriC+) achieves significantly lower query answering time compared to approaches that do not apply any clustering techniques (i.e., Algorithms Inv, Inv+, Inc, Inc+, and Neo4j).

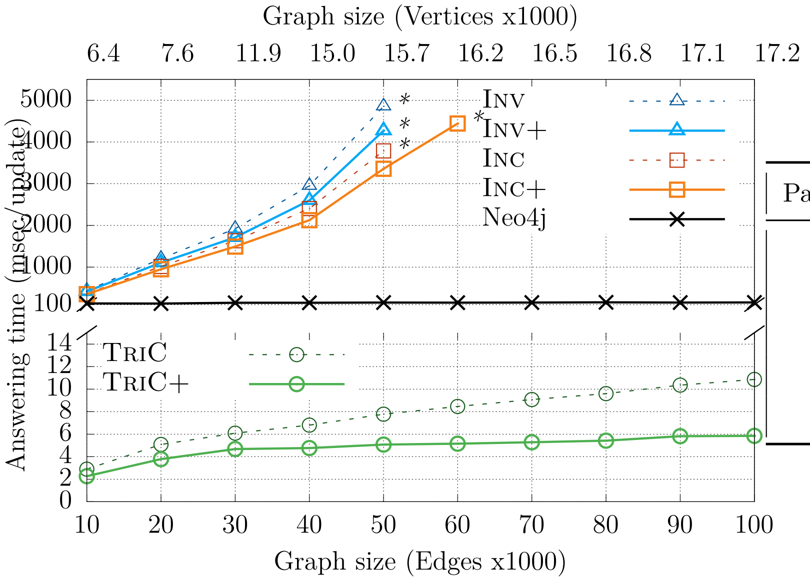
The *BioGRID* Dataset

Figures 5.26 and 5.27 present the results from the evaluation of the algorithms for the *BioGRID* dataset. In Figure 5.26 we present the results regarding the query



Parameter	Value
$\lvert Q_{DB}\rvert$	$5K$
$\lvert G_E\rvert$	$100K - 1M$
ℓ	5
σ	25%
o	35%

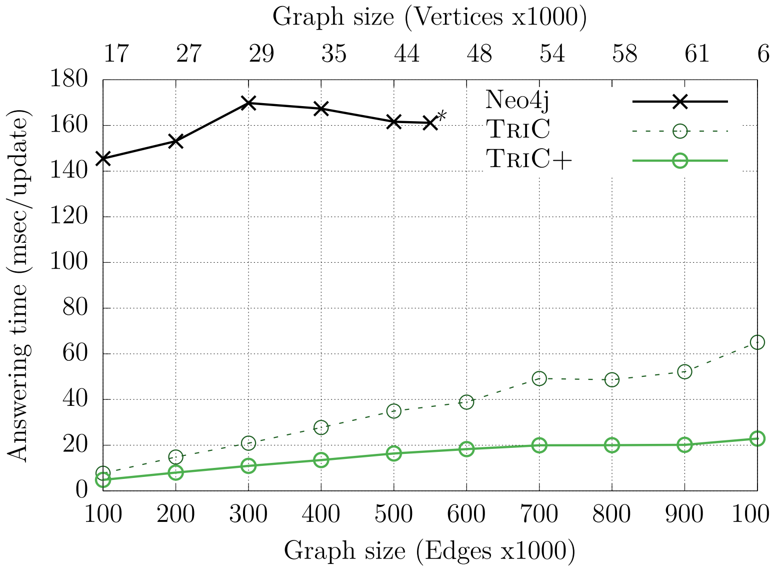
Figure 5.25: Examining the influence of graph size, while comparing the query answering time, when varying the graph size from $|G_E| = 100K$ to $|G_E| = 1M$ edges, for the *NYC* dataset.



Parameter	Value
$\lvert Q_{DB}\rvert$	$5K$
$\lvert G_E\rvert$	$10K - 100K$
ℓ	5
σ	25%
o	35%

Figure 5.26: Examining the influence of graph size, while comparing the query answering time, when varying the graph size from $|G_E| = 10K$ to $|G_E| = 100K$ edges, for the *BioGRID* dataset.

answering performance of the algorithms, when $Q_{DB} = 5K$, $\sigma = 25\%$ for a final graph size of $|G_E| = 100K$ and $|G_V| = 17.2K$. Additionally, we set an execution time threshold of 24 hours due to the high differences in the performance of the algorithms. The *BioGRID* dataset serves as a stress test for our algorithms, since it contains only one type of edge and vertex, thus each incoming update will affect



Parameter	Value
$\lvert Q_{DB}\rvert$	$5K$
$\lvert G_E\rvert$	$100K - 1M$
ℓ	5
σ	25%
o	35%

Figure 5.27: Examining the influence of graph size, while comparing the query answering time, when varying the graph size from $|G_E| = 100K$ to $|G_E| = 1M$ edges, for the *BioGRID* dataset.

(but not necessarily satisfy) the entire query database. To this end, Algorithms INV, INV+, and INC exceed the time threshold and time out at $|G_E| = 50K$, while Algorithm INC+ times out at $|G_E| = 60K$ as denoted by the asterisks in the plot.

Finally, Figure 5.27 presents the results for the *BioGRID* dataset for a final graph size of $|G_E| = 1M$ and $|G_V| = 63K$. We again observe that Algorithms TRIC and TRIC+ achieve the lowest answering time, while Neo4j exceeds the time threshold and times out at $|G_E| = 550K$. As it is demonstrated from the results yielded by the evaluation, Algorithms TRIC and TRIC+ are the most efficient of all; this is attributed to the fact that both algorithms create a combined representation of the query graph patterns that can efficiently be utilized during query answering time.

Comparing Memory Requirements

Table 5.1 presents the memory requirements of each algorithm, for the *SNB*, *NYC* and *BioGRID* datasets when indexing $|Q_{DB}| = 5K$ and a graph of $|G_E| = 100K$. We observe that across all datasets, Algorithms TRIC, INV and INC have the lowest main memory requirements, while, Algorithms TRIC+, INV+, INC+ and Neo4j exhibit higher memory requirements. The higher memory requirements

Algorithm	Dataset		
	SNB	*NYC*	*BioGRID*
TriC	$201MB$	$257MB$	$233MB$
TriC+	$248MB$	$273MB$	$262MB$
Inv	$205MB$	$273MB$	$271MB^{50\text{K}}$
Inv+	$228MB$	$381MB$	$301MB^{50\text{K}}$
Inc	$206MB$	$273MB$	$270MB^{50\text{K}}$
Inc+	$228MB$	$378MB$	$310MB^{60\text{K}}$
Neo4j	$443MB$	$590MB$	$314MB$

Table 5.1: Examining memory usage for $|Q_{DB}| = 5K$, $\ell = 5$, $\sigma = 25\%$, $o = 35\%$ and $|G_E| = 100K$, for the *SNB*, *NYC* and *BioGRID* datasets.

of algorithms that employ a caching strategy, (i.e., Algorithms TriC+, Inv+, and Inc+) is attributed to the fact that all structures calculated during the materialization phase are kept in memory for future usage; this results in higher memory requirements compared to algorithms that do not apply this caching technique (i.e., Algorithms TriC, Inv, and Inc). Finally, Algorithm Neo4j is a full fledged database management system, thus it occupies more memory to support the required specifications.

In this section, we presented the experimental evaluation using three different datasets from social networks, transportation, and biology domains. We compared the performance of Algorithms TriC and TriC+ against the baseline solutions (i.e. Algorithms Inv, Inv+, Inc, Inc+and Neo4j), and demonstrated that our solution can achieve up to orders of magnitude improvements in processing time. In the next section, we present application scenarios from various domains that could benefit the proposed multi-query answering paradigm.

5.6 Applications

Capturing the myriad of emerging patterns, through real-time graph changes, inside graphs could increase the quality of the network content and have applications in a plethora of domains, such as in content discovery and delivery, network monitoring and graph curation. In this section, we discuss additional application scenarios to highlight the importance of supporting continuous query evaluation over graph streams.

5.6.1 Social Networks

Social network graphs emerge naturally from the evolving social interactions and activities of the users. Many applications such as advertising, recommendation systems, and information discovery can benefit from continuous pattern matching. Prompt identification of influential users and active monitoring of content propagation inside the network could increase the effectiveness in those applications. In such scenarios, applications may leverage on sub-graph matching where patterns already observed in social networks can be utilized [133–135].

Moreover, social networks may provide a plethora of user-oriented services by leveraging on continuous subgraph matching. These services could provide users with querying capabilities and thus assist them in information discovery and trending topic exploration. Given that trending topics emerge in real-time and at a rapid pace, viral content could be captured in the form of retweet-trees [136] and message flows [137, 138], which commonly denote high information diffusion and breaking news events. Such applications, could notify users about trending topics that match their interests, or even emergencies such as earthquakes [139]. Finally, real-time monitoring of the social network and timely identification of fraudulent content publication could aid both users and network curators in preventing its spread. The proposed paradigm, could improve the overall content quality of social network platforms, by capturing subgraph patterns related to fake news dissemination [12, 13], user spamming patterns [102, 140, 141], and bot accounts [10].

5.6.2 Knowledge Bases

Knowledge bases have become a major asset behind many products spanning search, analytics, and recommendation. Quite naturally, knowledge graphs evolve over time, with the most common modification being the addition of new entities and facts (corresponding to edges), either because of new relationships that emerge in the real-world, or because of better harvesting methods and richer data repositories that become available. However, relationships and entities may also be subject of deletion, either because the knowledge graph is edited to remove noise (wrong facts) or because of duplicate entries for a single entity.

Given that knowledge graphs are dynamic and often manually curated by human users, monitoring their quality over the period of time becomes essential. In the past, focus has been given on solving the problem of query answering [17, 107, 142] and reasoning [143] over RDF streams and linked data. However, these aforementioned works consider a single query set, thus making these approaches unsuitable for large query sets. A pub/sub system where graph moderators/editors subscribe to and are notified about spurious and/or unusual connections in the knowledge graph, evolution of different structures, patterns, and subgraphs, and trending of news items would be an invaluable tool that would greatly simplify graph moderation.

5.6.3 Protein Interaction Graphs

Protein-protein interaction (PPI) graphs are important data repositories in which proteins are represented as vertices and identified interactions between them are represented as edges. The PPI graphs are typically stored in central repositories such as the BioGRID [131] and The Universal Protein Resource [15] repositories. PPI repositories are constantly updated due to: (i) the addition of new vertices/edges through the identification of new proteins/interactions, (ii) the deletion of edges due to false positives in the interaction identification methods, and (iii) the modification of edge weights through the verification (or invalidation) of already discovered interactions. Scientists are typically forced to manually query these repositories on a regular basis to discover new patterns they are interested in, since the existing tools are unable to capture new patterns in the evolving graphs. Therefore, there is a clear need for an efficient solution that provides the continuous subgraph matching functionality over PPI graphs.

5.6.4 Other Domains

The techniques proposed in this work can also be applied in a wide range of domains such as cybersecurity, road network monitoring, and co-authorship graphs. In cybersecurity, subgraph pattern matching could be applied to monitor the network traffic and capture denial of service and exfiltration attacks [144]. In road network monitoring, subgraph pattern matching could be applied to capture traffic congestion events and taxi route pricing [130]. In the domain of co-authorship graphs, users may utilize the continuous query evaluation algorithms in services similar to Google Scholar Alerts, when requesting to be notified about newly published content, by making use of appropriate graphical user interface tools.

5.7 Outlook and Future Directions

In this chapter, we proposed a new paradigm to efficiently capture the evolving nature of graphs through query graph pattern. We aim at efficiently solving the problem of efficient continuous pattern detection on graph streams while supporting a high number of continuous graph patterns and large graph streams. To this end, we proposed a novel method that indexes and continuously evaluates queries over graph streams, by leveraging on the shared restrictions present in query sets. We evaluated our solution using three different datasets from social networks, transportation and biological interactions domains, and demonstrated that our approach is up to two orders of magnitude faster when compared to typical join-and-explore inverted index solutions and the well-established graph database Neo4j. We plan on extending our methods to support graph deletions and increase expressiveness through query classes that aim at clustering coefficient, shortest path, and betweenness centrality.

6 Realizing Information Filtering: The Case of PING

In this chapter, we present a reference architecture and an operational, open-source information filtering system. Initially, we give the architecture employed by traditional IF systems, and subsequently we present and analyse its distinct operational components. Next, we put forward PING, a fully-functional information filtering system for scientific publications aiming: (i) to showcase the realizability of information filtering and (ii) to explore and test the suitability of the existing technological arsenal for information filtering tasks. The proposed system is entirely based upon open-source tools and components, is customizable enough to be adapted for different textual information filtering tasks, and puts emphasis on user profile expressivity, intuitive UIs, and timely information delivery. To assess the customizability of PING we deployed it in two distinct information filtering scenarios, while to assess its performance we designed and conducted a series of experiments for both scenarios. The results concerning PING have been published in [145].

The rest of the chapter is organized as follows. In Section 6.1, we give the motivation for this work and we elaborate on the importance of creating a fully-functional information filtering system. Section 6.2 presents a textbook example of a typical IF system architecture. Section 6.3 presents the PING system and its supported functionality, and provides details on the different modules of the system. Subsequently, Section 6.4 presents our experimental evaluation for the two different deployment scenarios, the DBLP and DBpedia domains. Finally, Section 6.5 concludes the chapter by discussing the results and provides the outlook and future directions for the Ping system.

6.1 Motivation

In the modern digital era, the creation and availability of new information has increased exponentially. A plethora of information sources, such as news delivery sites, weather reporting services, and digital libraries, constantly make new content available at an overwhelming pace. Tools and technologies developed within the field of information retrieval (IR) have made it possible for users to explore the richness of the content, but have always lacked to provide them with machinery to stay on top of the generated information avalanche. To assist users in coping with the vast amount of newly generated information and the cognitive overload associated with it, the *Information Filtering* (IF) paradigm was introduced. In an IF scenario, users are asked to (implicitly or explicitly) express their information needs through appropriate interfaces, tools and languages and submit *profiles* (or continuous queries) to a system or service. In this way, users create subscriptions that are continuously matched (by the system/service) against newly published content, and generate notifications whenever new content that matches users' information needs is published.

Over the past decades, IF research has mainly focused on providing efficient and effective algorithmic solutions [2, 4, 5, 21–24, 108] that worked well in the controlled research environment, but were never actually used "in the battlefield" as components of a larger IF system. This lack of IF tools that would integrate promising solutions and allow developers to use them for building added-value IF services over textual sources or streams, resulted in the lack of prominent IF systems that would act as demonstrators for the usefulness of the IF paradigm. Thus, currently, the only prominent demonstrator of the potential of IF is Google

Alerts [25], a proprietary closed-source service built upon the Google ecosystem. Although many users nowadays (mis-)use Google Alerts to monitor the web for marketing (e.g., brand mentions), social listening (e.g., comment follow-up), or even citation counting purposes (e.g., in the context of GoogleScholar), there is clearly a need for an extensible, customizable open-source IF system that could be modified to fit domain-specific IF tasks. Such a system, would act in favor of IF research in multiple ways by *(i)* showcasing the usefulness of the IF paradigm to end-users, *(ii)* providing a basis for developers to build added-value IF services in a number of different domains, *(iii)* testing the current technological arsenal in IF, and *(iv)* providing data (that are scarce in the IF domain) regarding system usage or user profiling.

Initially, in the following sections, we present a reference architecture of an IF system, where we present and analyse the components that formulate such a system. Subsequently, we instantiate the presented architecture in a fully-fledged, customizable, open-source IF system, coined PING. PING makes use of state-of-the-art tools and web technologies, while we concentrate on providing an operational system that is designed and implemented on IF-specific requirements. To this end, the presented system is equipped with profile administration (e.g., creation, modification, submission), publication management capabilities (e.g., collection, filtering), different content delivery options (e.g., email or on-site notifications), (interval- or batch-based) monitoring of different types of textual data, and an intuitive user interface. The front-end of the system is built upon modern Internet technologies, while the back-end relies on the well-established Apache Solr[1] platform. PING is designed with flexibility and customizability in mind; developers may use it to easily create textual IF engines for different domains, parameterize and deploy it for IF-specific tasks over their own textual information sources, or use it as a building block for added-value services. To demonstrate the customizability of PING, we deployed and experimented (see Section 6.4) with it in two different textual IF scenarios: the DBLP[2] database for scientific publications and the textual

[1] http://lucene.apache.org/solr/

[2] https://dblp.uni-trier.de/

part of the DBpedia[3] open-sourced knowledge graph. Using Ping, we easily created an IF system that allows users to express their information needs and stay notified for new and interesting publications.

To this end, our contributions may be summarized as follows:

- We present Ping, a novel, fully-functioning IF system build entirely upon open-source components; the proposed system is able to support complex IF tasks in a variety of domains. To the best of our knowledge, this is the first open-source textual IF system that is flexible enough *(i)* to be deployed as a standalone solution on different textual IF tasks and domains or *(ii)* to be used as a building block for other added-value services.
- We showcase the realizability of the developed system on two different domains (textual IF on scientific publications and crowd-sourced encyclopedia articles), and experimentally assess its performance.

More information about the system, along with a fully-functional working deployment over DBLP publications may be found at: `http://195.251.39.222/pingsys`.

In this section, we presented the motivation behind our work and gave a brief-overview of the Ping's functionalities. In the following section, we present a reference architecture for IF systems.

6.2 An Information Filtering System Architecture

In this section, we present the architecture of an information filtering system and analyze the various components that formulate a functional and operating service. More specifically, we present a textbook example of an architecture design employed by traditional IF systems. Figure 6.1 gives the system outline, a visual model, of an IF and its distinct components. The visual model, represents three distinct functionalities of a typical IF system, *user subscriptions indexing*, *publication filtering* and *notification delivery.*

[3] https://wiki.dbpedia.org

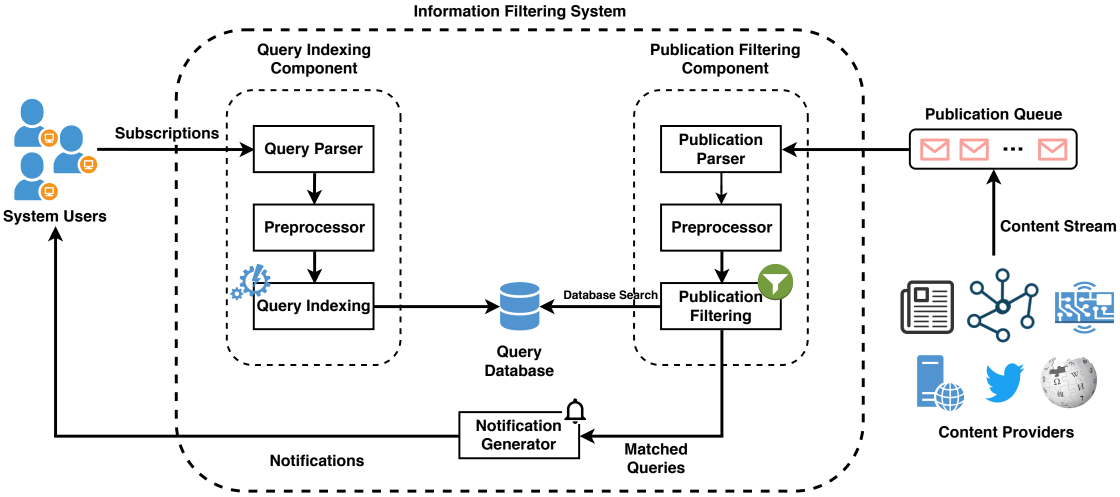


Figure 6.1: Architecture of an Information Filtering System.

User Subscription Indexing. The first functionality presented in Figure 6.1, is the *user subscription process*, which is denoted by the *users* (actors) interacting with the system by posing continuous queries (subscriptions) in it. While, the *query indexing component* is responsible for receiving the user subscriptions and indexing them into the *query database.*

Publication Filtering. The second functionality supported by an IF system (Figure 6.1), is the *publication filtering process*, which is represented by the *content providers* interacting with the system by continuously publishing new content. The IF system is responsible for receiving and filtering those incoming publications against the query database, a functionality implemented by the *publication filtering component.*

Notification Delivery. Finally, the third functionality demonstrated (Figure 6.1), is the *user notification delivery* process (depicted by the publication filtering component), which determines the set of queries that have been satisfied, and the *notification generator* that delivers appropriate notifications to the end users.

In the rest of this section, we give an abstract overview of each component of the system, the functional requirements, actors and relationships presented in Figure 6.1.

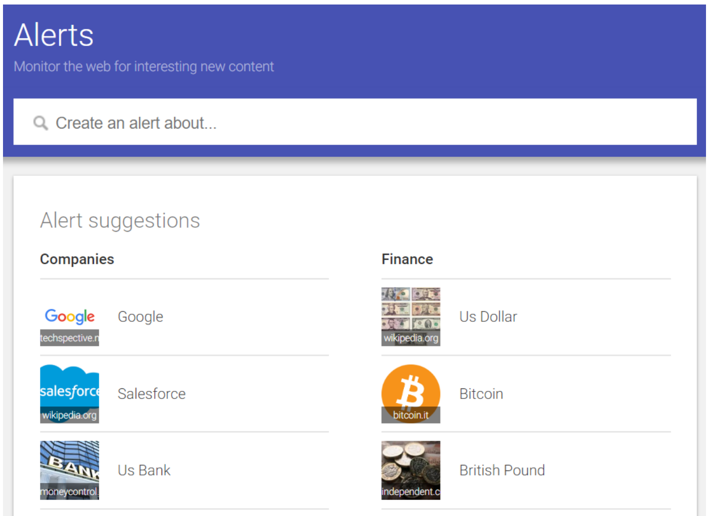


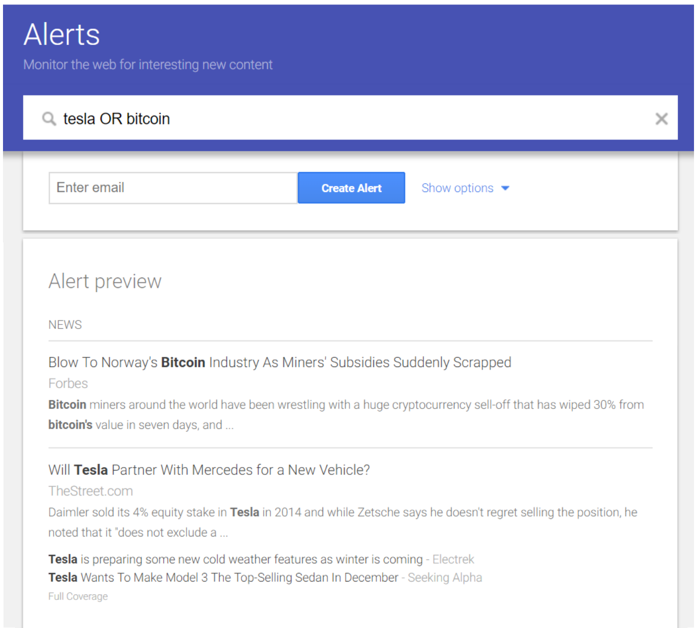
Figure 6.2: Interface for posing long-standing queries, as implemented by the Google Alerts system.

System Users and Subscriptions

In the information filtering paradigm, users can capitalize on the filtering capabilities of the system by posing *long standing queries*, also referred as *subscriptions*. These queries can express long-standing information needs of the users and essentially constitute their personally tailored profiles [38, 49]. The process of formulating these profiles can be achieved by providing users with interfaces and tools similar to the ones present in modern search engines. Figure 6.2 presents an interface from a real-world service, Google Alerts[4], that is designed to assist users in posing long-standing queries. The users can make use of this interface to pose their personalized queries by employing keywords coupled with IR operators, such as *phrase matching*, *wildcard matching* and the *OR operator*. Figure 6.3 demonstrates how a user can formulate a query by making use of the boolean operator OR, as well as a preview of the results that can be (potentially) returned in the form of alerts.

An alternative approach to formulating long-standing queries is services that monitor user behavior and generate profiles on behalf of the users. In such a scenario, services can make use of profiling techniques that automatically determine the user's interests based on several attributes. There is a wide range of research in

[4] `https://www.google.com/alerts`



Alert preview

NEWS

Blow To Norway's **Bitcoin** Industry As Miners' Subsidies Suddenly Scrapped
Forbes
Bitcoin miners around the world have been wrestling with a huge cryptocurrency sell-off that has wiped 30% from **bitcoin's** value in seven days, and ...

Will **Tesla** Partner With Mercedes for a New Vehicle?
TheStreet.com
Daimler sold its 4% equity stake in **Tesla** in 2014 and while Zetsche says he doesn't regret selling the position, he noted that it "does not exclude a ...

Tesla is preparing some new cold weather features as winter is coming - Electrek
Tesla Wants To Make Model 3 The Top-Selling Sedan In December - Seeking Alpha
Full Coverage

Figure 6.3: Interface for posing long-standing queries and notifications preview, as implemented by the Google Alerts system.

recommendation systems that aim at providing personalized information discovery in a wide range of a applications, which include e-commerce, stock monitoring, news delivery etc. Profiling attributes that are commonly employed in such application scenarios can include user activity on the World Wide Web, interactions on social media [146], user feedback both explicit [147–150] and implicit [45, 151–155], repeated searches on search engines, demographic data [156], and shopping patterns.

The Query Indexing Component and the Query Database

The query indexing component is responsible for receiving and handling new incoming subscriptions posted by users or services. The query indexing component, presented in Figure 6.1, is composed by three distinct subcomponents the *query parser*, the *query preprocessor*, and the *query indexing mechanism*. The query parser aims at receiving the incoming subscriptions, parsing their content, and converting plain text into name/value mappings. Subsequently, the query preprocessor makes use of data structures that can capture the query restrictions on keywords, attributes and fields of the incoming query. Thus, the mappings of the query are indexed under data structures that can be directly utilized by the query indexing mechanism. The query indexing mechanism is responsible for organizing and indexing the incoming subscriptions inside the query database, by employing specific algorithmic solutions. Finally, the query database can store the subscriptions on main memory, secondary memory, or employ a hybrid solution where subscriptions reside both in main and secondary memory and are swapped on-demand.

The Content Providers and Publications

In the context of IF systems content providers, also referred as information publishers/producers, denote a set of information sources that publish new content. The process of providing content is a decoupled and asynchronous procedure, where providers constantly push new items into the IF system through direct delivery mechanisms or dissemination services (e.g. RSS feeds). An alternative approach to receiving new content can be achieved by employing IR (also known as information pull) techniques. In such a scenario, the IF system bares the responsibility of collecting the new content by constantly monitoring information sources for changes. Monitoring information sources can be achieved by making use of webpage crawlers, real-time streaming APIs, monitoring tools, and periodic searches on online databases.

The type of content published, by the content providers, can range from a plethora of online application. Such applications can include news websites, social network interactions, sensor networks, IoT enabled devices, knowledge bases, traffic networks, protein-to-protein interaction databases, web databases etc. Given this wide range of applications, an efficient IF system must be designed based on the data model that is going to support in order to be able to provide efficient information delivery services.

The Publication Filtering Component

The publication processing component is responsible for receiving and handling incoming publications and events that arrive at the system. The publication indexing component, presented in Figure 6.1, is composed by three distinct subcomponents the *publication parser*, the *publication preprocessor*, and the *publication filtering* *mechanism.* In such a scenario, the publication parser receives incoming publications and parses their content into name/value mappings. Subsequently, the publication parser hands over the publication into the preprocessor, where only the required fields of the publication are kept, and the publication is stored into specialized data structures that are able to be utilized by the publication filtering mechanism.

The publication filtering component is responsible for receiving an incoming publication and determining if it satisfies the queries indexed inside the query database. The publication filtering component is at the heart of the information filtering system and is designed in order to support the data model under which the information filtering system operates. Information filtering functionality can be achieved through various existing algorithmic and commercial tools , as well as research oriented algorithmic solutions.

Finally, the IF system can additionally employ a message queue, inside which all incoming events arrive and are being stacked for processing. In such a scenario, the component parses the incoming events as they come and keeps the ones that can potentially satisfy the queries indexed under the query database. The formal definition of the information filtering problem follows.

Definition 6.1 *The problem of information filtering is defined as follows: given a database DB of continuous queries that reside on a server and an incoming publication p, retrieve all queries $q \in DB$ that match the incoming p.*

User Notifications

The notification component, presented in Figure 6.1, is responsible for providing users with notifications concerning the publications that match their interests and have been made available in the system. To this end, the notification component receives a list of profiles from the publication filtering mechanism and delivers appropriate notifications, usually coupled with the respective content, to the end users.

In this section, we gave a textbook example of a typical IF system architecture. Each component of the IF system described is dependent on the underlying data model employed to represent, capture, and satisfy information needs. In the following section, we present the Ping system, the functionalities we implemented, as well as it's underlying architecture.

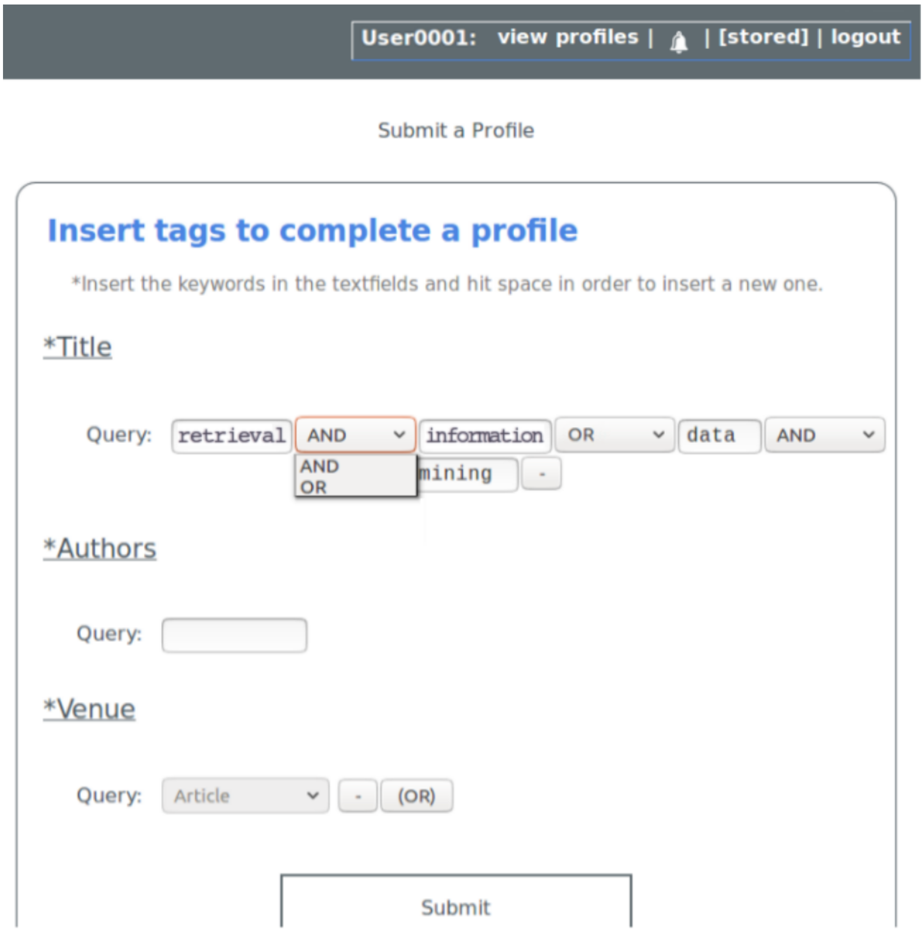
6.3 Ping: A Customizable, Open-Source Information Filtering System for Textual Data

The main idea behind Ping is to enable users to stay updated in a timely fashion, with new and interesting content that satisfies their needs. The Ping system achieves this objective by: *(i)* providing users with profile submission, *(ii)* information filtering, and *(iii)* notification delivery capabilities. In the following, we discuss the underlying functionalities of Ping, overview the system architecture and implementation details, and present how Ping may be deployed over different sources along with the main parameterization options.

6.3.1 Supported Functionality

Profile Submission

Ping enables users to express their information needs using several profiles. Each profile is formed by a set of constraints involving terms (expressed in any attribute of the incoming information) combined with textual operators. Profiles are submitted using a simple and intuitive user interface. For example, in Figure 6.4 a user of the Ping system submits a profile expressing her interest in receiving new content that contain in their *title* the terms “retrieval” and “information” or “data” and “mining”. The user may also specify additional constraints on the *authors* and *venue* attributes. Before submitting the profile to Ping for indexing, the user receives *a projection of the average notifications that will be produced* (per month) based on the constraints presented in the profile. This projection is an *estimate* from existing data and is used to help users assess the generality of their profiles. If needed, the user may refine the profile constraints; when, the profile submission is finalized, Ping indexes the profile in the data store.



User0001: view profiles | | [stored] | logout

Submit a Profile

Insert tags to complete a profile

*Insert the keywords in the textfields and hit space in order to insert a new one.

*Title

Query: retrieval AND information OR data AND mining

*Authors

Query:

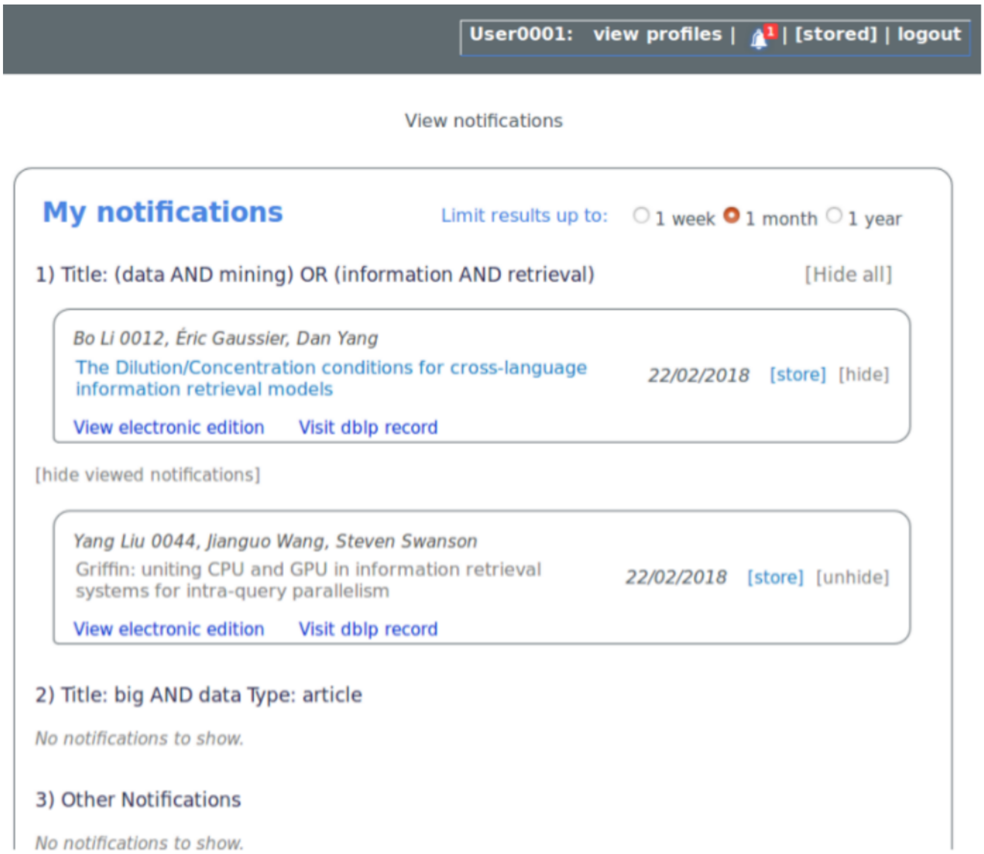
*Venue

Query: Article

Figure 6.4: The profile submission page, as presented to the end-user by the PING system.

Information Filtering

The information filtering process is triggered every time new content becomes available by the monitored repository. In order to locate the newly produced information, PING resides on periodically monitoring the source and retrieving all new published content that becomes available in XML format. When new publications arise, the PING system commences the filtering process. At first, PING retrieves and indexes all new content by making use of the Apache Solr framework. Subsequently, PING retrieves all user profiles from its profile store and prepares them for execution under Solr. Finally, PING executes each profile against the Solr framework that indexes the newly available content, thus determining which profiles match the new publication set. When the information filtering process



User0001: view profiles | 1 | [stored] | logout

View notifications

My notifications

Limit results up to: ○ 1 week ● 1 month ○ 1 year

1) Title: (data AND mining) OR (information AND retrieval) [Hide all]

Bo Li 0012, Éric Gaussier, Dan Yang
The Dilution/Concentration conditions for cross-language information retrieval models
22/02/2018 [store] [hide]
View electronic edition Visit dblp record

[hide viewed notifications]

Yang Liu 0044, Jianguo Wang, Steven Swanson
Griffin: uniting CPU and GPU in information retrieval systems for intra-query parallelism
22/02/2018 [store] [unhide]
View electronic edition Visit dblp record

2) Title: big AND data Type: article

No notifications to show.

3) Other Notifications

No notifications to show.

Figure 6.5: The notification center page, as presented to the end-user by the Ping system.

terminates, Ping sends appropriate notifications to the users.

We chose to implement Ping's information filtering functionality over the Solr framework, which is primarily designed for information retrieval tasks, as there are no publicly available frameworks that natively support information filtering tasks. To this end, Ping implements the information filtering functionality by executing each profile separately against the newly published content that is indexed by the Solr framework. This fact alone highlights the need for and the importance of developing efficient IF-specific machinery to facilitate higher-level IF systems.

Notification Delivery

When a profile is satisfied by an incoming publication, Ping delivers an appropriate notification to the user. This is achieved by the notification center presented in Figure 6.5. In the notification center, users may find notifications about

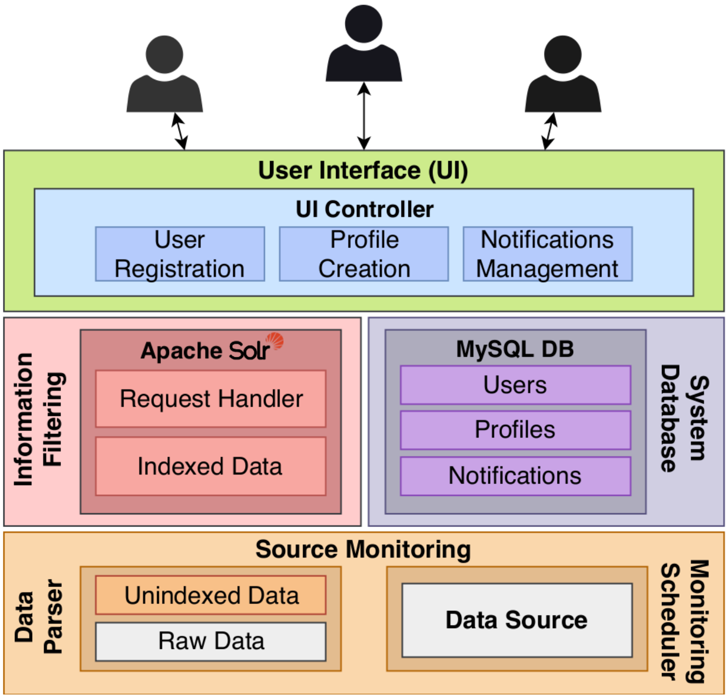


Figure 6.6: The system architecture of PING.

information that matched their profiles and have the ability to evaluate them and refine their profiles. Additionally, the notifications are coupled with direct links to the corresponding information, while the notification center provides notification management capabilities (e.g., storage, dismissal) and advanced visualization capabilities (e.g., notification timeline).

6.3.2 System Architecture

The PING system has been entirely designed on and developed using open-source software; it employs the Linux, Apache, MySQL and PHP (LAMP) stack as the back-end infrastructure, while the front-end modules have been developed using HTML, CSS and JavaScript. The information filtering capabilities of PING are supported by the Apache Solr framework. Figure 6.6 presents a high-level view of the PING architecture and the different modules that comprise the system.

The *Source Monitoring* module is responsible for extracting new available content from information sources; this procedure is implemented through either continuous or periodic (i.e., at predefined intervals) monitoring by means of a wide range of data parsers that are able to accommodate different types of data sources. Thus, the module may either monitor textual streams or extract the newly published information from whole repositories (e.g., provided as monolithic XML files) by parsing the data based on the date attribute and the set interval.

The Apache Solr search framework lies at the heart of the *Information Filtering* module of PING. This module receives all new content and indexes it under the Solr server. Additionally, the module retrieves all user profiles from the database and submits them trough appropriate service calls (using a REST API) to the Solr server for execution. Finally, this module receives the results from the Solr server and generates the appropriate notifications which are then handed over to the *System Database* module.

The *System Database* module is responsible for all the necessary storage and retrieval operations at the system back-end; it stores and manages all user account credentials, user profiles and associated notifications, relevant publications and other user-related data.

The *UI Controller* module is responsible for coordinating the operations of the PING system and enabling a seamless communication between the UI and the underlying architectural elements. Thus, the *UI Controller* serves as a mediator that receives and passes on data to various other modules. This module is also responsible for visualizing all user interactions including *(i)* user registration and account management activities, *(ii)* profile creation, submission and editing, and *(iii)* notification delivery and management.

Deployment over Different Sources

The PING system offers a variety of customizability options for its deployment over different data sources. The system administrator may easily set several information filtering parameters tailored for better monitoring of the data source of choice. Such parameters include *(i)* the type of the monitored data source, *(ii)* the source monitoring rate, *(iii)* the attributes that will correspond to the monitored data, *(iv)* the data manipulation rules (e.g., the employment of tokenization or stemming) for the different data types of interest, and *(v)* restrictions on the generated notifications and preferred delivery method. The deployed online[5] version of Ping (shown in the screenshots of Figures 6.4 and 6.5) is set up to work with scientific publications from DBLP, with the following parameter setup: *(i)* monolithic XML files, *(ii)* a monitoring rate of 24 hours, *(iii)* attributes corresponding to the *title*, *authors*, and *venue type*, *(iv)* tokenization and stemming enabled, and (v) on-site notifications only.

In this section, we presented the functionalities supported by the Ping system, as well as it's underlying architecture. In the following section, we present the experimental evaluation results that we obtained when deploying Ping under the DBpedia and DBLP information domains.

6.4 Experimental Evaluation

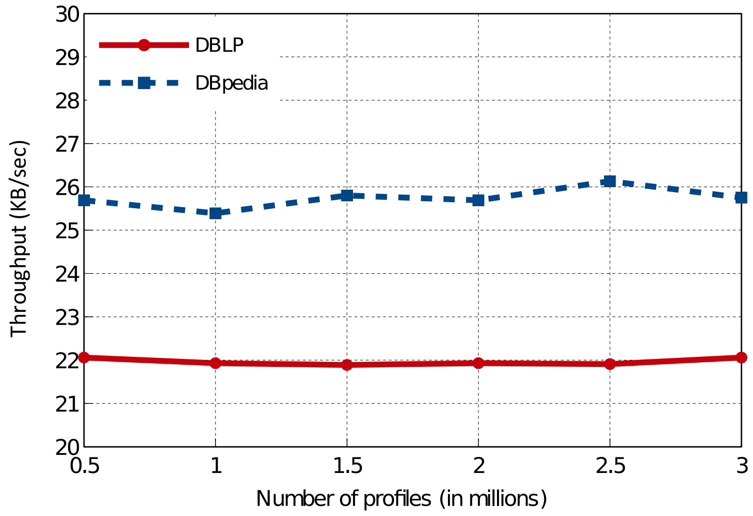
In this section, we present a series of experiments that assess Ping over two distinct deployment scenarios.

6.4.1 Data and Profile Sets

In order to evaluate the filtering performance of Ping and demonstrate its real-world capabilities (and ease of customizability), we designed and experimented with two deployment scenarios. In the first scenario, we deploy Ping over the DBLP database for scientific publications. Thus, we utilize the DBLP corpus that contains all entries published during 2018 and consists of $786K$ publications, with a vocabulary size of $162K$ unique terms. As DBLP is a focused domain, we designed an additional deployment scenario that assesses the performance of Ping under a more general domain. To this end, we deployed and evaluated Ping also over the textual part of DBpedia, which covers a wide range of topics, with a total vocabulary of $3.2M$ unique terms.

Since no databases of profiles for neither scenarios were available to us, we synthetically generated one for each deployment scenario (according to previously

[5] Available at: `http://195.251.39.222/pingsys`



Parameter	Value
P_L	5
I_Q	$500K$
I_D	$50K$
DB	$0.5M - 3M$

Figure 6.7: Examining the influence of the profile database size, when comparing the filtering throughput, for $P_L = 5$.

used methodologies [4, 5, 108]). These two profile databases are formed by conjuncts of different terms; each term conjunct is selected equiprobably among the multi-set of words forming DBLP and DBpedia respectively. Finally, for each profile set we randomly selected $50K$ publications and used them for the filtering task.

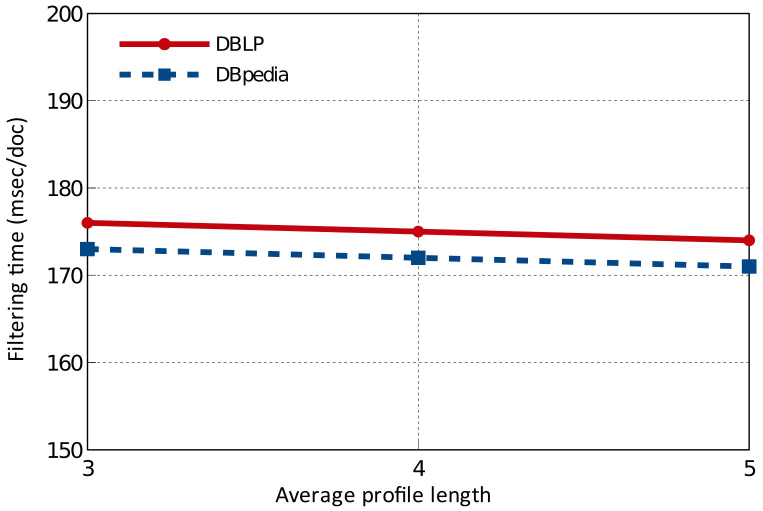
6.4.2 Technical Configuration

A machine with Intel Xeon CPU E5-2650 2.00GHz, 32GB RAM, and Ubuntu Linux 18.04 was used to host the two deployment scenarios of Ping. The Apache Solr server was assigned $2GB$ of main memory. Time measurement report wall-clock time and the results of each experiment is averaged over 10 runs, to eliminate fluctuations in time measurements.

6.4.3 Evaluation Results

Figures 6.7, 6.8 and 6.9 present the most interesting results regarding the deployment and evaluation of Ping under the two deployment scenarios.

Figure 6.7 shows the throughput in KB/sec needed to filter $50K$ incoming publications against a profile database of different sizes for an average number of 5 terms in the profile ($P_L = 5$), under both deployment scenarios. We observe that the throughput of Ping is consistent across the increasing profile databases for



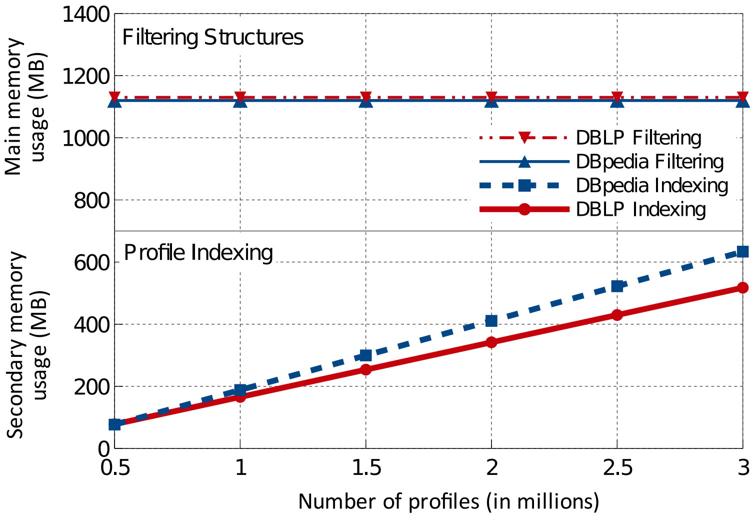
Parameter	Value
P_L	$3-5$
I_Q	–
I_D	$50K$
DB	$3M$

Figure 6.8: Examining the influence of the average profile length, when comparing the filtering time, for a database size of $DB_P = 3M$.

both cases. Moreover, the lower throughput of Ping under the DBLP domain, is attributed to the restricted vocabulary ($126K$ terms) of the topic-specific domain. A restricted profile vocabulary increases the probability to match a user profile against an incoming publication; these matches increase filtering time and hence reduce throughput.

Figure 6.8 shows the time (in milliseconds) that Ping requires to filter an incoming document in a database storing $DB_P = 3M$ profiles, when the average number of terms in the profile P_L varies. We observe that the performance of the system (slightly) improves as the average profile size increases since longer profiles (i.e., profiles with more constraints) exhibit a lower probability to match an incoming publication (high selectivity), and thus require less time to be evaluated by Ping.

Finally, Figure 6.9 presents the (primary and secondary) memory requirements of Ping for different sizes of the profile database. We observe that, in both deployment scenarios Ping exhibits low memory requirements, while transitioning from a topic-specific to a more general domain has minimum impact on memory needs. Ping indexes solely the incoming publication set during filtering time and executes each profile against it, while after the filtering phase completes intermediate results and publications are discarded. This approach allows Ping to exhibit constant memory



Parameter	Value
P_L	5
I_Q	$500K$
I_D	$50K$
DB	$0.5M - 3M$

Figure 6.9: Examining the influence of the profile database size, when comparing memory usage, for $P_L = 5$.

requirements at filtering time, and allows it to efficiently support both topic-specific and general domains with low memory requirements.

The performance evaluation results reported in Figures 6.7, 6.8 and 6.9 suggest that PING may be utilized to efficiently support different textual IF tasks with high throughput and a relatively small memory footprint. The demonstrated efficiency, alongside the inherent profile expressivity delivered from the Solr engine, the different customization options, and the open-source nature of the system, make it a promising solution for complex textual IF tasks in many domains.

6.5 Outlook and Future Directions

In this chapter, we presented PING, a customizable textual IF system built entirely over open source software, and experimentally assessed its performance in two different deployment scenarios. In what follows, our work involves extending the system by implementing a version for multi-core and cluster environments, integrating more formats for source monitoring, and incorporating non-textual IF (e.g., structural or graph constraints). Apart from the source code that will be released upon publication, we also plan to openly provide the research community with any (anonymized) profile dataset that may be constructed by the end-usage of PING, in an effort towards realistic benchmarks for IF research.

7 Conclusions

In this chapter we conclude our work by providing a short summary of the research conducted (Section 7.1), we highlight the contributions of our work (Section 7.2) and finally discuss possible directions for future research (Section 7.3).

7.1 Summary

In this thesis, we studied the problem of multi-query answering under the scope of information filtering where we applied it over three interrelated information domains. The filtering problem is of high importance and to this end, we designed and developed novel algorithms solutions that aim at efficiently capturing the nature of information streams. Finally, our research concluded with the implementation of an information filtering system that aims at can be deployed as a stand-alone solution, as well as, as a building block for other added-value services. In this section, we give a summary of our work.

Efficient Continuous Multi-Query Processing over Textual Data

Initially, we explored efficiency issues under the scope of Boolean text-based IF. Although previous solutions in the literature had focused on providing efficient filtering under various data models and query languages, they were mainly designed under a static query workload assumption. Adopting a static query workload approach may cause a degradation in answering times over time, as the query database is bound to frequent modifications. In our approach, we chose to tackle the performance degradation issues through the proposal of a main-memory algorithmic solution, coined StaR, that employs database reorganization techniques and achieves efficient textual IF under the Boolean model. The key idea of our approach was to utilize trie data structures and capture the common elements of the query set. Capturing common elements of the query set, was previously studied in the literature [3–5]. However, these works focused on minimizing the size of the trie forest, as there was an implicit conjecture that a small forest would result in high filtering performance. To this end, our solution aimed at collecting and utilizing statistics based on the keyword importance, while it employed reorganization of the query database based on those statistic insights. Finally, our findings prove that the forest size is not the domination optimization factor when aiming to achieve high filtering throughput, on the contrary we demonstrate that the nature of the tries and their qualitative characteristics (based on heuristics) are a better optimization factor.

Efficient Continuous Multi-Query Processing over Textual and RDF Data

In the domain of ontology-based IF research, we aimed at addressing the lack of a complete full-text filtering mechanism, beyond existing regular expression and equality support. We proposed a SPARQL extension with full-text operators that was designed to provide expressive continuous queries and address the versatile user needs in applications like digital libraries and news filtering. Aiming to support the proposed extension and providing efficient filtering under ontology-based systems, we developed a family of main-memory Algorithms coined RTF. Our solution can support SPARQL queries with full-text constraints and is able to filter incoming RDF publications in a few milliseconds. We presented indexing methods and solutions that capture and exploit commonalities between continuous queries at indexing time and leverage on the natural proprieties of RDF during the filtering phase. Our family of algorithms, is the first in the literature that is able to support

SPARQL queries with full-text constraints, while we experimentally demonstrated its efficiency against existing state-of-the-art solutions.

Efficient Continuous Multi-Query Processing over Evolving Graph Data

Subsequently, our research focused at capturing patterns of interest that emerge in evolving graphs. Monitoring and finding patterns that emerge on graph streams can have several applications in the domains of social networks, protein-to-protein interaction graphs, network monitoring and knowledge graphs. To this end, we proposed a query subscription language in the form of continues sub-graph queries that can be utilized to express sub-graph constraints and thus capture the emerging patterns over streams of graph updates. In our research, we developed a novel algorithmic solution, coined TriC to index and cluster continuous graph queries. Our solution is the first in the literature that considers thousands of continuous graph queries, contrary to previous approaches that considered only a hand-full of continuous queries. We experimentally demonstrated that by identifying shared patterns among the query workload we can minimize the number of operations necessary to answer continuous sub-graph queries and thus achieve high throughput.

Realizing Information Filtering: The Case of Ping

Concluding our research, we presented a novel full-functioning IF system build entirely upon open-source components. The proposed system, coined Ping, is able to support complex IF tasks in a variety of domains. The presented system is equipped with profile administration (e.g., creation, modification, submission), publication management capabilities (e.g., collection, filtering), different content delivery options (e.g., email or on-site notifications), (interval- or batch-based) monitoring of different types of textual data, and an intuitive user interface. To demonstrate the customizability of Ping, we deployed and experimented with it in two different textual IF scenarios: the DBLP database for scientific publications and the textual part of the DBpedia knowledge graph.

7.2 Contributions

In this section, we present an overview of the contributions of our research. Initially we highlight our four major research contributions, while subsequently we present a detailed discussion and contribution lists for each information domain. To this end, our research makes the following four important contributions:

- It studies and proposes novel algorithmic solutions in the domain of text-based IF, that surpass current state-of-the-art approaches. The research yielded important insights that challenge the status quo with regard to efficiency optimization strategies.

- It builds upon the groundwork conducted in text-based IF systems, and introduces full-text support in ontology-based IF. The research proposes, a full-text extension for the SPARQL query language, while it yields the first multi-query processing algorithmic solution in the literature that supports full-text operators over the RDF data model as a first class citizen.

- It is the first in the literature that breaks ground in multi-query processing over evolving graphs. The research capitalizes on the knowledge gathered from previous research insights and builds novel algorithmic solutions to efficiently address the resource-demanding multi-query processing over graph streams.

- It exploits the important insights and research experience gathered, and employs it in the development of a modern open-source IF solution that can be deployed under multiple information domains.

In our research, we presented an algorithmic solution under the text-based Boolean IF paradigms, where we identified the importance of query insertion order in the construction of the indexing data structures that facilitate the query workload. To this end, our research makes the following improvements:

- It demonstrates that contrary to previous research works, the nature of the constructed tries, rather than their compactness is the key optimization factor for efficient filtering, especially in datasets with rare clustering opportunities.
- It proves, through experimental evaluation, that constructing trie data structures with rare words at the higher level of the tries leads to improve filtering performance.
- It presents different reorganization strategies, while it showcases their effect in filtering efficiency using both real-world and synthetic datasets.
- It provides extensions of the proposed algorithmic solutions under modern multi-core processors.

In the domain of ontology-based and RDF data, we proposed a SPARQL extension with full-text operators that was designed to provide expressive continuous queries and address the versatile user needs in applications like digital libraries and news filtering. Thus, in this context, our research makes the following important advancements:

- It proposes a SPARQL extension that can support full-text operators under the Boolean model, coupled with word proximity and phrase matching capabilities.
- It provides a family of continuous query indexing algorithms that support SPARQL full-text queries and are able to filter incoming publications efficiently.
- It extends a third party algorithm for ontology pub/sub, to offer full-text support and utilizes it under the experimental evaluation scenarios.

Subsequently, we aimed at capturing patterns of interest that emerge in evolving graphs. We motivated our work by identifying and presenting *application scenarios* from various domains, that could benefit from the proposed continuous multi-query answering paradigm. In the context of multi-query processing over evolving graph streams, our research makes the following important advancements:

- It studies and formalizes the problem of continuous multi-query answering over graph streams.
- It presents a novel query graph clustering algorithm that is able to efficiently handle large numbers of continuous graph queries by resorting on (i) the decomposition of continuous query graphs to covering paths and (ii) the utilization of tries for capturing the common parts of those paths.
- Since no prior work in the literature had considered continuous multi-query answering, it presents two algorithmic solutions that utilize inverted indexes for the query answering, as well as we deployed and extended Neo4j to support our proposed paradigm.
- It identifies different variations of the main algorithmic solution and the baseline approaches, which utilized caching strategies, and experimentally evaluates the effect of such solutions on the problem at hand.
- It assesses the performance, of the proposed solutions using three different datasets from social networks, transportation, and biology domains, and compared the performance against the developed baseline solutions. The experimental evaluation demonstrates that the main algorithmic solution can achieve up to two orders of magnitude improvement in query processing time.

Finally, we presented Ping, a novel, fully-functioning IF system build entirely upon open-source components; the proposed system is able to support complex IF tasks in a variety of domains. To the best of our knowledge, this is the first open-source textual IF system is flexible enough that:

- Can be deployed as a standalone solution on different textual IF tasks and domains or to be used as a building block for other added-value services.
- Showcases the realizability of the developed system on two different domains (textual IF on scientific publications and crowd-sourced encyclopedia articles), and experimentally assess its performance.

- It provides an implementation of the information filtering functionality over the Solr framework, which is primarily designed for information retrieval tasks, and highlights the lack of native information filtering tools.

7.3 Open Problems

In this section we discuss of open problems related to our research. This list of problems is directly related to our research findings and serves as a guideline for future potential extensions of our work.

In the domain of textual based IF filtering we aimed at providing efficient algorithms under the Boolean model. An interesting direction would be to extend our algorithmic solution to other data models such as the VSM model. Future research, can also include the adaptation of automata and graph-based techniques as in [24, 50] to facilitate the Boolean IF paradigm while comparing it against our trie-based approaches.

In our research we proposed a SPARQL extension with full-text operators that was designed to provide expressive continuous queries and address the versatile user needs. The proposed SPARQL extension can be extended to support more Boolean operators, as well as to be extended to support VSM queries for text representation in SPARQL. Additionally, an interesting approach would be to adapt our algorithms under modern computer cluster architectures such as Apache Hadoop and Apache Spark.

In the context of multi-query answering over graph streams we motivated the importance of capturing patterns that emerge in evolving graphs. In this thesis we identified and presented application scenarios that could benefit under the proposed continuous sub-graph matching paradigm. To this end, there is a plethora of graph metrics that could be adopted in the proposed continuous multi-query answering paradigm and further enhance the usefulness of the proposed paradigm. These metrics could potentially include clustering coefficient, shortest paths, k-motifs, betweenness centrality and node degree measurements. In the following, we give a brief overview of how these metrics can be utilized under a continuous multi-query answering setup.

In the case of clustering coefficient, users or service can subscribe to emerging subgraphs with average clustering coefficient above a certain threshold C_{th} with at least κ nodes in it. Thus, when C_i is the local clustering coefficient of a node n_i, then notify whenever a subgraph G' with clustering coefficient of at least C_{th} emerges. Continuous queries that specify a *clustering coefficient* (or any similar) measure will be useful to identify communities that form dynamically for targeted advertising, predict/validate PPI interactions, and track trending entities/items in knowledge graphs.

The shortest path between two nodes u and v is defined as the path from node u to node v such that the sum of weights of the constituent edges in the path is minimized. Shortest path continuous queries are especially useful on PPI graphs as they enable biologists to perform functional correlations and structural annotations between (closely located) proteins. In this scenario, biologists want to get notified when the shortest path between two given proteins drops below a provided threshold; tracking shortest paths between nodes can be beneficial for several other continuous queries such as betweenness centrality and Steiner tree computation discussed below.

Continuous queries that are used to subscribe for certain thresholds or top-K style statistics for given *cliques* and *motifs* are particularly useful in PPI graphs for detecting functionally related proteins and protein complexes. In this scenario, a user would like to be notified when a given clique or motif becomes frequent (i.e., its number of instances exceeds a predefined threshold or is within the top-k most frequent patterns) in the PPI graph.

Betweenness centrality is defined as the fraction of shortest paths passing through a node. Intuitively, nodes with higher betweenness centrality in social graphs have higher visibility and injecting promoted content at those nodes would increase the advertising effect. Similarly, betweenness centrality is a key measure for identifying important proteins in a PPI network since proteins that demonstrate high betweenness centrality are more likely to be essential proteins with interesting functional and dynamic properties. Betweenness centrality may be used as an additional constraint in continuous queries once a subgraph of interest (e.g., a subgraph with specific attribute values or above a certain clustering coefficient) is identified.

Even though *node degree* is a simple metric, together with betweenness centrality constitute the two key characteristics for identifying important proteins in PPI graphs, while continuous queries with node degree constraints could be used (in conjunction with other metrics) to notify knowledge graph curators of new/trending entities/items. The node degree is a key measure for identifying important proteins in a PPI network. Proteins that demonstrate high betweenness centrality are more likely to be essential proteins with interesting functional and dynamic properties. In this setting the user wants to get notified whenever a given protein has a node degree above a certain threshold. Additionally tracking nodes with high degree could be a valuable tool in detecting emerging topics in co-authorship graphs (e.g. DBLP), knowledge bases, and social networks.

Trending story/topic detection is an important area where *dense subgraphs* are known to be of benefit [104]. Contrary to [104], where the focus is on identifying the top-*k* densest subgraphs in a social media stream, our focus is on providing *dense subgraph* as a threshold constraint in continuous queries (in conjunction with attribute/structural matching) to allow monitoring of developing stories/users in social graphs or trending items/entities in knowledge graphs.

In knowledge graphs, continuous queries could be used to notify graph curators when a new *Steiner tree* is formed between given entities consisting of specifically labeled edges, in the spirit of [157] but modified for pub/sub in evolving graphs. Subscription to Steiner tree formation between nodes (or to Steiner points) could be used to monitor knowledge graph quality or emergence of interesting links.

Finally, in our research we designed and presented the Ping IF system, a fully-functioning system capable to support complex IF tasks in a variety of information domains. An interesting direction for the development of Ping would be its deployment over multi-core and computer cluster architecture, thus achieving greater scalability, availability, and facilitating more resource-demanding operations.

Furthermore, a promising approach would be incorporating more formats for source monitoring, as well as incorporating non-textual IF (e.g., structural or graph constraints). Providing a plethora of information sources would increase Ping's effectiveness and thus make it a lucrative option for the end-users.

References

[1] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval.* Cambridge University Press, 2008.

[2] Peter M. Fischer and Donald Kossmann. “Batched Processing for Information Filters”. In: *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan.* 2005, pp. 902–913. URL: https://doi.org/10.1109/ICDE.2005.25.

[3] Christos Tryfonopoulos, Manolis Koubarakis, and Yannis Drougas. “Filtering algorithms for information retrieval models with named attributes and proximity operators”. In: *SIGIR 2004: Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Sheffield, UK, July 25-29, 2004.* 2004, pp. 313–320. URL: http://doi.acm.org/10.1145/1008992.1009047.

[4] Christos Tryfonopoulos, Manolis Koubarakis, and Yannis Drougas. “Information filtering and query indexing for an information retrieval model”. In: *ACM Trans. Inf. Syst.* 27.2 (2009), 10:1–10:47. URL: http://doi.acm.org/10.1145/1462198.1462202.

[5] Tak W. Yan and Hector Garcia-Molina. “Index Structures for Selective Dissemination of Information Under the Boolean Model”. In: *ACM Trans. Database Syst.* 19.2 (1994), pp. 332–364. URL: http://doi.acm.org/10.1145/176567.176573.

[6] Milenko Petrovic, Ioana Burcea, and Hans-Arno Jacobsen. “S-ToPSS: Semantic Toronto Publish/Subscribe System”. In: *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany.* 2003, pp. 1101–1104. URL: http://www.vldb.org/conf/2003/papers/S36P06.pdf.

[7] Jinling Wang, Beihong Jin, and Jing Li. “An Ontology-Based Publish/Subscribe System”. In: *Middleware 2004, ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18-20, 2004, Proceedings.* 2004, pp. 232–253. URL: https://doi.org/10.1007/978-3-540-30229-2%5C_13.

[8] Milenko Petrovic, Haifeng Liu, and Hans-Arno Jacobsen. “G-ToPSS: fast filtering of graph-based metadata”. In: *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005.* 2005, pp. 539–547. URL: http://doi.acm.org/10.1145/1060745.1060824.

[9] Myung-Jae Park and Chin-Wan Chung. “iBroker: An Intelligent Broker for Ontology Based Publish/Subscribe Systems”. In: *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China.* 2009, pp. 1255–1258. URL: `https://doi.org/10.1109/ICDE.2009.214`.

[10] Yazan Boshmaf, Ildar Muslukhov, Konstantin Beznosov, and Matei Ripeanu. “The socialbot network: when bots socialize for fame and money”. In: *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011.* 2011, pp. 93–102. URL: `https://doi.org/10.1145/2076732.2076746`.

[11] Alex Hai Wang. “Don’t Follow Me - Spam Detection in Twitter”. In: *SECRYPT 2010 - Proceedings of the International Conference on Security and Cryptography, Athens, Greece, July 26-28, 2010, SECRYPT is part of ICETE - The International Joint Conference on e-Business and Telecommunications.* 2010, pp. 142–151.

[12] Chunyao Song, Tingjian Ge, Cindy X. Chen, and Jie Wang. “Event Pattern Matching over Graph Streams”. In: *PVLDB* 8.4 (2014), pp. 413–424. URL: `http://www.vldb.org/pvldb/vol8/p413-ge.pdf`.

[13] Dan Williams. *Detecting Fake News with Neo4j & KeyLines.* 2017. URL: `https://neo4j.com/blog/detecting-fake-news-neo4j-keylines/`.

[14] Ioannis Xenarios, Lukasz Salwínski, Xiaoqun Joyce Duan, Patrick Higney, Sul-Min Kim, and David Eisenberg. “DIP, the Database of Interacting Proteins: a research tool for studying cellular networks of protein interactions”. In: *Nucleic Acids Research* 30.1 (2002), pp. 303–305. URL: `https://doi.org/10.1093/nar/30.1.303`.

[15] The UniProt Consortium. “UniProt: the universal protein knowledgebase”. In: *Nucleic Acids Research* 45.Database-Issue (2017), pp. D158–D169. URL: `https://doi.org/10.1093/nar/gkw1099`.

[16] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. “C-SPARQL: a Continuous Query Language for RDF Data Streams”. In: *Int. J. Semantic Computing* 4.1 (2010), pp. 3–25. URL: `https://doi.org/10.1142/S1793351X10000936`.

[17] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. “C-SPARQL: a Continuous Query Language for RDF Data Streams”. In: *Int. J. Semantic Computing* 4.1 (2010), pp. 3–25. URL: `https://doi.org/10.1142/S1793351X10000936`.

[18] Dennis E. Shasha, Jason Tsong-Li Wang, and Rosalba Giugno. "Algorithmics and Applications of Tree and Graph Searching". In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. 2002, pp. 39–52. URL: `http://doi.acm.org/10.1145/543613.543620`.

[19] Huahai He and Ambuj K. Singh. "Closure-Tree: An Index Structure for Graph Queries". In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. 2006, p. 38. URL: `https://doi.org/10.1109/ICDE.2006.37`.

[20] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. "Efficient Subgraph Matching on Billion Node Graphs". In: *PVLDB* 5.9 (2012), pp. 788–799. URL: `http://vldb.org/pvldb/vol5/p788%5C_zhaosun%5C_vldb2012.pdf`.

[21] Tak W. Yan and Hector Garcia-Molina. "Index Structures for Information Filtering Under the Vector Space Model". In: *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*. 1994, pp. 337–347. URL: `https://doi.org/10.1109/ICDE.1994.283049`.

[22] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter M. Fischer. "Path sharing and predicate evaluation for high-performance XML filtering". In: *ACM Trans. Database Syst.* 28.4 (2003), pp. 467–516. URL: `http://doi.acm.org/10.1145/958942.958947`.

[23] Mehmet Altinel and Michael J. Franklin. "Efficient Filtering of XML Documents for Selective Dissemination of Information". In: *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. 2000, pp. 53–64. URL: `http://www.vldb.org/conf/2000/P053.pdf`.

[24] Weixiong Rao, Lei Chen, Shudong Chen, and Sasu Tarkoma. "Evaluating continuous top-k queries over document streams". In: *World Wide Web* 17.1 (2014), pp. 59–83. URL: `https://doi.org/10.1007/s11280-012-0191-3`.

[25] Google Inc. *Google Alerts*. Accessed: 2018-12-01. URL: `https://www.google.com/alerts`.

[26] Kevin Chen-Chuan Chang, Hector Garcia-Molina, and Andreas Paepcke. "Predicate Rewriting for Translating Boolean Queries in a Heterogeneous Information System". In: *ACM Trans. Inf. Syst.* 17.1 (1999), pp. 1–39. URL: `https://doi.org/10.1145/297117.297120`.

[27] Reinhard Diestel. "Graph theory. 2005". In: *Grad. Texts in Math* 101 (2005).

[28] Jim Webber. "A programmatic introduction to Neo4j". In: *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*. 2012, pp. 217–218. URL: `https://doi.org/10.1145/2384716.2384777`.

[29] Hans Peter Luhn. “A Business Intelligence System”. In: *IBM Journal of Research and Development* 2.4 (1958), pp. 314–319. URL: `https://doi.org/10.1147/rd.24.0314`.

[30] Julian A Yochum. “A high-speed text scanning algorithm utilizing least frequent trigraphs”. In: *Proceedings of the IEEE International Symposium on New Directions in Computing.* 1985, pp. 114–121.

[31] Timothy A. H. Bell and Alistair Moffat. “The Design of a High Performance Information Filtering System”. In: *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'96, August 18-22, 1996, Zurich, Switzerland (Special Issue of the SIGIR Forum).* 1996, pp. 12–20. URL: `http://doi.acm.org/10.1145/243199.243203`.

[32] Alistair Moffat and Timothy A. H. Bell. “In Situ Generation of Compressed Inverted Files”. In: *JASIS* 46.7 (1995), pp. 537–550. URL: `https://doi.org/10.1002/(SICI)1097-4571(199508)46:7%5C%3C537::AID-ASI7%5C%3E3.0.CO;2-P`.

[33] James P. Callan. “Document Filtering With Inference Networks”. In: *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'96, August 18-22, 1996, Zurich, Switzerland (Special Issue of the SIGIR Forum).* 1996, pp. 262–269. URL: `http://doi.acm.org/10.1145/243199.243273`.

[34] Howard R. Turtle and W. Bruce Croft. “Evaluation of an Inference Network-Based Retrieval Model”. In: *ACM Trans. Inf. Syst.* 9.3 (1991), pp. 187–222. URL: `http://doi.acm.org/10.1145/125187.125188`.

[35] James P. Callan, W. Bruce Croft, and Stephen M. Harding. “The INQUERY Retrieval System”. In: *Proceedings of the International Conference on Database and Expert Systems Applications, Valencia, Spain, 1992.* 1992, pp. 78–83.

[36] Michael J. Franklin and Stanley B. Zdonik. “"Data In Your Face": Push Technology in Perspective”. In: *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.* 1998, pp. 516–519. URL: `http://doi.acm.org/10.1145/276304.276360`.

[37] Mehmet Altinel, Demet Aksoy, Thomas Baby, Michael J. Franklin, William Shapiro, and Stanley B. Zdonik. “DBIS-Toolkit: Adaptable Middleware for Large Scale Data Delivery”. In: *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.* 1999, pp. 544–546. URL: `http://doi.acm.org/10.1145/304182.304571`.

[38] Tak W. Yan and Hector Garcia-Molina. “The SIFT Information Dissemination System”. In: *ACM Trans. Database Syst.* 24.4 (1999), pp. 529–565. URL: `http://doi.acm.org/10.1145/331983.331992`.

[39] Françoise Fabret, H Arno Jacobsen, François Llirbat, Joăo Pereira, Kenneth A Ross, and Dennis Shasha. “Filtering algorithms and implementation for very fast publish/subscribe systems”. In: *ACM Sigmod Record*. Vol. 30. 2. ACM. 2001, pp. 115–126.

[40] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. “Monitoring XML Data on the Web”. In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*. 2001, pp. 437–448. URL: http://doi.acm.org/10.1145/375663.375723.

[41] Alexis Campailla, Sagar Chaki, Edmund Clarke, Somesh Jha, and Helmut Veith. “Efficient filtering in publish-subscribe systems using binary decision diagrams”. In: *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society. 2001, pp. 443–452.

[42] Mohammad Sadoghi and Hans-Arno Jacobsen. “BE-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. 2011, pp. 637–648. URL: http://doi.acm.org/10.1145/1989323.1989390.

[43] Mohammad Sadoghi and Hans-Arno Jacobsen. “Analysis and optimization for boolean expression indexing”. In: *ACM Trans. Database Syst.* 38.2 (2013), 8:1–8:47. URL: http://doi.acm.org/10.1145/2487259.2487260.

[44] Manolis Koubarakis, Christos Tryfonopoulos, Paraskevi Raftopoulou, and T. Koutris. “Data Models and Languages for Agent-Based Textual Information Dissemination”. In: *Cooperative Information Agents VI, 6th International Workshop, CIA 2002, Madrid, Spain, September 18-20, 2002, Proceedings*. 2002, pp. 179–193. URL: https://doi.org/10.1007/3-540-45741-0%5C_16.

[45] Masahiro Morita and Yoichi Shinoda. “Information Filtering Based on User Behaviour Analysis and Best Match Text Retrieval”. In: *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*. 1994, pp. 272–281. URL: http://dl.acm.org/citation.cfm?id=188583.

[46] Ye-In Chang, Jun-Hong Shen, and Tsu-I Chen. “A Data Mining-Based Method for the Incremental Update of Supporting Personalized Information Filtering”. In: *J. Inf. Sci. Eng.* 24.1 (2008), pp. 129–142. URL: http://www.iis.sinica.edu.tw/page/jise/2008/200801_09.html.

[47] David A. Hull, Jan O. Pedersen, and Hinrich Schütze. “Method Combination For Document Filtering”. In: *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR’96, August 18-22, 1996, Zurich, Switzerland*

(Special Issue of the SIGIR Forum). 1996, pp. 279–287. URL: http://doi.acm.org/10.1145/243199.243275.

[48] Yuefeng Li, Xujuan Zhou, Peter Bruza, Yue Xu, and Raymond Y. K. Lau. "A two-stage text mining model for information filtering". In: *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*. 2008, pp. 1023–1032. URL: http://doi.acm.org/10.1145/1458082.1458218.

[49] Peter W. Foltz and Susan T. Dumais. "Personalized Information Delivery: An Analysis of Information Filtering Methods". In: *Commun. ACM* 35.12 (1992), pp. 51–60. URL: http://doi.acm.org/10.1145/138859.138866.

[50] Nikolaos Nanas, Manolis Vavalis, and Anne N. De Roeck. "A network-based model for high-dimensional information filtering". In: *Proceeding of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2010, Geneva, Switzerland, July 19-23, 2010*. 2010, pp. 202–209. URL: http://doi.acm.org/10.1145/1835449.1835485.

[51] James P. Callan. "Learning While Filtering Documents". In: *SIGIR '98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 24-28 1998, Melbourne, Australia*. 1998, pp. 224–231. URL: http://doi.acm.org/10.1145/290941.290998.

[52] Yi Zhang and James P. Callan. "Maximum Likelihood Estimation for Filtering Thresholds". In: *SIGIR 2001: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, September 9-13, 2001, New Orleans, Louisiana, USA*. 2001, pp. 294–302. URL: http://doi.acm.org/10.1145/383952.384012.

[53] Thomas Hofmann. "Collaborative filtering via gaussian probabilistic latent semantic analysis". In: *SIGIR 2003: Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, July 28 - August 1, 2003, Toronto, Canada*. 2003, pp. 259–266. URL: http://doi.acm.org/10.1145/860435.860483.

[54] Raymond Y. K. Lau, Peter D. Bruza, and Dawei Song. "Towards a belief-revision-based adaptive and context-sensitive information retrieval system". In: *ACM Trans. Inf. Syst.* 26.2 (2008), 8:1–8:38. URL: http://doi.acm.org/10.1145/1344411.1344414.

[55] Dionisis Margaris, Costas Vassilakis, and Panagiotis Georgiadis. "Recommendation information diffusion in social networks considering user influence and semantics". In: *Social Netw. Analys. Mining* 6.1 (2016), 108:1–108:22. URL: https://doi.org/10.1007/s13278-016-0416-z.

[56] Dionisis Margaris and Costas Vassilakis. "Exploiting Rating Abstention Intervals for Addressing Concept Drift in Social Network Recommender Systems". In: *Informatics* 5.2 (2018), p. 21. URL: https://doi.org/10.3390/informatics5020021.

[57] Dionisis Margaris, Costas Vassilakis, and Panagiotis Georgiadis. "Query personalization using social network information and collaborative filtering techniques". In: *Future Generation Comp. Syst.* 78 (2018), pp. 440–450. URL: `https://doi.org/10.1016/j.future.2017.03.015`.

[58] Parisa Haghani, Sebastian Michel, and Karl Aberer. "The gist of everything new: personalized top-k processing over web 2.0 streams". In: *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010.* 2010, pp. 489–498. URL: `http://doi.acm.org/10.1145/1871437.1871502`.

[59] Yi-Hung Wu and Arbee L. P. Chen. "Index Structures of User Profiles for Efficient Web Page Filtering Services". In: *Proceedings of the 20th International Conference on Distributed Computing Systems, Taipei, Taiwan, April 10-13, 2000.* 2000, pp. 644–673. URL: `https://doi.org/10.1109/ICDCS.2000.840981`.

[60] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. "Efficient Filtering of XML Documents with XPath Expressions". In: *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002.* 2002, pp. 235–244. URL: `https://doi.org/10.1109/ICDE.2002.994713`.

[61] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. "Processing XML streams with deterministic automata and stream indexes". In: *ACM Trans. Database Syst.* 29.4 (2004), pp. 752–788. URL: `https://doi.org/10.1145/1042046.1042051`.

[62] Shuang Hou and Hans-Arno Jacobsen. "Predicate-based Filtering of XPath Expressions". In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA.* 2006, p. 53. URL: `https://doi.org/10.1109/ICDE.2006.115`.

[63] Feng Peng and Sudarshan S. Chawathe. "XPath Queries on Streaming Data". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003.* 2003, pp. 431–442. URL: `https://doi.org/10.1145/872757.872810`.

[64] Yves Raimond and Guus Schreiber. *RDF 1.1 Primer.* W3C Note. http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/. W3C, 2014.

[65] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF.* W3C Recommendation. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. W3C, 2008.

[66] Marios Meimaris, George Alexiou, Katerina Gkirtzou, George Papastefanatos, and Theodore Dalamagas. "RDF Resource Search and Exploration with LinkZoo". In: *DATA 2015 - Proceedings of 4th International Conference on Data Management Technologies and Applications, Colmar, Alsace, France, 20-22 July, 2015.* 2015, pp. 232–239. URL: `https://doi.org/10.5220/0005499602320239`.

[67] Katerina Gkirtzou, Kostis Karozos, Vasilis Vassalos, and Theodore Dalamagas. “Keywords-To-SPARQL Translation for RDF Data Search and Exploration”. In: *Research and Advanced Technology for Digital Libraries - 19th International Conference on Theory and Practice of Digital Libraries, TPDL 2015, Poznań, Poland, September 14-18, 2015. Proceedings.* 2015, pp. 111–123. URL: https://doi.org/10.1007/978-3-319-24592-8%5C_9.

[68] Katerina Gkirtzou, George Papastefanatos, and Theodore Dalamagas. “RDF Keyword Search based on Keywords-To-SPARQL Translation”. In: *Proceedings of the First International Workshop on Novel Web Search Interfaces and Systems, NWSearch 2015, Melbourne, Australia, October 23, 2015.* 2015, pp. 3–5. URL: https://doi.org/10.1145/2810355.2810357.

[69] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. “Processing XML Streams with Deterministic Automata”. In: *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings.* 2003, pp. 173–189. URL: https://doi.org/10.1007/3-540-36285-1%5C_12.

[70] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. “Efficient filtering of XML documents with XPath expressions”. In: *VLDB J.* 11.4 (2002), pp. 354–379. URL: https://doi.org/10.1007/s00778-002-0077-6.

[71] Jinling Wang, Beihong Jin, Jing Li, and Danhua Shao. “A Semantic-Aware Publish/Subscribe System with RDF Patterns”. In: *28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings.* 2004, pp. 141–14. URL: https://doi.org/10.1109/CMPSAC.2004.1342818.

[72] Lucie Xyleme. “Xyleme: A Dynamic Warehouse for XML Data of the Web”. In: *International Database Engineering & Applications Symposium, IDEAS '01, July 16-18, 2001, Grenoble, France, Proceedings.* 2001, pp. 3–7. URL: https://doi.org/10.1109/IDEAS.2001.938066.

[73] Thomas Neumann and Gerhard Weikum. “RDF-3X: a RISC-style engine for RDF”. In: *PVLDB* 1.1 (2008), pp. 647–659. URL: http://www.vldb.org/pvldb/1/1453927.pdf.

[74] Alexandre Passant and Pablo N. Mendes. “sparqlPuSH: Proactive Notification of Data Updates in RDF Stores Using PubSubHubbub”. In: *Proceedings of the Sixth Workshop on Scripting and Development for the Semantic Web, Crete, Greece, May 31, 2010.* 2010. URL: http://ceur-ws.org/Vol-699/Paper6.pdf.

[75] Srdjan Komazec, Davide Cerri, and Dieter Fensel. "Sparkwave: continuous schema-enhanced pattern matching over RDF data streams". In: *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012.* 2012, pp. 58–68. URL: https://doi.org/10.1145/2335484.2335491.

[76] Maria Golemati, Constantin Halatsis, Costas Vassilakis, Akrivi Katifori, and Giorgos Lepouras. "A Context-Based Adaptive Visualization Environment". In: *10th International Conference on Information Visualisation, IV 2006, 5-7 July 2006, London, UK.* 2006, pp. 62–67. URL: https://doi.org/10.1109/IV.2006.5.

[77] Akrivi Katifori, Constantin Halatsis, George Lepouras, Costas Vassilakis, and Eugenia G. Giannopoulou. "Ontology visualization methods - a survey". In: *ACM Comput. Surv.* 39.4 (2007), p. 10. URL: https://doi.org/10.1145/1287620.1287621.

[78] Akrivi Katifori, Costas Vassilakis, George Lepouras, Ilias Daradimos, and Constantin Halatsis. "Visualizing a temporally-enhanced ontology". In: *Proceedings of the working conference on Advanced visual interfaces, AVI 2006, Venezia, Italy, May 23-26, 2006.* 2006, pp. 488–491. URL: https://doi.org/10.1145/1133265.1133365.

[79] Akrivi Katifori, Costas Vassilakis, George Lepouras, Elena Torou, and Constantin Halatsis. "Visualization Method Effectiveness in Ontology-Based Information Retrieval Tasks Involving Entity Evolution". In: *9th International Workshop on Semantic and Social Media Adaptation and Personalization, SMAP 2014, Corfu, Greece, November 6-7, 2014.* 2014, pp. 14–19. URL: https://doi.org/10.1109/SMAP.2014.24.

[80] Paul-Alexandru Chirita, Stratos Idreos, Manolis Koubarakis, and Wolfgang Nejdl. "Publish/Subscribe for RDF-based P2P Networks". In: *The Semantic Web: Research and Applications, First European Semantic Web Symposium, ESWS 2004, Heraklion, Crete, Greece, May 10-12, 2004, Proceedings.* 2004, pp. 182–197. URL: https://doi.org/10.1007/978-3-540-25956-5%5C_13.

[81] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. "Publish/Subscribe with RDF Data over Large Structured Overlay Networks". In: *Databases, Information Systems, and Peer-to-Peer Computing, International Workshops, DBISP2P 2005/2006, Trondheim, Norway, August 28-29, 2005, Seoul, Korea, September 11, 2006, Revised Selected Papers.* 2005, pp. 135–146. URL: https://doi.org/10.1007/978-3-540-71661-7%5C_12.

[82] Laurent Pellegrino, Fabrice Huet, Françoise Baude, and Amjad Alshabani. "A Distributed Publish/Subscribe System for RDF Data". In: *Data Management in Cloud, Grid and P2P Systems - 6th International Conference, Globe 2013, Prague, Czech Republic, August 28-29, 2013.*

Proceedings. 2013, pp. 39–50. URL: https://doi.org/10.1007/978-3-642-40053-7%5C_4.

[83] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. “RDFS Reasoning and Query Answering on Top of DHTs”. In: *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*. 2008, pp. 499–516. URL: https://doi.org/10.1007/978-3-540-88564-1%5C_32.

[84] Hans-Arno Jacobsen, Alex King Yeung Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. “The PADRES Publish/Subscribe System”. In: *Principles and Applications of Distributed Event-Based Systems*. 2010, pp. 164–205. URL: http://www.igi-global.com/Bookstore/chapter.aspx?titleid=44400.

[85] Guoli Li, Shuang Hou, and Hans-Arno Jacobsen. “Routing of XML and XPath Queries in Data Dissemination Networks”. In: *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), 17-20 June 2008, Beijing, China*. 2008, pp. 627–638. URL: https://doi.org/10.1109/ICDCS.2008.31.

[86] Vinay Setty, Maarten van Steen, Roman Vitenberg, and Spyros Voulgaris. “PolderCast: Fast, Robust, and Scalable Architecture for P2P Topic-Based Pub/Sub”. In: *Middleware 2012 - ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, December 3-7, 2012. Proceedings*. 2012, pp. 271–291. URL: https://doi.org/10.1007/978-3-642-35170-9%5C_14.

[87] Vinay Setty, Roman Vitenberg, Gunnar Kreitz, Guido Urdaneta, and Maarten van Steen. “Cost-Effective Resource Allocation for Deploying Pub/Sub on Cloud”. In: *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*. 2014, pp. 555–566. URL: https://doi.org/10.1109/ICDCS.2014.63.

[88] Vinay Setty, Gunnar Kreitz, Guido Urdaneta, Roman Vitenberg, and Maarten van Steen. “Maximizing the number of satisfied subscribers in pub/sub systems under capacity constraints”. In: *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*. 2014, pp. 2580–2588. URL: https://doi.org/10.1109/INFOCOM.2014.6848205.

[89] Vinay Setty, Gunnar Kreitz, Roman Vitenberg, Maarten van Steen, Guido Urdaneta, and Staffan Gimåker. “The hidden pub/sub of spotify: (industry article)”. In: *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS ’13, Arlington, TX, USA - June 29 - July 03, 2013*. 2013, pp. 231–240. URL: https://doi.org/10.1145/2488222.2488273.

[90] *Reasoning on RDF streams*. http://streamreasoning.org/publications. last accessed, July 2013.

[91] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. “Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 2013, pp. 337–348. URL: `https://doi.org/10.1145/2463676.2465300`.

[92] Xuguang Ren and Junhu Wang. “Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs”. In: *PVLDB* 8.5 (2015), pp. 617–628. URL: `http://www.vldb.org/pvldb/vol8/p617-ren.pdf`.

[93] Xuguang Ren and Junhu Wang. “Multi-Query Optimization for Subgraph Isomorphism Search”. In: *PVLDB* 10.3 (2016), pp. 121–132. URL: `http://www.vldb.org/pvldb/vol10/p121-ren.pdf`.

[94] Changliang Wang and Lei Chen. “Continuous Subgraph Pattern Search over Graph Streams”. In: *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. 2009, pp. 393–404. URL: `https://doi.org/10.1109/ICDE.2009.132`.

[95] Lei Chen and Changliang Wang. “Continuous Subgraph Pattern Search over Certain and Uncertain Graph Streams”. In: *IEEE Trans. Knowl. Data Eng.* 22.8 (2010), pp. 1093–1109. URL: `https://doi.org/10.1109/TKDE.2010.67`.

[96] César Cañas, Eduardo Pacheco, Bettina Kemme, Jörg Kienzle, and Hans-Arno Jacobsen. “GraPS: A Graph Publish/Subscribe Middleware”. In: *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015*. 2015, pp. 1–12. URL: `http://doi.acm.org/10.1145/2814576.2814812`.

[97] Andrei Z. Broder, Shirshanka Das, Marcus Fontoura, Bhaskar Ghosh, Vanja Josifovski, Jayavel Shanmugasundaram, and Sergei Vassilvitskii. “Efficiently evaluating graph constraints in content-based publish/subscribe”. In: *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*. 2011, pp. 497–506. URL: `http://doi.acm.org/10.1145/1963405.1963476`.

[98] Shirui Pan and Xingquan Zhu. “CGStream: continuous correlated graph query for data streams”. In: *21st ACM International Conference on Information and Knowledge Management, CIKM’12, Maui, HI, USA, October 29 - November 02, 2012*. 2012, pp. 1183–1192. URL: `http://doi.acm.org/10.1145/2396761.2398419`.

[99] Shirui Pan and Xingquan Zhu. “Continuous top-k query for graph streams”. In: *21st ACM International Conference on Information and Knowledge Management, CIKM’12, Maui, HI, USA, October 29 - November 02, 2012*. 2012, pp. 2659–2662. URL: `http://doi.acm.org/10.1145/2396761.2398717`.

[100] Jun Gao, Chang Zhou, and Jeffrey Xu Yu. “Toward continuous pattern detection over evolving large graph with snapshot isolation”. In: *VLDB J.* 25.2 (2016), pp. 269–290. URL: https://doi.org/10.1007/s00778-015-0416-z.

[101] S. Choudhury, L.B. Holder, A. Ray, G. Chin Jr., and J. Feo. “Continuous Queries for Multi-Relational Graphs”. In: *CoRR* (2012).

[102] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. “A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs”. In: *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.* 2015, pp. 157–168. URL: https://doi.org/10.5441/002/edbt.2015.15.

[103] Bin Shao, Haixun Wang, and Yatao Li. “Trinity: a distributed graph engine on a memory cloud”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* 2013, pp. 505–516. URL: http://doi.acm.org/10.1145/2463676.2467799.

[104] Albert Angel, Nick Koudas, Nikos Sarkas, and Divesh Srivastava. “Dense Subgraph Maintenance under Streaming Edge Weight Updates for Real-time Story Identification”. In: *PVLDB* 5.6 (2012), pp. 574–585. URL: http://vldb.org/pvldb/vol5/p574%5C_albertangel%5C_vldb2012.pdf.

[105] Chayant Tantipathananandh, Tanya Y. Berger-Wolf, and David Kempe. “A framework for community identification in dynamic social networks”. In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007.* 2007, pp. 717–726. URL: http://doi.acm.org/10.1145/1281192.1281269.

[106] Charu C. Aggarwal and Karthik Subbian. “Evolutionary Network Analysis: A Survey”. In: *ACM Comput. Surv.* 47.1 (2014), 10:1–10:36. URL: http://doi.acm.org/10.1145/2601412.

[107] Syed Gillani, Gauthier Picard, and Frédérique Laforest. “Continuous graph pattern matching over knowledge graph streams”. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS ’16, Irvine, CA, USA, June 20 - 24, 2016.* 2016, pp. 214–225. URL: http://doi.acm.org/10.1145/2933267.2933306.

[108] Lefteris Zervakis, Christos Tryfonopoulos, Spiros Skiadopoulos, and Manolis Koubarakis. “Query Reorganization Algorithms for Efficient Boolean Information Filtering”. In: *IEEE Trans. Knowl. Data Eng.* 29.2 (2017), pp. 418–432. URL: https://doi.org/10.1109/TKDE.2016.2620140.

[109] Google Inc. *Google Advanced Search.* Accessed: 2018-12-01. URL: https://www.google.com/advanced_search.

[110] Oracle Corporation. *Oracle Text.* Accessed: 2018-12-01. URL: `https://www.oracle.com/technetwork/database/enterprise-edition/index-098492.html`.

[111] Apache Software Foundation. *Apache Lucene.* Accessed: 2018-12-01. URL: `http://lucene.apache.org/`.

[112] Apache Software Foundation. *Apache Solr.* Accessed: 2018-12-01. URL: `http://lucene.apache.org/solr/`.

[113] Manolis Koubarakis, Spiros Skiadopoulos, and Christos Tryfonopoulos. "Logic and Computational Complexity for Boolean Information Retrieval". In: *IEEE Trans. Knowl. Data Eng.* 18.12 (2006), pp. 1659–1666. URL: `https://doi.org/10.1109/TKDE.2006.193`.

[114] Ben Carterette, Virgiliu Pavlu, Hui Fang, and Evangelos Kanoulas. "Million Query Track 2009 Overview". In: *Proceedings of The Eighteenth Text REtrieval Conference, TREC 2009, Gaithersburg, Maryland, USA, November 17-20, 2009.* 2009. URL: `http://trec.nist.gov/pubs/trec18/papers/MQ09OVERVIEW.pdf`.

[115] Gerard Salton, Edward A. Fox, and Harry Wu. "Extended Boolean Information Retrieval". In: *Commun. ACM* 26.11 (1983), pp. 1022–1036. URL: `https://doi.org/10.1145/182.358466`.

[116] Lefteris Zervakis, Christos Tryfonopoulos, Spiros Skiadopoulos, and Manolis Koubarakis. "Full-Text Support for Publish/Subscribe Ontology Systems". In: *The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Proceedings.* 2016, pp. 233–249. URL: `https://doi.org/10.1007/978-3-319-34129-3%5C_15`.

[117] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. "Navigation- vs. Index-Based XML Multi-Query Processing". In: *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India.* 2003, pp. 139–150. URL: `https://doi.org/10.1109/ICDE.2003.1260788`.

[118] Ashish Kumar Gupta and Dan Suciu. "Stream Processing of XPath Queries with Predicates". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003.* 2003, pp. 419–430. URL: `https://doi.org/10.1145/872757.872809`.

[119] Pat Case, Michael Dyck, Mary Holstege, Sihem Amer-Yahia, Chavdar Botev, Stephen Buxton, Jochen Doerre, Jim Melton, Michael Rys, and Jayavel Shanmugasundaram. *XQuery and XPath Full Text.* 2011. URL: `http://www.w3.org/TR/xpath-full-text-10/`.

[120] Fabien Gandon and Guus Schreiber. *RDF 1.1 XML Syntax.* W3C Recommendation. http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/. W3C, 2014.

[121] Lefteris Zervakis, Christos Tryfonopoulos, Vinay Setty, Stephan Seufert, and Spiros Skiadopoulos. “Towards Publish/Subscribe Functionality on Graphs”. In: *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016.* 2016. URL: `http://ceur-ws.org/Vol-1558/paper13.pdf`.

[122] Facebook Inc. *Facebook Reports Fourth Quarter and Full Year 2018 Results.* 2018. URL: `https://investor.fb.com/investor-news/press-release-details/2019/Facebook-Reports-Fourth-Quarter-and-Full-Year-2018-Results/default.aspx`.

[123] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. “The LDBC Social Network Benchmark: Interactive Workload”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* 2015, pp. 619–630. URL: `https://doi.org/10.1145/2723372.2742786`.

[124] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. URL: `http://mitpress.mit.edu/books/introduction-algorithms`.

[125] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2008.

[126] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. “Maintaining Views Incrementally”. In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993.* 1993, pp. 157–166. URL: `https://doi.org/10.1145/170035.170066`.

[127] Neo4j Inc. *Cypher Query Language.* 2019. URL: `https://neo4j.com/developer/cypher/`.

[128] Neo4j Inc. *Parameters.* 2019. URL: `https://neo4j.com/docs/cypher-manual/current/syntax/parameters/`.

[129] Neo4j Inc. *Transactions.* 2019. URL: `https://neo4j.com/docs/cypher-manual/current/introduction/transactions/`.

[130] Chris Whong. *FOILing NYC’s Taxi Trip Data.* 2014. URL: `https://chriswhong.com/open-data/foil_nyc_taxi/`.

[131] Chris Stark, Bobby-Joe Breitkreutz, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, and Mike Tyers. “BioGRID: a general repository for interaction datasets”. In: *Nucleic Acids Research* 34.Database-Issue (2006), pp. 535–539. URL: `https://doi.org/10.1093/nar/gkj109`.

[132] Jayanta Mondal and Amol Deshpande. “CASQD: continuous detection of activity-based subgraph pattern queries on dynamic graphs”. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS ’16, Irvine, CA, USA, June 20 - 24, 2016.* 2016, pp. 226–237. URL: `https://doi.org/10.1145/2933267.2933316`.

[133] Jure Leskovec, Ajit Singh, and Jon M. Kleinberg. “Patterns of Influence in a Recommendation Network”. In: *Advances in Knowledge Discovery and Data Mining, 10th Pacific-Asia Conference, PAKDD 2006, Singapore, April 9-12, 2006, Proceedings.* 2006, pp. 380–389. URL: `https://doi.org/10.1007/11731139%5C_44`.

[134] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. “The dynamics of viral marketing”. In: *TWEB* 1.1 (2007), p. 5. URL: `https://doi.org/10.1145/1232722.1232727`.

[135] Chengxi Zang, Peng Cui, Chaoming Song, Christos Faloutsos, and Wenwu Zhu. “Quantifying Structural Patterns of Information Cascades”. In: *Proceedings of the 26th International Conference on World Wide Web Companion, Perth, Australia, April 3-7, 2017.* 2017, pp. 867–868. URL: `https://doi.org/10.1145/3041021.3054214`.

[136] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. “What is Twitter, a social network or a news media?” In: *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010.* 2010, pp. 591–600. URL: `https://doi.org/10.1145/1772690.1772751`.

[137] Shaozhi Ye and Shyhtsun Felix Wu. “Measuring Message Propagation and Social Influence on Twitter.com”. In: *Social Informatics - Second International Conference, SocInfo 2010, Laxenburg, Austria, October 27-29, 2010. Proceedings.* 2010, pp. 216–231. URL: `https://doi.org/10.1007/978-3-642-16567-2%5C_16`.

[138] Shaozhi Ye and Shyhtsun Felix Wu. “Measuring message propagation and social influence on Twitter.com”. In: *IJCNDS* 11.1 (2013), pp. 59–76. URL: `https://doi.org/10.1504/IJCNDS.2013.054835`.

[139] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. “Earthquake shakes Twitter users: real-time event detection by social sensors”. In: *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010.* 2010, pp. 851–860. URL: `https://doi.org/10.1145/1772690.1772777`.

[140] Leman Akoglu, Mary McGlohon, and Christos Faloutsos. “oddball: Spotting Anomalies in Weighted Graphs”. In: *Advances in Knowledge Discovery and Data Mining, 14th Pacific-Asia Conference, PAKDD 2010, Hyderabad, India, June 21-24, 2010. Proceedings. Part II.* 2010, pp. 410–421. URL: `https://doi.org/10.1007/978-3-642-13672-6%5C_40`.

[141] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and P. Krishna Gummadi. “Measuring User Influence in Twitter: The Million Follower Fallacy”. In: *Proceedings of the Fourth International Conference on Weblogs and Social Media, ICWSM 2010, Washington, DC, USA, May 23-26, 2010*. 2010. URL: `http://www.aaai.org/ocs/index.php/ICWSM/ICWSM10/paper/view/1538`.

[142] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. “A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data”. In: *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*. 2011, pp. 370–388. URL: `https://doi.org/10.1007/978-3-642-25073-6%5C_24`.

[143] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. “Incremental Reasoning on Streams and Rich Background Knowledge”. In: *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part I*. 2010, pp. 1–15. URL: `https://doi.org/10.1007/978-3-642-13486-9%5C_1`.

[144] Cliff Joslyn, Sutanay Choudhury, David Haglin, Bill Howe, Bill Nickless, and Bryan Olsen. “Massive scale cyber traffic analysis: a driver for graph database research”. In: *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*. 2013, p. 3. URL: `http://event.cwi.nl/grades2013/03-Joslyn.pdf`.

[145] Thanasis Chantzios, Lefteris Zervakis, Spiros Skiadopoulos, and Christos Tryfonopoulos. “Demo: Ping - A customizable, open-source information filtering system for textual data”. In: *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019*. 2019.

[146] Xujuan Zhou, Yue Xu, Yuefeng Li, Audun Jøsang, and Clive Cox. “The state-of-the-art in personalized recommender systems for social networking”. In: *Artif. Intell. Rev.* 37.2 (2012), pp. 119–132. URL: `https://doi.org/10.1007/s10462-011-9222-1`.

[147] Gene Golovchinsky, Morgan N. Price, and Bill N. Schilit. “From Reading to Retrieval: Freeform Ink Annotations as Queries”. In: *SIGIR ’99: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 15-19, 1999, Berkeley, CA, USA*. 1999, pp. 19–25. URL: `https://doi.org/10.1145/312624.312637`.

[148] Jay Budzik and Kristian Hammond. “Watson: Anticipating and contextualizing information needs”. In: *In 62nd Annual Meeting of the American Society for Information Science*. Citeseer. 1999.

[149] Stuart E. Middleton, Nigel Shadbolt, and David De Roure. "Ontological user profiling in recommender systems". In: *ACM Trans. Inf. Syst.* 22.1 (2004), pp. 54–88. URL: `https://doi.org/10.1145/963770.963773`.

[150] Silvia N. Schiaffino and Analía Amandi. "Intelligent User Profiling". In: *Artificial Intelligence: An International Perspective.* 2009, pp. 193–216. URL: `https://doi.org/10.1007/978-3-642-03226-4%5C_11`.

[151] Jon M. Kleinberg. "Authoritative Sources in a Hyperlinked Environment". In: *J. ACM* 46.5 (1999), pp. 604–632. URL: `http://doi.acm.org/10.1145/324133.324140`.

[152] Ryen White, Ian Ruthven, and Joemon M. Jose. "Finding relevant documents using top ranking sentences: an evaluation of two alternative schemes". In: *SIGIR 2002: Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 11-15, 2002, Tampere, Finland.* 2002, pp. 57–64. URL: `https://doi.org/10.1145/564376.564389`.

[153] Rachael Rafter and Barry Smyth. "Passive profiling from server logs in an online recruitment environment". In: *Workshop on Intelligent Techniques for Web Personalization at the the 17th International Joint Conference on Artificial Intelligence, Seattle, Washington, USA, August, 2001.* 2001.

[154] Mark Claypool, Phong Le, Makoto Waseda, and David Brown. "Implicit interest indicators". In: *Proceedings of the 6th International Conference on Intelligent User Interfaces, IUI 2001, Santa Fe, NM, USA, January 14-17, 2001.* 2001, pp. 33–40. URL: `https://doi.org/10.1145/359784.359836`.

[155] Diane Kelly and Jaime Teevan. "Implicit feedback for inferring user preference: a bibliography". In: *SIGIR Forum* 37.2 (2003), pp. 18–28. URL: `https://doi.org/10.1145/959258.959260`.

[156] Bruce Krulwich. "LIFESTYLE FINDER: Intelligent User Profiling Using Large-Scale Demographic Data". In: *AI Magazine* 18.2 (1997), pp. 37–45. URL: `http://www.aaai.org/ojs/index.php/aimagazine/article/view/1292`.

[157] Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M. Suchanek, and Gerhard Weikum. "STAR: Steiner-Tree Approximation in Relationship Graphs". In: *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China.* 2009, pp. 868–879. URL: `https://doi.org/10.1109/ICDE.2009.64`.