Full-text Support for Publish/Subscribe Ontology Systems

Full-text Support for Publish/Subscribe Ontology Systems

Lefteris Zervakis

Master Thesis, Department of Informatics and Telecommunications

University of the Peloponnese, May 2014

Full-text Support for Publish/Subscribe Ontology Systems

Lefteris Zervakis

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master in Computer Science and Technology

Committee

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

University of Peloponnese
2014

Abstract

We envision a publish/subscribe ontology system that is scalable to millions of user profiles and filters ontology data in a streaming fashion. In such a system, users submit their subscriptions to the system which indexes them using scalable data structures and matches them with streaming RDF data, to notify the interested subscribers. In this work, we initially propose a SPARQL extension appropriate for a publish/subscribe setting; our extension builds on the natural semantic graph matching of the language and supports the creation of full-text subscriptions. Leveraging on the natural properties of our structures we have developed a family of filtering algorithms which perform both structural and full-text matching at low complexity and minimal filtering time. Thus, when ontology data are published matching subscriptions are efficiently identified and notifications are forwarded to users. Our solution is faster than state-of-the-art competitors, managing to achieve improvement of more than 98% in filtering performance, when tested with real-world datasets.

Περίληψη

Οραματιζόμαστε ένα σύστημα διάχυσης πληροφορίας για δεδομένα οντολογίας που είναι αποδοτικό για εκατομμύρια προφίλ χρηστών και έχει την δυνατότητα να φιλτράρει δεδομένα από οντολογίες που εισέρχονται με συνεχόμενη ροή. Σε ένα τέτοιο σύστημα οι χρήστες εγγράφονται δημιουργώντας προφίλ, τα οποία θα ευρετηριάζονται χρησιμοποιώντας αποδοτικές δομές δεδομένων και θα ταιριάζονται με μια ροή RDF δεδομένων, για να αποκαλύψουν εκείνα τα προφίλ τα οποία ταιριάζουν με τη ροή δεδομένων, και να ειδοποιηθούν οι χρήστες τους. Στην παρούσα εργασία, αρχικά προτείνουμε μια επέκταση της γλώσσας ερωτήσεων SPARQL προσαρμοσμένη στις ανάγκες των συστημάτων διάχυσης πληροφορίας. Η προτεινόμενη επέκταση βασίζεται στους υπάρχοντες εννοιολογικούς ορισμούς της γλώσσας και υποστηρίζει την δημιουργία προφίλ με όρους κειμένων, κατόπιν παρουσιάζουμε μια οικογένεια αλγορίθμων φιλτραρίσματος που χρησιμοποιούν ειδικά σχεδιασμένες δομές και είναι ικανοί να φιλτράρουν τόσο συντακτικά όσο και σε επίπεδο όρων κειμένου τις δημοσιεύσεις με μικρή πολυπλοκότητα και ελάχιστο χρόνο φιλτραρίσματος. Έτσι, όταν τα δεδομένα οντολογίας δημοσιεύονται, φιλτράρονται με τα προφίλ που βρίσκονται στο ευρετήριο και οι ειδοποιήσεις αποστέλνονται στους χρήστες των οποίων τα προφίλ ταίριαξαν. Η λύση που προτείνουμε είναι ταχύτερη από αντίστοιχες στην βιβλιογραφία καθώς επιτυγχάνει κατά 98% ταχύτερο φιλτράρισμα, όπως διαπιστώθηκε σε πειράματα με πραγματικά δεδομένα.

Acknowledgements

Writing a thesis needs assistance and supervision. Sincere thanks to my supervisor, Assist. Prof. *Christos Tryfonopoulos* for his belief in me, his patience and his unending support throughout the years of my master. His enthusiasm and desire for perfection was an inspiration for me. Special thanks go to *Paraskevi Raftopoulou* for her support and positive spirit that brought into the group.

I would also like to thank the members of my committee, Assoc. Prof. *Spiros Skiadopoulos* and Assoc. Prof. *Costas Vassilakis* for their advices, comments and suggestions on improvements and extensions of this work.

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

I would also like to thank the *Software and Database Systems Lab* at *University of Peloponnese* for the resources supplied in order to complete my research. Through the master I received financial support from *Athena-Research and Innovation Center in Information.* I would like to thank all the people in the institution for these research grants. Finally I received additional financial support from the project *TELEIOS-Virtual Observatory Infrastructure for Earth Observation Data.* I would also like to thank all the people in the project and especially Assoc. Prof. *Manolis Koubarakis* for these research grants.

Contents

List of Tables

List of Figures

List of Abbreviations

APS	Access Point Structure
ASL	Adaptive Space Linearisation
BS	Biased Sampling
CAN	Context Addressable Network
DHT	Distributed Hash Table
DL	Digital Library
GUID	Globally Unique Identifier
GW	Gradient Walk
HTTP	HyperText Transfer Protocol
IF	Information Filtering
IP	Internet Protocol
IR	Information Retrieval
LSI	Latent Semantic Indexing
P2P	Peer-to-Peer
RDF	Resource Description Framework
RDF(S)	Resource Description Framework Schema

RT	Routing Table
RS	Random Sampling
RW	Random Walk
SDI	Selective Dissemination of Information
SL	Symmetric Links
SON	Semantic Overlay Network
SRT	Semantic Routing Table
SSW	Semantic Small World
TTL	Time-To-Live
URL	Uniform Resource Locators
VSM	Vector Space Model
XML	eXtensible Markup Language

Chapter 1

Introduction

This thesis addresses the problem of delivering full-text support in ontology based publish/subscribe systems. In this chapter, we define the problem, highlight our approach and present our contributions.

1.1 Problem statement

In the last decade we witness the exponential growth of the internet and consecutively the high availability of newly published information on a daily base. This information flow is intended to reach the users and satisfy their needs on updating their knowledge in their field of interest. Users currently handle the information flow by resorting to periodical searches of newly published information and by trying to distinguish this information to relevant or irrelevant. Consecutively these tasks consume a high percentage of users' time reducing the total time dedicated to the process of information assimilation. The never-ending process of discovering and classifying new data has become tedious for the users, and this cognitive avalanche of information has created the need for applications that can assist the users in the information seeking task.

The need for applications that can assist users to information delivery has created publish/subscribe systems focused on news delivery. These publish/subscribe systems can be used in news alerts, digital libraries or RSS feeds. Users may express their needs to a server by creating a profile describing these needs, a task that may be achieved by utilizing modern subscription languages. Thereafter the alerting systems will be able to notify the subscribed users automatically when they determine the relevance of a new publication to a user's profile. Publications can be generated from news feeds, digital libraries, or even other users who post new items to blogs, social media or other Internet communities. Based on the above, the problem of information filtering examined in this thesis may be defined as follows: given a database DB of continues queries (profiles) that reside on a server and a publication p, retrieve all queries $q \in DB$ that match p.

1.2 Solution outline

The Resource Description Framework (RDF) [68] constitutes a conceptual model and a formal language for representing resources in the Semantic Web. RDF is ideal for representing the publications generated in a publish/subscribe system that focuses on information delivery and can combine a wide variety of sources. Moreover the SPARQL query language [62] is the current recommendation of W3C for querying the Semantic Web. The SPARQL query language can give the tools and flexibility to users to define their interests.

Research has been conducted in the field of P2P publish/subscribe systems where the publications and subscriptions are expressed in RDF forms [18, 43], however there is lack of research on information delivery systems where the load is directed to a single server and the information flow is constant. Furthermore, there is lack of full-text operators in the SPARQL query language, while research has been conducted in the Information Retrieval field in order to support such operations [5, 46, 54–56]. The same work can not be applied in Information Filtering where the principles are fundamentally different as well as the objectives of the system. The need of a publish/subscribe system that can support expressive RDF subscriptions with full-text operators is essential and has to be addressed.

In this work we concentrate on delivering an ontology publish/subscribe system that can index millions of user profiles. The proposed algorithm coined RTF (acronym for RDF Text Filtering), indexes subscriptions defined in the SPARQL query language. In order to provide users with better expressivity we extend the SPARQL query language with full-text operators. The utilization of full-text subscriptions enhances the flexibility and expressiveness of the subscription language. Furthermore the full-text extension can deliver publications to the users with higher precision aiming also at content, and not only structure of publications their field of interest. We support publications from ontology data expressed in native RDF form. The publications data can be extended to a great amount of sources (i.e., XML data). We concentrate on structural and textual Information Filtering of RDF data and present a novel trie-based main-memory algorithm that is able to match incoming publications against millions of profiles in a few milliseconds.

1.3 Contributions

A major part of the works conducted in the field of publish/subscribe systems supporting RDF data and focuses on the semantic matching of profiles [57, 60, 61, 74]. In this work, we concentrate on solving both the textual and structural matching process providing the user with an expressive query language that is a SPARQL extension, and covers both structure and content. We propose methods that leverage on the natural properties of the RDF data and develop structures that assist the indexing and filtering procedures both on structural and textual level which has been greatly overlooked. Thus we concentrate on solving the filtering problem efficiently both for the structural and textual part. Semantic matching can be a complementary process to our developed algorithms without reducing their performance. In the light of the above, our contributions are:

- We *extend* existing publish/subscribe solutions [57] from text equality to full-text matching including keywords and boolean operators.
- We *extend* the existing SPARQL query language with full-text operators, thus increasing the expressivity of the language, the flexibility in profile definition, and precision of delivered information.
- We present a *novel* SPARQL query indexing algorithm that supports Boolean

Information Filtering up to 96% faster that its state-of-the-art competitors.

- We identify and present different algorithmic alternatives to facilitate the structural and textual indexing of profiles and experimentally assess their performance with a real-word data set.

1.4 Thesis structure

The rest of the work is organised as follows. Chapter 2 surveys related work in the fields of information filtering, publish/subscribe systems supporting RDF data, and appropriate data structures for profile indexing and publication filtering. Chapter 3 presents the data model developed to solve the problem of representing user profiles wit text constraints in SPARQL, and describes the indexing Algorithm RTF and it's variants developed to solve the indexing and filtering problem. Chapter 4 gives the experimental evaluation of the developed algorithms with a real-world data set and compares them against existing state-of-the-art solutions. Finally, Chapter 5 gives conclusions and future directions for our work.

Chapter 2

Related Work

In this chapter we present related work in RDF-based publish/subscribe systems. At first, we briefly provide a background discussion on Information Filtering. Subsequently we present works that focus on RDF-based publish/subscribe systems. Finally, we give the basis for the trie indexing structures used in the Information Filtering field.

2.1 Information Filtering

Information Retrieval(IR) and Information Filtering(IF) are often referred as the two sides of the same coin [9]. While a few common points overlap in the research fields, such as the aim of both, to provide users with information delivery, fundamental differences are present. Some of them are the timeliness of *information delivery*, as Information Retrieval provides users with informations that match their interests independently of the publication time, while Information Filtering aims for newly provided content as soon as this is published. The definition of *user needs* also presents variations, as in Information Retrieval users pose a short-sized one-time query to satisfy their short-term interests while the delivered results could be hundreds of items of matching data, leaving users to decide which one satisfies the most their needs. On the other hand, queries on Information Filtering systems are rather extensive in order to capture specific, usually long-term, interests of the users without overwhelming them with data, but aiming at the delivery of the exact information the users desire. Given these needs, the representation of information as well as the indexing algorithms and techniques used to fulfil the requirements of IR and IF vary vastly.

In the field of selective dissemination of information the first work that addressed the topic was conducted by H.P. Luhn [44], where a “Business Intelligence System” is described. In the proposed system, users subscribe by describing their interests with a profile creation; subsequently the document filtering system used this profile to deliver to the users a list of newly published documents. From this delivered list of publications users are able to select and order the documents that meet their interests. At the time of publishing this work, the procedure of selecting documents was coined with the term of *selective dissemination of new information*. The term Information Filtering was later conceived by P.J. Denning [23], where the need of a system capable to filter incoming mail messages and arrange them in a prioritized order is described.

In the first work conducted in the field, researchers in the area of Information Filtering focused their efforts to accurately represent the user interests [47], while others emphasized on the optimization of the filtering process [33]. The work done by M. Morita and Y. Shinoda [47] included the utilization of techniques to observe and record the user interests , while indexing subsets of terms in order to determine the documents that really concern a user. The work done by D.A. Hull et al. [33] focused on studying the filtering process by applying machine learning techniques, where a combination of known techniques was used in order to increase the accuracy of the results delivered to the users. Other techniques include the utilization of statistical data for filtering incoming documents such as LSI-SDI [29] which uses the LSI technique in the filtering process.

One of the first studies conducted to address the performance factor on Information Filtering is [10], where an Information Filtering system capable to filter large volumes of data is described. The researchers developed a server where great amounts of data are published in a high rate, and algorithms that support the *Vec-*

tor Space Model of Information Retrieval, by enhancing the SQI algorithm of T.W. Yang and H. Garcia-Molina [76] are suggested. Another influential system is InRoute [13], which is based on network topology and aims at more efficient filtering. InRoute created networks of documents and queries, while it utilized belief propagation techniques to deliver results. Apart from performance issues researchers also focused on the adaptiveness that a filtering system must exhibit [14, 80] and the way queries represented in Vector Space Model must be altered depending on the past filtered documents.

Besides the statistical approaches described above, Information Filtering systems supporting the Boolean model of Information Retrieval were also developed. One influential system in this area is LMDS [79], that is based on the idea of least common trigramms to deliver faster document filtering. LMDS indexes user profiles and documents under their least common trigramms to facilitate matching. During the filtering process a table that defines which profiles matched with the incoming document is constructed; since false positive results may occur from this process, a post-filtering procedure is required in order to determine the true positives.

One of the main problems in publish/subscribe systems is the determination of user interests and an optimal way to represent them. Approaches such as S. Nilsson's and G. Karlsson's [49] for the creation of representative profiles of user interests , suggest solutions of using genetic and machine learning algorithms. This approach aims in the creation of richer user profiles where users rate the information that is delivered to them; with this indirect way the higher rated. Queries that users suggested will be used to construct the next generation of queries. Y.-I. Chang et al. [17] based on the observation that user interests change over the time, developed a personalized filtering system with the ability to easily reflect the changes of the users interest on their profiles. This was achieved with the usage of a knowledge mining system, attempted to identify and categorise the user interests in short term and long term. This categorisation aimed at a better indexing of the profiles as well as the better reorganization when changes are frequent. Other approaches stir away from the classic model of profile representation as a set of words and study the correlation of terms [50]. In these approaches, a network is created consisting of weighted word nodes in order to represent closely user interests as well as to increase the filtering precision.

Other works [40, 81, 82] pay attention in the filtering process of the documents and divide it in two stages in order to accelerate the procedure and to provide higher precision in the delivered results to the users. In the first stage the most irrelevant incoming documents are excluded from the filtering procedure, while in the second stage the remaining documents, which is a relatively smaller number, are filtered with a pattern matching method. Their objective is to reduce the total of non relevant documents that reach the end user. On the same assumption that word patters offer a richer representation of data, than the one of bags of words [45], and the window of words, N. Nanas and M. Vavalis [48] attempted to locate semantic similarities in the documents and profiles. In this spirit works [3, 41, 42] developed methods to evaluate the negative feedback of users for documents, combine them with pattern recognition and deliver filtering results with reduced noise.

An intriguing approach in the filtering stage is given by W. Vanderbauwhede et al. [73] as the filtering procedure is implemented by using Field Programmable Gate Arrays (FPGAs) in order to speed up the most resource intensive routines. In a similar spirit, [26, 63] try to make use of modern processors with threads and achieve higher throughput of filtering documents with parallelization of the filtering algorithms.

2.1.1 Information Retrieval

In this section we present some characteristic works on information retrieval on RDF systems with full-text query support.

Panzeri and Pasi [54, 55, 56] build an extension on XQuery [16] to support full-text query capabilities for Information Retrieval systems. This work introduces two new predicates *below* and *near* for the XQuery full-text language. The purpose is to provide users with ranked results for the structural matching of their queries.

Users can define constraints about the structure and content of the data they want to retrieve. This is achieved while the underlying structure of the data remains unknown to the users. The proposed extension provides a ranking on structural or content level of nodes or a combination of both rankings. Each XML element is ranked upon the closeness it has to the provided query. Panzeri and Pasi [54] specify the syntax of the proposed predicates, their semantics and evaluation functions for computing the relevance score of a document path. Additionally in [55, 56] the implementation of the proposed extension on the BaseX query engine [32] is presented. However more time is required for BaseX to calculate the ranking of the matched data, as the evaluation functions must be applied for the *below* and *near* predicates of the extension.

Amer-Yahia et al. [5] developed FleXPath an Information Retrieval framework for querying XML documents. The proposed framework supports queries formulated in XPath, an XML [12] query language, with full-text support. The framework treats the users queries as templates rather than strict representations. By adopting this approach and relaxing the query expressions the framework gives the flexibility to match a higher number of XML documents structurally and provide the users with results similar to their interests. FleXPath provides a set of operators for the queries and a schema for ranking documents depending on the satisfiability on the submitted query. The framework is the first that provides a loose interpretation of the XPath rather than the conventional exact semantic matching.

Mishra et al. [46] developed an Information Retrieval engine that processes and answers SPARQL full-text queries by querying a relational store of structured XML documents. [46] provides a method to transform SPARQL queries with full-text conditions into SQL queries by making use of temporary indices, while the XML documents are stored in the relational database by applying a similar procedure. Finally a scoring mechanism is provided for the documents in the database that matched the issued query.

2.1.2 Information filtering in databases

The majority of research conducted for Information Filtering in the field of databases is based upon the work of M. Franklin and S. Zdonik [30] where the term *selective dissemination of information* initially appears to characterise their research on the DBIS [4] system. The term publish/subscribe system originates from the field of distributed systems but has also been used by researchers on databases systems. Yet another one influential system is SIFT [75, 77]. In SIFT publications are in the form of plain text and queries are represented as word sets; SIFT was the first system that emphasized the importance of the query indexing method, addressed the problem efficiently and achieved high performance rates against a high volume of data [75].

2.1.3 Information filtering in distributed systems

Finally, a few influential distributed publish/subscribe systems will also be described. Driven by the lack of a large scale publish/subscribe system A. Carzaniga et al. [15] developed SIENA, a distributed system that offers alerting services based on events. SIENA used an attribute based data model with capabilities that included user subscriptions and alerting while it supported event publishing. It is the first distributed system that focused in providing expressiveness to user searches and to serve a high number of distributed users. Driven by the concept of SIENA, M. Koubarakis et al. [38] developed DIAS and P2P-DIET [34, 39] while they used the AWP and AWPS data models, the creation of which is based on ideas derived from the Information Retrieval field. These two systems incorporate ideas from Information Retrieval and databases in a single framework, while maintaining the characteristics of SIENA that concern the publish/subscribe side. Another contribution of P2P-DIET is the suggested representation of one-time queries, and the support of super-peer protocols [78]. On the other hand the system iClusterDL [64] presented a way to use a Semantic Overlay Networks (SON) [21] to support both Information Retrieval and publish/subscribe functionalities in a digital library domain.

2.2 RDF-based publish/subscribe systems

In this section we present the work done in publish/subscribe systems that support RDF data natively on user profiles and document publishing.

2.2.1 Centralized publish/subscribe systems

Petrovic et al. [60] present a prototype system S-ToPSS (Semantic Toronto Publish/Subscribe System) to address the problem of semantic matching on publish/-subscribe systems through an ontology. The purpose of this work is to extend the matching process on syntactically different data but with semantically similar information. Three methods are presented in order to enhance the matching process. The first approach includes the translation of an event's attributes to synonyms and matching them also with the existing subscriptions. The second method suggests the usage of taxonomies, that represent general and specialized concepts, and their placing in a hierarchic structure. Attributes of an event, that represent specialized concepts, match with generalized concepts in subscriptions. On the other hand more generalized concepts of publications do not match with specialized concepts in subscriptions, since users have explicitly requested more specific informations. The final proposed approach includes the usage of mapping functions in order to determine relations between attribute-value pairs that are not discoverable by a concept hierarchic or a synonym relationship. The proposed approaches can be used either together or separately, while the user can determine how generalized/specialized publications wants to receive. On the other hand S-ToPSS [60] aimed for a better semantic matching of published data on an ontology, G-ToPSS [61] (Graph-based Toronto Publish/Subscribe System) focused on information dissemination of RDF data on ontologies, by emphasising on scalability and fast filtering of RDF data such as RSS publications and PDF documents. The publications on G-ToPSS are represented as a directed labelled graph and a set of triples (subject, predicate, object) is used to represent the graph on the system. Similarly the user subscriptions are represented as a directed graph pattern, while 5-tuples are used to capture the profiles, similar to publications plus the constraints on object and subject. A two-level hash table is used to represent the graph that indexes the subscriptions. The matching algorithm is based on the traversal of the publications and subscriptions graphs. Similarly to S-ToPSS [60] a class taxonomy is used to enhance the semantic matching of G-ToPSS.

Lately a lot of work is focused in supporting publish/subscribe systems with ontology capabilities. Wang et al. [74] propose an ontology-based publish/subscribe system which can support events with complex data structures. The Ontology-based Publish/Subscribe (OPS) system makes use of semantic events to filter them against subscriptions in an ontology. All published events are at first processed into a RDF graph and thereafter are filtered by the system. This conversion aims for a uniform representation of the published data. In the same manner the subscriptions of the users are also represented in a RDF graph, while an effective graph matching algorithm for facilitating the filtering process is provided. During the filtering process the published event's and the subscriptions' RDF graphs are traversed in a BFS manner. Subsequently OPS checks the matching trees that emerge from the graph traversal to determine the subscriptions that matched with the publishing event.

Related work focused in publish/subscribe systems that support data models with attributes and query languages that utilize arithmetic and string operators. Such systems are Le Subscribe [25], the monitoring subsystem of Xyleme [52] and the Xyleme system [52], which uses an attribute-based query system but differentiate from the majority of existing systems by supporting more than conjunctive queries [25].

Park and Chung [57] developed a broker for OWL-based [8] publish/subscribe systems. The focus of the work is to support profiles expressed in SPARQL [62], a form appropriate for filtering incoming documents and events from OWL ontologies. The proposed system, named iBroker, uses algorithms to semantically and syntactically match incoming events generated from an ontology. iBroker makes use of semantics like inverse, symmetric and transitive properties to locate additional relevant profiles that may not match directly. The full-text support in SPARQL is provided without emphasizing the importance of an optimization for the text indexing of user profiles. The text fields included in a SPARQL query are indexed using a value field that can only operate a simple comparison of alphanumerics. iBroker is a work that has common goals with ours and later in the thesis we are going to extend iBroker to our data model and use it to evaluate the algorithms that we propose.

2.2.2 P2P publish/subscribe systems

Chirita et al. [18] build the first P2P publish/subscribe system based on RDF data. All users of the system employ L, a typed-first language subset of QEL [51], to conduct every transaction they seek i.e. advertising new publications, subscribing to the system by expressing their interests and user notifications on new publications. Additionally a set of types, constants and predicates is provided to allow users to describe the data exchanges on the network. Publishers advertise the content they intent to provide in the network using L. In the same manner users subscribe in the system by describing their interests in the publications' content they want to be notified about by making use of the aforementioned set of tools. The system is build on a topology of super-peer architecture. Initially publications, queries and subscriptions are routed to the super-peers, and then they are distributed in the peers depending of their content. The content of data is determined by the RDF schema, property, or value of the data. Each peer is responsible for specific schemas and properties, so profiles or parts of profiles that concern a specific schema are routed for indexing to the corresponding peers. The indexing is coordinated by the super peers as they maintain routing indices about which peers are responsible for every schema or property. In the same manner when an advertisement/publication occurs, super peers rout the data to the responsible peers in order to filter the advertisement and notify the interested users about the publication. This work provides optimization of processing RDF-based subscriptions and publications, by reducing the network traffic, with the utilization of subscriptions' content that express similar interests.

Liarou et al. [43] studies the problem of evaluating multi predicate conjunctive queries in publish/subscribe systems. The proposed system operates on a P2P network while the routing algorithms utilize Distribute Hash Tables (DHTs), in order to achieve fast transactions between the peers, more specifically the protocol of Chord [69] is used. As conjunctive queries require more than one triple to be satisfied, the focus of this research is to distribute the load of the matching process. Since the set of triples consisting a profile can be satisfied asynchronously and the network must maintain knowledge of all the satisfied triples; thus it must remain updated about sub-queries (triples) that have already matched and detect when a query is answered. The speed up of the matching process is achieved by organizing efficiently the sub-queries (triples) in the network nodes in order to accelerate the filtering process. Two algorithms are proposed for the query evaluation process. In the first algorithm, all peers involved in a subscription are organized in a chain-like manner while a sequential matching of sub-queries is used for the matching procedure. Finally in the second algorithm multiple peers are responsible for the query indexing and matching, thus creating multiple node chains that speed up the filtering procedure.

Similarly Pellegrino et al. [58] build a publish/subscribe P2P network based on CAN [65]. Their goal is to support RDF events on the P2P network and study the messaging paradigm. Particularly the proposed model allows users to formulate queries and profiles making use of SPARQL and publish data and events using the RDF data model. Users subscriptions are divided in quadruples and indexed in the responsible peer, determined by the CAN protocol, in a multi-dimensional indexing space. An algorithm for parallel matching of publishing events is provided. Given that a publishing event occurs more frequently than a subscription, a scheme is provided for not indexing the publications multiple times on the network. The data of the event that concern a peer are stored while the rest of the event is forwarded to other peers. Although the publication process becomes more resource intensive a much faster notification of the subscribers is achieved.

2.3 Using tries for Information Filtering

In this section we present the basis for the trie indexing structures used in the Information Dissemination field as well as their variations that have arisen. The proposed algorithm RTF utilizes the trie structures both in the structural and the textual parts of the RDF-based user profiles for it's indexing purposes.

The concept of tries was first presented in the research of de la Briandais [22], but the real term trie was *tries* was conceived by Fredkin [31] and was derived from the term re*trie*val. Tries are used in a high variety of applications including, dictionary management [1, 6, 37], text compression [11], natural language processing [7, 59], pattern recognition [28, 67], IP routing [53] and even for searching reserved words in a compiler [2]. The range of applications that tries apply classify them as general purpose data structures with characteristics known to us by a large number of studies [24, 27, 35, 37, 66, 67].

There are quite a few methods on how the trie nodes are implemented and the selection of one depends on the type of application their destined to be used. Two methods prevail as they are the most commonly used. The first method requires the usage of tables in the size of the vocabulary [31], while the second method favours the usage of linked lists with non empty elements as the roots of sub-Tries [22, 36]. Tries implemented with tables are suitable for applications with small vocabulary sizes; on the other hand the implementation of linked lists is suitable for applications that aim in large vocabulary sizes or nodes that have a small number of children.

There are two main methods applied in order to reduce the size of a given trie, either by reducing the size of the nodes or reducing the number of nodes needed to represent a given set of terms. Compact Tries [70] are formations that aim to reduce the number of nodes used to represent a given set of terms. This is achieved by minimising the path of nodes that lead to a leaf of the trie, reducing the branching factor to one node. In another approach the word index can be treated as a total rather than a sequence of terms, affecting the number of nodes and creating a sparser forest. However, [20] proved that the problem of computing the smallest possible trie is *NP-complete* , therefore several heuristic methods have been proposed [19].

Chapter 3

Data Model and Algorithms

In this chapter we present the data model defined for the publication and profile model in our system. Additionally we present the algorithms developed to facilitate the indexing of profiles and the matching of incoming publications.

3.1 Data and profile model

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 would enable 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 publish /subscribe ontology systems which demand sophisticated filtering mechanisms for matching massive ontology data against thousands of user profiles.

3.1.1 The SPARQL query language

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 provided with several query forms for querying datasets SPARQL represents

Constraint	::=	FTExpression
FTExpression	::=	ftcontains()
FTExpression	::=	FTMain ("ftand“ FTMain)*
FTMain	::=	String

Table 3.1: SPARQL extension grammar.

a fully-fledged language. However, is still lacks the support of a complete full-text retrieval mechanism which uses sophisticated algorithms and data structures to minimise processing load and memory requirements. Only regular expression queries are supported in the specification where an XPath function is invoked neither providing any optimizations for the syntactic advantages that SPARQL provides, nor leveraging on the structural properties of the datasets.

Since we focus our attention on full-text filtering of ontology data we are interested only in property elements with an 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, where as typed literals have a lexical form and a datatype URI. We only consider the case where the datatype URI is an XML Schema built-in datatype.

The equivalent extension for information retrieval on XML publications [16] 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. Context queries can be subsequently broken down into phrase and proximity queries.

3.1.2 SPARQL full-text extension

We propose a similar extension to the SPARQL syntax to address these types of queries in RDF datasets. Notice that SPARQL supports different profile forms that affect only the form of the answers returned and not the graph matching process itself. To preserve this expressibility we view the full text operations as an additional

filter of the profile variables hence leaving cost-efficient strategies to the profile optimizer. In this context we define a new binary operator $ftcontains$ (full-text contain) that takes as input a variable of the profile and a full-text expression that operates on the values of this variable. The profile signature of the operator is expressed as the following function:
$xsd : boolean : ftcontains(variablevar, fulltextexpression)$

In this context the subject of the 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 either typed or untyped. A full text expression is evaluated only against a literal so the variable *var* is always the object of the triple pattern in the graph pattern. The subject and/or predicate of the triple pattern can be constants.

The rules for the extended SPARQL syntax are listed in Table 3.1. Using this syntax we carefully designed a new set of full-text queries shown in Figure 3.1 which currently can not be efficiently evaluated by existing publish-subscribe ontology systems. These start from simple keyword and phrase matching queries and are easily extended to specialized conjunctive matching. SPARQL Query 1 matches all publications that are of type Article and have a property Title with a string literal content that contains the keyword “Greece”. SPARQL Query 2 is a phrase matching profile which is represented by a proximity formula of zero minimum and maximum distance between the words of the phrase. It matches all publications that are published and are of type Article, while the body of the publication must contain the phrase “economic crisis”. Finally, SPARQL Query 3 matches all articles that are published by the “Economist”, contain the word "Greece in their title and their body contains the words "economic“ and "measures” in it at any given distance.

SPARQL profile 1

```
SELECT publication
WHERE {publication type Article.
       Article hasAttribute Title.
       FILTER ftcontains(Title, "Greece")
}
```

SPARQL profile 2

```
SELECT publication
WHERE { publication type Article.
        Article hasAttribute Body.
        FILTER ftcontains(Body, "economic crisis")
}
```

SPARQL profile 3

```
SELECT publication
WHERE   {publicatition type Article.
    Article hasAttribute Publisher.
    Publisher hasAttribute Name
    Article hasAttribute Title.
    Article hasAttribute Body.
FILTER ftcontains(Name, "Economist")
FILTER ftcontains(Title, "Greece")
FILTER ftcontains(Body, ("economic" ftand "measures"))
}
```

Figure 3.1: SPARQL Queries presenting the extended syntax proposed.

3.1.3 SPARQL subscription representation

Users express their interests in a defined manner by making use of the SPARQL query language and the full-text extension we supply. Thus in this section we explain the notion for processing and representing SPARQL queries in the form of RDF triples.

As discussed earlier, SPARQL provides a formal way for querying RDF data. SPARQL is a standardized language for the users to define their interests, providing expressiveness and comprehensive tools to exploit language features. Apart from the aforementioned, SPARQL is nevertheless a representation of data that can be translated in a series of conjuncts using RDF triples. This transformation in triples can provide us with great advantages. As the data retain their information we can utilize better techniques for managing user profiles, thus providing clustering opportunities in the indexing phase and delivering better performance during the filtering phase. Based on this notion, every SPARQL profile consists of individual parts represented in a form that include three fields named *subject*, *predicate* and *object*. In our approach fields of subject, predicate and object are fixed values that must match triples with these exact values.

Definition 1 *We define a profile p as a series of conjuncts using RDF-triples. Each triple has three* mandatory *attributes, namely* subject *(s),* predicate *(p) and* object *(o). There is an additional, non-mandatory, attribute that facilitates the representation of the full-text operators and their textual restrictions. The following formula is used in order represent the user profiles:*
$p = t(S_1, p_1, O_1, \{FT_1()\}) \wedge t(S_1, p_2, O_2, \{FT_2()\}) \wedge ... \wedge t(S_1, p_n, O_n, \{FT_n()\})$

In our system each SPARQL profile has a mandatory field called *publication*; every profile posed in RTF must bear this field and use it in the *SELECT* clause. Additionally, in the *WHERE* clause of a SPARQL profile the conditions that the profile must meet in order to match with a publication are described. The conditions that form the *WHERE* clause are conjunctive. This means that when the conditions are satisfied the user can be notified about the publication. To better demonstrate our approach, in Figure 3.2 we give two representative examples of SPARQL queries, without full-text operators. We are going to proceed and demonstrate how these SPARQL queries are processed in RDF-triples and later on extend these triples to include full-text support.

SPARQL profile 4

```
1 SELECT publication
2 WHERE {publication type Article.
3        Article hasAttribute Title.
4        Article hasAttribute Body.
5 }
```

SPARQL profile 5

```
1 SELECT publication
2 WHERE {publication type Book.
3        Book hasAttribute Title.
4        Book hasAttribute Body.
5        Book hasAttribute Publisher.
6        Publisher hasAttribute Name.
7 }
```

Figure 3.2: SPARQL profile examples presenting simple user subscriptions.

Example 1 *We present SPARQL profile 4 (Figure 3.2), a simple profile, where the user wants to be informed about all publications that are articles. This is achieved by specifically asking in his request that the publication's type must match with the defined type* Article *hence the second line of the profile* publication type Article. *Additionally the user requests that the publication has a field named Title and a field named Body, hence the third and fourth lines of the profile* Article hasAttribute Title *and* Article hasAttribute Body, *respectively. As we can observe each part of the profile can be matched independently thus it can be breaked into autonomous parts and stored separately; these parts are called RDF-triples. Consequently the SPARQL profile 4 can be expressed in the following form as a set of conjunctive RDF-triples:*

$$\begin{aligned} q_1 =& (publication, type, Article) \wedge (Article, hasAttribute, Title) \\ & \wedge (Article, hasAttribute, Body) \end{aligned} \tag{3.1}$$

Observe that the information of the profile are retained in a different form, while giving us a more flexible representation of data to work upon.

Example 2 *In the same manner we can exam SPARQL profile 5 (Figure 3.2) which contains three constraints in it's WHERE clause. The user requests a publication that must be published by a publisher (*Book hasAttribute Publisher*) and be of a type Book (*publication type Book*), the publication should have two mandatory fields named Title and Body (*Book hasAttribute Title *and* Book hasAttribute Body*), and the publisher should have a name (*Publisher hasAttribute Name*). Respectively to the first example, Query 5 is expressed with the following series of RDF-triple conjuncts:*

$$\begin{aligned} q_2 = &(publication, type, Book) \wedge (Book, hasAttribute, Title) \\ &\wedge (Book, hasAttribute, Body) \wedge (Book, hasAttribute, Publisher) \\ &\wedge (Publisher, hasAttribute, Name) \end{aligned} \tag{3.2}$$

We can see that RDF-triples are sufficient for representing a sub-set of SPARQL queries. Below we present how we can extend RDF-triples to accommodate additional information for supporting our full-text operator *ftcontains* and consequently other full-text operators that may be similarly defined.

Full-text restrictions in SPARQL queries can be applied in every triple. The user, while posing a profile defines the full-text operators. These restrictions concern the *object* attribute in the RDF-triple. The *FILTER* clause in SPARQL queries 6 and 7 shown in Figure 3.3 includes the operators that must be applied on the chosen field. These restrictions must be represented in a simple way in order to be indexed by algorithm RTF. To support this functionality RDF-triples must have an additional non-mandatory field for the object that will be also indexed. This field includes the full-text operator and it's supplementary fields as specified in Section 3.1.2. A RDF-triple that bears a full-text operator is named *quadruple*. We proceed to give examples that demonstrate the usage of quadruples in supporting full-text operators.

Example 3 *In SPARQL profile 6 (Figure 3.3) the user subscribes for a publication that is of type Article as shown in the second line* publication type Article *and it*

SPARQL profile 6

```
1 SELECT publication
2 WHERE {publication type Article.
3        Article hasAttribute Title.
4        Article hasAttribute Body.
5 FILTER ftcontains(Title, "olympic" ftand "games")
6 FILTER ftcontains(Body, "olympic" ftand "games" ftand "rio")
7 }
```

SPARQL profile 7

```
 1 SELECT publication
 2 WHERE {publication type Book.
 3        Book hasAttribute Title.
 4        Book hasAttribute Abstract.
 5        Book hasAttribute Body.
 6        Book hasAttribute Publisher.
 7        Publisher hasAttribute Name.
 8 FILTER ftcontains(Title, "olympic")
 9 FILTER ftcontains(Abstract, "olympic" ftand "rio")
10 FILTER ftcontains(Body, "olympic" ftand "committee")
11 FILTER ftcontains(Name, "the" ftand "wall" ftand "street" ftand "
      journal")
12 }
```

Figure 3.3: SPARQL profile examples presenting the utilization of full-text operators.

*has two attributes named Title and Body as shown in the third and fourth line of the profile respectively (*Article hasAttribute Title *and* Article hasAttribute Body*). Additionally attribute of Title must be filtered and should contain the words "olympic" and "games" in it's text field, attribute Body must contain the words "olympic", "games" and "rio". The two FILTER clauses can be found in the fifth and sixth line of Query 6. The filtering clause must be applied to the attributes Title and Body respectively, therefore must be included in the respective triples as additional infor-*

mation. The set of triples and quadruples that represent this SPARQL profile is the following:

$$\begin{aligned} q_3 =&(publication, type, Article) \wedge \\ &\wedge(Article, hasAttribute, Title, \\ &\quad ftcontains(\text{"}olympic\text{"} ftand \text{"}games\text{"})) \\ &\wedge(Article, hasAttribute, Body, \\ &\quad ftcontains(\text{"}olympic\text{"} ftand \text{"}games\text{"} ftand \text{"}rio\text{"})) \end{aligned} \tag{3.3}$$

Hence that we use a quadruple only where there is a full-text operator that needs to be applied to the *object* of a triple.

Example 4 *In the same manner we process SPARQL profile 7 (Figure 3.3), where the user requests for a publication of type Book. Each attribute of this publication Title, Abstract, Body and Publisher's Name must be filtered and contain specific keywords. The filtering clauses can be found in lines eight to eleven of the SPARQL profile 7. This profile is represented with the following set of triples and quadruples:*

$$\begin{aligned} q_4 =&(publication, type, Book) \\ &\wedge(Book, hasAttribute, Title, \\ &\quad ftcontains(\text{"}olympic\text{"})) \\ &\wedge(Book, hasAttribute, Abstract, \\ &\quad ftcontains(\text{"}olympic\text{"} ftand \text{"}rio\text{"})) \\ &\wedge(Book, hasAttribute, Body, \\ &\quad ftcontains(\text{"}olympic\text{"} ftand \text{"}committee\text{"})) \\ &\wedge(Book, hasAttribute, Publisher) \\ &\wedge(Publisher, hasAttribute, Name, \\ &\quad ftcontains(\text{"}the\text{"} ftand \text{"}wall\text{"} ftand \text{"}street\text{"} ftand \text{"}journal\text{"})) \end{aligned} \tag{3.4}$$

Users typically, are not familiar with the structure of publications to be made available in a system, when submitting a subscription. Thus, applying specific restrictions to the publications may result in missing notifications that may match

SPARQL profile 8

```
SELECT publication
WHERE {publication type *.
       * hasAttribute Title.
       * hasAttribute Body.
FILTER ftcontains(Title, "olympic" ftand "commitee" ftand "
   president")
FILTER ftcontains(Body, "olympic" ftand "rio" ftand "stadium")
}
```

SPARQL profile 9

```
SELECT publication
WHERE {publication type *.
       * hasAttribute Title.
       * hasAttribute Abstract.
       * hasAttribute Body.
       * hasAttribute Publisher.
       Publisher hasAttribute *.
FILTER ftcontains(Body, "olympic" ftand "committee")
FILTER ftcontains(*, "bbc" ftand "sports")
}
```

Figure 3.4: SPARQL profile examples presenting the utilization of wildcards operators.

a users' interest. The need for a more flexible subscription scheme seems necessary in order to better describe users' interests. Additionally when less restrictions are applied to the published data a better quality of information delivery can be achieved. In addition to supporting full-text we extended our publish/subscribe indexing scheme to support the wildcard operator applied in RDF-triples. With a combination of full-text and wildcard operators we are able to provide a set of tools to support more expressive subscriptions tailored to user needs. We provide users with the flexibility to define profiles where the subject, predicate or object of a triple may match any value of the publication. This flexibility results to better content delivery as the description of interests is more expressive.

Example 5 *SPARQL profile 8 (Figure 3.4) demonstrate an example of a user that subscribes for a publication independently of the type (*publication type **) with two constraints: it must contain a Title and a Body attribute (** hasAttribute Title *and* * hasAttribute Body*). Additionally the Title and Body attributes must contain specific keywords on their text fields. The profile is represented with the following set of triples and quadruples:*

$$\begin{aligned} q_5 =&(publication, type, *)\\ &\wedge(*, hasAttribute, Title,\\ &\quad ftcontains(\text{"}olympic\text{"}ftand\text{"}commitee\text{"}ftand\text{"}president\text{"}))\\ &\wedge(*, hasAttribute, Body,\\ &\quad ftcontains(\text{"}olympic\text{"}ftand\text{"}rio\text{"}ftand\text{"}stadium\text{"})) \end{aligned} \tag{3.5}$$

Example 6 *SPARQL profile 9 (Figure 3.4) gives us an example of the use of wildcard in the object field while supporting full-text restrictions. In this example a user subscribes to publications of any type that must have the attributes of Title, Abstract, Body and Publisher while the publisher may have any attribute (* Publisher hasAttribute **) that contains the words “bbc” and “sports” (*ftcontains(*, "bbc" ftand "sports")*) as shown in the seventh and ninth lines of the profile. The profile is represented with the following set of triples and quadruples:*

$$\begin{aligned} q_6 =&(publication, type, *) \wedge (*, hasAttribute, Title)\\ &\wedge(*, hasAttribute, Abstract)\\ &\wedge(*, hasAttribute, Body, ftcontains(\text{"}olympic\text{"}ftand\text{"}committee\text{"}))\\ &\wedge(*, hasAttribute, Publisher)\\ &\wedge(Publisher, hasAttribute, *, ftcontains(\text{"}bbc\text{"}ftand\text{"}sports\text{"})) \end{aligned} \tag{3.6}$$

We give the flexibility to the users to define a triple that has all it’s fields as wildcards, *(* * *)*. A profile like this may be applied in order to return all publications regardless of their structure. By applying a full-text operator the user may filter all publications and utilize our system ignoring the structural form of the data published in our system. The result will be exactly as if the user would be using a publish/subscribe system without RDF support.

In this section we have demonstrated the usage of SPARQL queries to express user interests. We show how we transform SPARQL queries to RDF-triples and how to support full-text operator with this representation. We also showed how wildcards may be used as a mean to enrich the expressivity of the profile language. We do not expect from the average user to utilize all the presented capabilities of our subscription language. However a user can use the full potential of the subscription language through a suitably adapted and intuitive interface.

3.1.4 The publication model

Conceptually, the context in our system is defined as a publication which is represented as a set of triples that are expressed using RDF. 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 our proposed 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. RTF may 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 2 *We define a publication pub as a series of RDF-triples. Each triple has three attributes, namely* subject *(s),* predicate *(p) and* object *(o). There is an additional, field that represent the textual information that an attribute may have. The following formula is used in order to represent a publication in our system:*
$pub = t(S_1, p_1, O_1, \{text_1\}) \wedge t(S_1, p_2, O_2, \{text_2\}) \wedge ... \wedge t(S_1, p_n, O_n, \{text_n\})$

3.2 Algorithms

We present RTF, an algorithm developed for indexing multi-predicate continuous queries in publish/subscribe ontology systems. We extend the current standard of the industry for querying RDF data, SPARQL, to support full-text filtering in user profiles. In this chapter we provide the algorithms and methodologies we developed for processing SPARQL queries as well as the indexing structures used to represent RDF-based subscriptions and publications. In Section 3.2.1 In the following sections we present the algorithm RTF which was developed to study the filtering problem, and give a detailed analysis of the indexing structures used to store the RDF-triples. Thereafter the indexing structures for the text part of subscriptions are presented in Section 3.2.2. The filtering process is outlined in Section 3.2.3. Finally, in Section 3.2.4 iBroker a state-of-the-art competitor is presented.

3.2.1 The algorithm RTF

In the previous section we demonstrated the idea of representing SPARQL queries in the form of RDF-triples. In this section we will give the data structures developed to index RDF-triples and support fast filtering of incoming ontology data. We present the indexing of the attributes *subject*, *predicate* and *object*, and omit the indexing of full-text as it will be discussed in detail in Section 3.2.2.

In publish/subscribe systems users pose hundreds of thousands of queries that must be indexed in an efficient way. User subscriptions tend to contain a plethora of common elements, and as the number of queries increases so tend to do their common parts. We aim to exploit those commonalities in order to achieve better clustering in the indexing phase and leverage this clustering to deliver better filtering times during the publication phase.

Common solutions to capture profile commonalities in the modern literature suggest the usage of a two-level hash table [57, 60, 61] as a means to represent RDF-triples and their correlations. However this approach may miss a lot of clustering opportunities since clustering common common triples that occur in different users' profiles is not possible. On the other hand, our approach is based on *tree structures*. Tree structures are often used in applications where clustering of common elements is essential [1, 6, 11, 37, 72]. The deployment of such structures will provide better opportunities to capture the common elements between user profiles.

To index sets of RDF-triples RTF uses three main data structures: (i) a *forest of tries* that store the three attributes of RDF-triples *subject*, *predicate* and *object*, and (ii) a *hash table* that provides efficient access to the roots of the tries in the forest, (iii) a *two level table* with *linked lists* for the indexing of the profiles called the *Profile Table*.

RTF during the indexing phase receives a profile that consists of two fields, a unique *profile identifier* and a set of conjunctive RDF-triples and quadruples that represent a SPARQL profile q_i. During the indexing phase RTF assigns every triple a unique identifier t_i. The identifier t_i will be used to be link every triple to the profile stored in the *RDF-triple forest* and in the Profile Table.

Example 7 *For instance consider the SPARQL queries presented in Figures 3.2, 3.3 and 3.4. These SPARQL queries were processed into RDF-triples as shown in Section 3.1.3 and the resulting RDF-triples and quadruples are presented in Table 3.2. On the first column of Table 3.2 the* profile identifier *for each profile is shown, in the second column of Table 3.2 the set of triples that constitute the respective profile is presented, while in the third column of Table 3.2 the* triples identifier *is unique.*

After the assignment of triples identifiers, RTF proceeds for each profile to store the profile identifier along with the triples' identifiers into the Profile Table. The Profile Table is comprised by two fields: (i) the unique identifier of the profile, (ii) a table that stores the unique identifiers of the RDF-triples and quadruples that form the profile.

Example 8 *Let us consider the SPARQL profile 4 that is presented in Figure 3.2. The set of RDF-triples and quadruples that represent the SPARQL profile are presented in Table 3.2. On the first column of Table 3.2 the* profile identifier *and it's* *value* q_1 *is shown. The three RDF-triples that represent* q_1 *are shown in the second column of Table 3.2, while in the third column you may see the unique* triple identifiers *assigned to the three triples of* q_1 t_1*,* t_2 *and* t_3 *respectively.* RTF *indexes the* profile identifier *and the* triple *identifiers into the Profile Table. Figure 3.5 presents the Profile Table after indexing all the profiles, triples and quadruples identifiers submitted to* RTF*. At the first cell of the table the profile identifier* q_1 *is stored accompanied by list* $\{t_1, t_2, t_3\}$ *representing the RDF-triple identifiers.*

In the same manner of Example 8 all profile identifiers and their RDF-triples' identifiers of Table 3.2 are indexed into the Profile Table. As presented in Figure 3.5 profiles q_2, q_3, q_4, q_5 and q_6 are stored in the second, third, fourth, fifth and sixth cell of the Profile Table respectively.

The *triple forest* is populated in order to store the triples compactly by exploiting their common elements. *Every triple forest* consists of trie nodes n, where:

- An attribute of an RDF-triple, denoted by $atrb(n)$, is stored.
- A list of the children it has, denoted by $chld(n)$, is stored
- If n is a leaf node the n also maintains two lists of triples' identifiers. One dedicated to storing the identifiers of triples that bare no full-text operators, denoted by $tripIds(n)$. While the other list is used to store the identifiers of triples that bare full-text operators, namely quadruples, denoted by $quadIds(n)$.

When a new profile p arrives RTF iterates through the set of triples $t_1 ... t_i$ that form p. The algorithm indexes every triple t_n of the profile p separately in the *triple forest*. During the indexing phase RTF searches the *triple forest* for a suitable place to index the current triple t_c. The first triple that arrives in an empty triple forest, creates three new nodes n_s, n_p and n_o that represent it's three fields *subject*, *predicate* and *object* respectively. The newly created nodes are linked with a hierarchy of parent-child, more specifically node n_o is assigned as a child to node n_p while node n_p is assigned as a child to node n_s. This results to a tree T that has as root node n_s and leaf node n_o while it's depth equals two. Finally the triple identifiers are stored into the Profile Table.

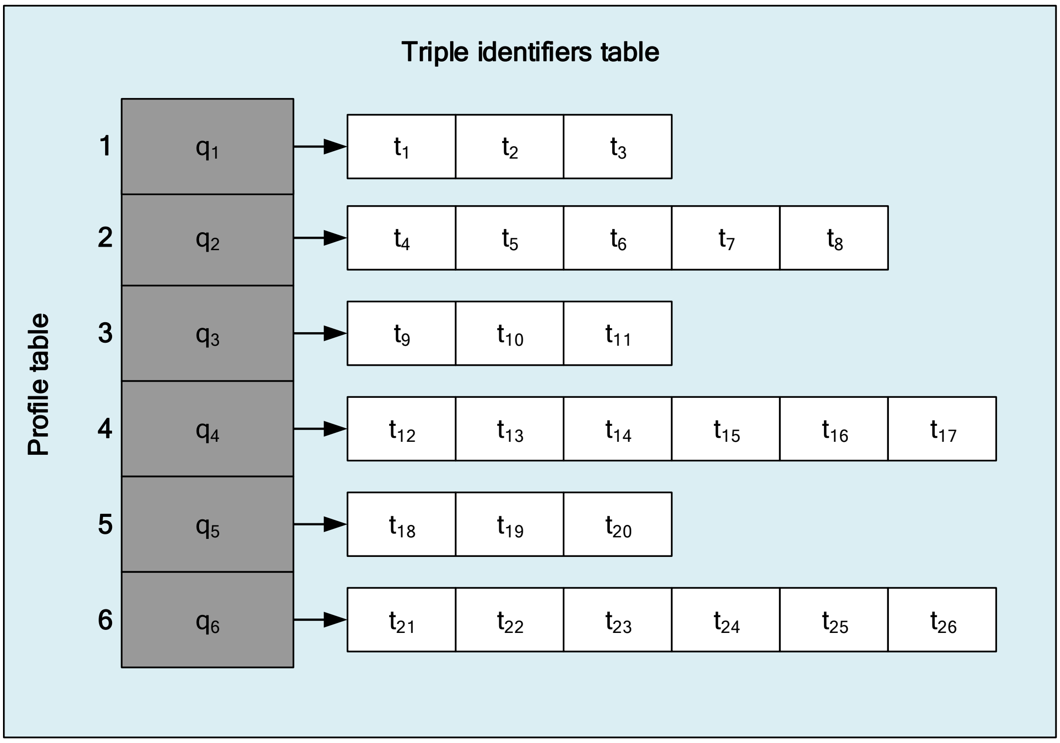

Triple identifiers table
Profile table
1 q_1 → t_1 t_2 t_3
2 q_2 → t_4 t_5 t_6 t_7 t_8
3 q_3 → t_9 t_{10} t_{11}
4 q_4 → t_{12} t_{13} t_{14} t_{15} t_{16} t_{17}
5 q_5 → t_{18} t_{19} t_{20}
6 q_6 → t_{21} t_{22} t_{23} t_{24} t_{25} t_{26}

Figure 3.5: The data structure Profile Table after the indexing phase of Algorithm RTF.

Example 9 *Consider the first profile q_1 of Table 3.2. The first triple of that profile that is going to be indexed into the currently empty triple forest is t_1 namely* (publication, type, Article)*. Triple t_1 is an RDF-triple without any full-text operator.* RTF *will use t_1's subject to create the first node n_s and use it as a root. The reference to node n_s with $atrb(n_s) =$* publication *will be also stored to Hash Table using the $atrb(n_s)$ as the key to index it. Subsequently the indexing algorithm will proceed to the predicate element of triple t_1 and create a node n_p to store it. The newly created node n_p will have $atrb(n_s) =$* type *and will be assigned as a child to the node n_s by making an entry in the list $chld(n_s)$. Then* RTF *will index the* object *of the triple t_1 by creating one last node named n_o. The n_o node will store the object of t_1 namely $atrb(n_o) =$* Article*. In the next step, the algorithm will proceed to assign the newly created node n_o to the predicate node n_p as a child, that was created in the previous step. This will be achieved by inserting the node n_o in the children list*

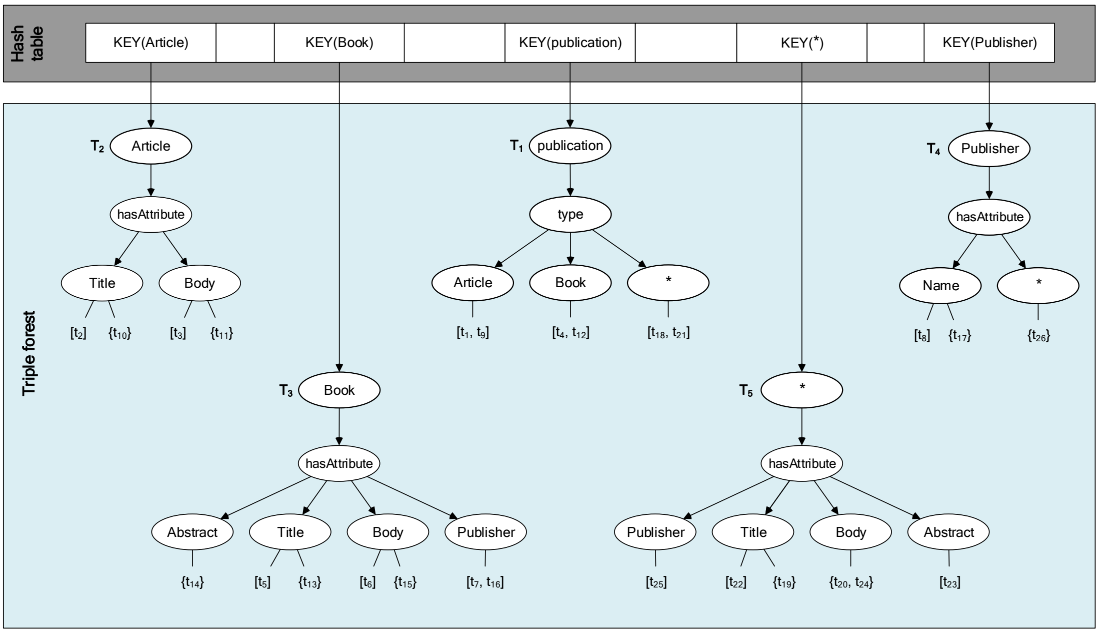


Figure 3.6: The data structure triple forest after the indexing phase of RTF.

of n_p*, denoted as* $chld(n_p)$*. Finally* RTF*, as it indexes a triple without full-text operators will proceed and store* t_1*'s identifier into the leaf node* n_o *namely the list* $tripIds(n_o)$*.*

The result of indexing t_1 *can be found in Figure 3.6, under the trie with the identifier* T_1*. The list tripIds is shown in the figure under the object node and enclosed into the square brackets.*

RTF when inserting a new triple proceeds with the following steps. The first triple t_1 that arrives, creates a tree with it's three fields as demonstrated above. The second triple that arrives will consider to be stored at the existing tree or create a new tree. In general, to insert a new triple t_n, RTF examines the subject s_n of the triple and utilizes the Hash Table to find if there is a candidate tree i.e., a tree T_i that $atrb(T_i) = subject(t_n)$. If a root is found, the indexing algorithm proceeds to examine the node T_i's children in order to determine if there is a child that represents the same predicate as triple's t_n predicate i.e., $atrb(chld(T_i)) = predicate(t_n)$. In the case that a node $child_i$ is located that full fills the above requirements RTF examines the children of *child$_i$* that are responsible for indexing the object fields. If in the list of nodes $chld(child_i)$ is found a node $child_j$ that $atrb(child_j) = object(t_n)$. During the indexing phase if RTF at any give step, fails to locate an existing position, a tree node, to index a field of t_n proceeds to create a new node that stores the needed field. To complete insertion RTF proceeds to store at the final node the triple's t_n identifier. The final node is the one that stores the field of the *object*. Every leaf node of the forest maintains two separate lists $tripIds(n)$ and $quadIds(n)$. The first list aims to store the identifiers of triples that have no full-text operator. While the second list is reserved for storing quadruples, namely RDF-triples that bear a full-text operator. This differentiation is made in order to keep track which triples may match immediately during the filtering phase and which must be further investigated in order to determine if the full-text operator conditions are met. Take a look at Example 10 for a more thorough look at the insertion process applied by RTF.

Example 10 *Consider the first profile of Table 3.2, q_1. The first triple of the profile is indexed as demonstrated in Example 9. The second triple that arrives at the triple forest is t_2* (Article, hasAttribute, Title). RTF *will examine triple's t_2 subject and try to locate a tree T to index it. As there is only one Tree at the current moment in the triple forest with a root $root(T_1)$ where $atrb(root(T_1)) = publication$,* RTF *will determine that there is no existing tree to index the new triple t_2. So it will proceed to create a new tree T_2 that will index the attributes of t_2. The steps that* RTF *is going to follow are exactly the same as described in Example 9 for t_1. More specifically three new nodes n_s, n_p and n_o with $atrb(n_s)$ =* Article, *$atrb(n_p)$ =* hasAttribute *and $atrb(n_o)$ =* Title *respectively will be created. These nodes will be inserted into the trie, i.e. n_s is going to be declared as parent node of n_p and n_p as parent node of n_o. While the node n_s will obtain a reference in the Hash Table. Finally the identifier of t_2, will be added to list of node n_o $tripIds(n_o)$. The resulting tree T_2 after indexing the triple t_2 seen in Figure 3.6.*

We proceed examining how an incoming profile q is indexed by RTF under existing tries, thus utilizing the clustering techniques.

Example 11 *Consider the first profile of Table 3.2, q_1. The first and second profile are indexed as demonstrated in Example 9 and Example 10 respectively. The third triple that arrives at the triple forest after indexing t_1 and t_2 is triple t_3 of q_1 namely* (Article, hasAttribute, Body)*. At first* RTF *will examine triple's t_3 subject,* Article *,and try to locate a tree T to index it. At the current moment in the triple forest there are two trees, T_1 and T_2 where root $atrb(root(T_1)) = publication$ and $atrb(root(T_2)) = Article$.* RTF *will identify T_2 as a tree that can index the subject of t_3 then the algorithm will proceed to examine triple's t_3 predicate where $predicate(t_3) = hasAttribute$. By examining all children of $root(T_2)$ will locate the one and only child that stores the $atrb(child_1) = hasAttribute$. As t_3's predicate can be indexed in $child_1$ the algorithm will proceed to examine t_3's object where $object(t_3) = Body$. The algorithm will examine all $chld(child_1)$ and fail to locate an object with $atrb(child_k) = object(t_3)$, therefore* RTF *will proceed to create a node to store $object(t_3)$. The new node will be attached as a child to $child_1$ while the identifier of triple t_3 will be stored to the list tripIds as it is a triple without a full-text operator. The resulting tree T_2 after indexing the triple t_3 seen in Figure 3.6 under the tree with the identifier T_2.*

After indexing the set of triples t_1, t_2, and t_3 that represent q_1 we may see that the node reusability is high as nodes that represent attributes of the triples are reused and a compact indexing is achieved. Now let us examine what happens when indexing quadruples, i.e., RDF-triples that have a full-text operators. In general RTF when indexes a quadruple proceeds with the following steps. The indexing algorithm, at first, sets aside the full-text operators and proceeds to find a suitable place to store the RDF-triple, as described above. When the indexing place is found RTF instead of storing the triples identifier to list *tripIds* of the object node, it chooses to store the t_i to list *quadIds*. In this way during the filtering phase RTF is aware which triples match and which one will need further investigation. After the insertion of identifier to the list *quadIds* RTF will examine the full-text operator words and index them in a separate indexing structure that is going to be discussed in Section 3.2.2. As the indexing of triples and their full text operators is a different operation we will examine the full-text indexing later in this section.

The list paradigm is easily extendible to more than one full-text operators as a dedicated list for every full-text operator may exist and the triples that correspond to this operator may be indexed in this list.

Example 12 *Consider the third profile of Table 3.2, q_3. We will examine q_3 and the triples that represent it $t_7, ..., t_{10}$ and describe how* RTF *indexes them in the triple forest.*

The first triple that forms q_3 is triple t_9 (publication, type, Article)*, the indexing algorithm will look up the Hash Table and locate tree T_1 that can store the t_9. Triple t_9 will be indexed under T_1 by exploiting the existing tree T_1 which can facilitate the indexing of $subject(t_9)$, $predicate(t_9)$ and $object(t_9)$ without the creation of new nodes. The only addition that will be made is at the object node in which the identifier of t_9 will be added in the list of tripIds.*

In the next step RTF *will proceed to examine triple t_{10} namely* (Article, hasAttribute, Title, ftcontains("olympic" ftand "games"))*. The indexing algorithm will look up the Hash Table and locate the tree T_2 that can index the triple t_{10}, since that $root(T_2) = subject(t_10)$. The child of $root(T_2)$ indexes an attribute with value "hasAttribute", additionally T_2 stores an object node with the same value as $object(t_{10})$. The only addition that will be made is at the node's object list that indexes triple's id. This time the list that is going to be updated with the new triple identifier is list quadIds. As in this case t_{10} is a quadruple that contains a full-text operator and must be distincted from the triples without full-text operators. The full text operators fields will be indexed in the indexing structure especially allocated for this purpose.*

In the final step of indexing profile q_3 there is one last triple that needs to be indexed in the triple forest. Namely that triple is t_{11} (Article, hasAttribute, Body, ftcontains("olympic" ftand "games" ftand "rio"))*, a triple with a full-text operator. In the same manner of indexing the two previous triples,* RTF *will look for a tree T where $root(T) = subject(t_{11})$, and tree T_2 will be located as a candidate. Subsequently the indexing algorithm will examine the only child of $root(T_2)$ and determine that can index the predicate of t_{11}. Then the two children of the predicate node will be* *examined and a node with $atrb(node) = object(t_{11})$ will be found. Finally* RTF *will determine that t_{11} is able to be indexed in T_2 without creating any new nodes and will insert an entry with t_{11} into the list that is responsible for indexing triple's identifiers quadIds.*

The result of indexing q_3 can be found in Figure 3.6, under tries T_1 and T_2. The list quadIds is represented as the structure under the object nodes of T_1 and T_2 and are encloses into braces.

Example 13 *Figure 3.6 shows the forest of triples created when inserting the triples $t_1, ..., t_{26}$ as given in Table 3.2. The first triple t_1 creates the tree T_1, the second triple t_2 creates the forest T_2. The third triple t_3 is indexed under T_2 with the addition of a new object node. The fourth triple t_4 is indexed under T_1 with an addition of a new object node, while t_5 creates a new tree T_3. Similarly,* RTF *inserts the remaining twenty two triples.*

Finally notice that SPARQL queries that include wildcard operators on their fields are not treated differently during the indexing phase. In this way grouping of common elements in these queries is also achieved in the case of wildcards. However, wildcards are treated differently during the filtering phase of the algorithm as we will discuss in Section 3.2.3. Figure 3.7 presents the pseudocode for the RDF-triples indexing. In the next section we give the details of indexing the full-text restrictions of the profiles.

Profile ID	Triplets	tripletID
q_1	(publication, type, Article)	t_1
	(Article, hasAttribute, Title)	t_2
	(Article, hasAttribute, Body)	t_3
q_2	(publication, type, Book)	t_4
	(Book, hasAttribute, Title)	t_5
	(Book, hasAttribute, Body)	t_6
	(Book, hasAttribute, Publisher)	t_7
	(Publisher, hasAttribute, Name)	t_8
q_3	(publication, type, Article)	t_9
	(Article, hasAttribute, Title, ftcontains("olympic" ftand "games"))	t_{10}
	(Article, hasAttribute, Body, ftcontains("olympic" ftand "games" ftand "rio"))	t_{11}
q_4	(publication, type, Book)	t_{12}
	(Book, hasAttribute, Title, ftcontains("olympic"))	t_{13}
	Book, hasAttribute, Abstract, ftcontains("olympic" ftand "rio"))	t_{14}
	(Book, hasAttribute, Body, ftcontains("olympic" ftand "committee"))	t_{15}
	(Book, hasAttribute, Publisher)	t_{16}
	(Publisher, hasAttribute, Name, ftcontains("the" ftand "wall" ftand "street" ftand "journal"))	t_{17}
q_5	(publication, type, *)	t_{18}
	(*, hasAttribute, Title, ftcontains("olympic" ftand "commitee" ftand "president"))	t_{19}
	(*, hasAttribute, Body, ftcontains("olympic" ftand "rio" ftand "stadium"))	t_{20}
q_6	(publication, type, *)	t_{21}
	(*, hasAttribute, Title)	t_{22}
	(*, hasAttribute, Abstract)	t_{23}
	(*, hasAttribute, Body, ftcontains("olympic" ftand "committee"))	t_{24}
	(*, hasAttribute, Publisher)	t_{25}
	(Publisher, hasAttribute, *, ftcontains("bbc" ftand "sports"))	t_{26}

Table 3.2: Table presenting the RDF-triplets obtained from SPARQL queries of Figures 3.2 3.3 3.4

Algorithm index

```
BEGIN
$position \leftarrow NULL$
foreach $t_i \in q$ do                                          ▷ For all triplets forming
                                                                SPARQL Query $q$
  foreach field $f_l \in t_i$ do                                ▷ For each field $f_l$ of $t_i$
    if node $n_k$ such as $atrb(n_k) = f_l$ then                ▷ If a node $n_k$ is found that
                                                                can index $f_l$
      $position \leftarrow n_k$                                 ▷ Store the node's position
      remove $f_l$                                              ▷ Remove field $f_l$ as an
                                                                indexing position has been found
    end if
  end for

  if $position = NULL$ then                                     ▷ If no indexing position has
                                                                been found
    create node $n_s$ such as $atrb(n_s) = subject(t_i)$        ▷ Create node $n_s$ for the
    create tree $T'$ with $root(T') = n_s$                      subject of $t_i$ and use it as root
                                                                for new tree $T'$
    create node $n_p$ such as $atrb(n_p) = predicate(t_i)$      ▷ Create node $n_p$ for the
    $chld(n_s) \leftarrow n_p$                                  predicate of $t_i$ and assign it as
                                                                child to $n_s$
    create node $n_o$ such as $atrb(n_o) = object(t_i)$         ▷ Create node $n_o$ for the object
    $chld(n_p) \leftarrow n_o$                                  of $t_i$ and assign it as child
                                                                $n_p$
    if $t_i$ has $FT$ operator then                             ▷ If $t_i$ is a quadruplet store $t_i$
      $quadIds(n_o) \leftarrow t_i$                             in the list for quadruplets
      indexTest($t_i$)                                          and index the text constraints
    else
      $tripIds(n_o) \leftarrow t_i$                             ▷ Else store $t_i$ in the triplets
                                                                list
    end if
  else
    foreach remaining field $f_r \in t_i$ do                    ▷ For all un-indexed fields of
      create node $n_j$ such as $atrb(n_j) = f_r$               $t_i$ create node $n_j$ to index them
      $chld(position) \leftarrow n_j$                           ▷ Assign each new node to the
      $position \leftarrow n_j$                                 previous one as child
    end for

    if $t_i$ has $FT$ operator then                             ▷ If $t_i$ is a quadruplet store $t_i$
      $quadIds(position) \leftarrow t_i$                        in the list for quadruplets
      indexTest($t_i$)                                          and index the text constraints
    else
      $tripIds(position) \leftarrow t_i$                        ▷ Else store $t_i$ in the triplets
                                                                list
    end if
  end if
end for
END
```

Figure 3.7: Pseudocode for RDF-triple indexing.

3.2.2 Full-text indexing data structures

In the previous section we separated the indexing phase of RDF-triples and the indexing phase of their full-text constraints. We gave the data structures that RTF utilises for indexing RDF-triples. In this section we proceed to describe the data structures that RTF utilises to index the textual parts of the RDF-quadruples. The text indexing algorithms that are used to facilitate the indexing of text fields are algorithms FT-Index and FT-Reorg, which are algorithms based on BestFit-Trie [71] and ReTrie [72] appropriately modified versions to cope with full-text requirements posted by our SPARQL extension. At first a description on how FT-Index operates is provided and subsequently FT-Reorg is presented as it is largely dependent/based on algorithm FT-Index.

Consider the quadruples that were extracted from SPARQL queries 6, 7, 8 and 9 as presented in Table 3.2. In the previous section we presented how RTF indexes their RDF part, now we will demonstrate the indexing their text constraints. Therefore RTF is going to use of their text constraints and their triple identifier in order to index them in the data structures of FT-Index. The text constraints from Table 3.2 are presented solely with their identifiers in Table 3.3.

Algorithm Full-text index

To index queries, that are conjuncts of keywords, Algorithm FT-Index uses two data structures: a *forest of tries* that organises 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 3.3 are organised in the structures of Figure 3.8. Each node n of the trie:

- Stores a keyword of a profile, denoted by $kwrd(n)$.
- If the keywords in a path from the root to node n spell out a profile q then n also stores a reference to q. The list of all references stored in node n is denoted by $id(n)$.
- If n is a leaf node then n also stores one list for each profile q, denoted by

$uexp(n, q)$, containing the keywords of profile q that are not already included in the path from the root to n.

$t_{10} = \{\text{olympic, games}\}$

$t_{11} = \{\text{olympic, games, rio}\}$

$t_{13} = \{\text{olympic}\}$

$t_{14} = \{\text{olympic, rio}\}$

$t_{15} = \{\text{olympic, committee}\}$

$t_{17} = \{\text{the, wall, street, journal}\}$

$t_{19} = \{\text{olympic, committee, president}\}$

$t_{20} = \{\text{olympic, rio, stadium}\}$

$t_{24} = \{\text{olympic, committee}\}$

$t_{26} = \{\text{bbc, sports}\}$

Table 3.3: Text part of RDF-triples.

For instance consider Figure 3.8, where $kwrd(n_1) = \text{olympic}$ and node n_1 stores one reference to triple t_{10}. Consider also node n_2 of trie T_3 that stores the text constraints of $q_{17} = \{\text{the, wall, street, journal}\}$. Since $kwrd(n_2) = \{\text{the}\}$, we have that $uexp(n_2, t_{17}) = \{\text{wall, street, journal}\}$. Finally, note that n_2 contains all keywords of t_{17} (since $t_{17} = kwrd(n_2) \cup uexp(n_2, t_{17})$), thus it also maintains a reference to q_5. The purpose of list $uexp(n, q)$ is to allow for the delayed creation of nodes in a trie; this allows us to choose which keywords from the $uexp(n, q)$ 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 a trie node.

The forest of tries is populated in order to store queries compactly by exploiting their *common keywords*. When a new profile q arrives, Algorithm FT-INDEX considers its keywords and inserts them in a (new or existing) trie in the forest. For

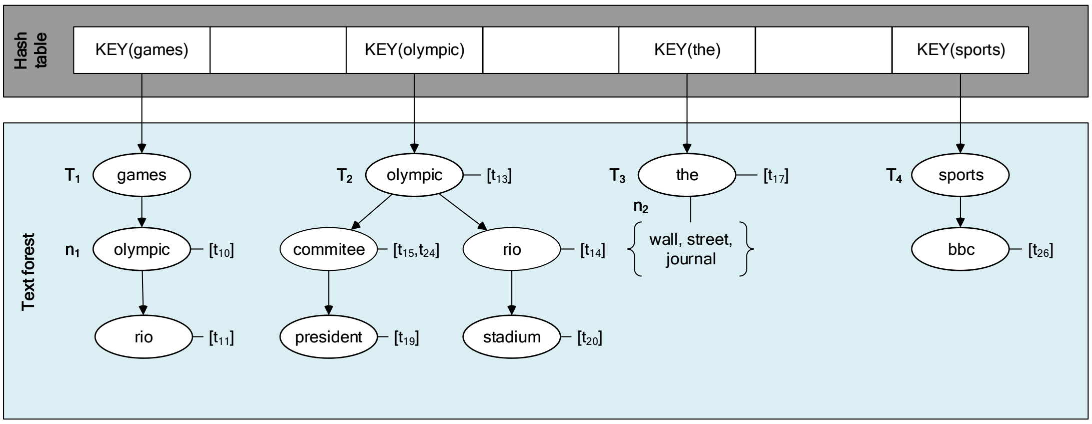


Figure 3.8: The data structure text forest after the indexing phase of RTFutilizing FT-Index.

this task, FT-Index 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, FT-Index 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 staring 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 T is considered as a poor candidate for q, whereas when $nr(q,T)$ is close to 1 T is considered as a good candidate.

Example 14 *Let us consider the text constraints of triples and their organisation illustrated in Figure 3.8. We have $nr(t_{10}, T_1) = \frac{2}{2}$, since the 2 keywords of t_{10} are both stored in a path starting from the root of T_1 and also used in a different profile (i.e., t_{11}). We also have $nr(t_{11}, T_1) = \frac{2}{3}$, since only 2 keywords of t_{11} (out of 3) are stored in a path starting form the root of T_1 and also used in a different profile (i.e., t_{10}), as keyword* rio *is used solely for t_{11}. Similarly, $nr(t_{13}, T_2) = \frac{1}{1}$, $nr(t_{14}, T_2) = \frac{2}{2}$, $nr(t_{15}, T_2) = \frac{2}{2}$, $nr(t_{17}, T_3) = \frac{1}{3}$, $nr(t_{19}, T_2) = \frac{2}{3}$, $nr(t_{20}, T_2) = \frac{2}{3}$, $nr(q_{24}, T_2) = \frac{2}{2}$ and $nr(q_{26}, T_4) = \frac{0}{2}$.*

The algorithm for inserting a new profile proceeds as follows. The first profile 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. The second profile will consider being stored at the existing trie or create a new trie. In general, to insert a new profile q, FT-Index iterates through its keywords and utilises the hash table to find all *candidate tries*; i.e., tries having a root storing a keyword of q. To compactly store q, FT-Index then chooses the trie T among the candidates for which if q was inserted $nr(q, T)$ would be maximised. To compute $nr(q, T)$, FT-Index performs a depth-limited search with depth limit $|q| - 1$ in *all* candidate tries. This search results in the node n in T where q should be inserted. Note that the chosen path from the root to n is the longest path in T that exclusively contains keywords of q. Also, if more than one tries maximising nr are found, FT-Index randomly chooses one.

To complete insertion, the path from the root of trie T to node n, that already stores the identifier of a profile p and the set of keywords K, is then extended with new child nodes having as keywords the intersection of $uexp(n, p)$ and $q \setminus K$. 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, *(b)* node m becomes a new leaf in trie T, *(c)* $id(m)$ will contain the reference to profile p (previously stored in $id(n)$) plus a reference to q, and *(d)* reference to p is removed from $id(n)$. In this way, list $uexp(n, p)$ is fully expanded to trie nodes, profile q is indexed in this subtrie under all its keywords, and node m now indexes two profile 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 $uexp(m, q)$. 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 $K \cup uexp(n, p)$. 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 complexity of inserting a profile in the forest is *linear* in the size of the forest.

Example 15 *Figure 3.8 shows the forest of tries created when inserting the text parts of triples t_{10}, t_{11}, t_{13}, t_{14}, t_{15}, t_{17}, t_{19}, t_{20}, t_{24} and t_{26} (shown in Table 3.3) in that order. The first triple t_{10} creates trie T_1 and is indexed under the (randomly chosen) keyword* games*. The second profile t_{11} does not create a new trie but is indexed under T_1, since this maximises its node reusability $nr(t_{11}, T_1)$. The third profile t_{13} cannot be indexed in T_1, since it does not contain the keyword* games*, and thus a new trie T_2 is created and t_{13} is indexed under the keyword* olympic*. Similarly,* FT-Index *inserts the remaining seven text parts of triples.*

Figure 3.9 presents the pseudocode for the indexing procedure that FT-Index implements.

Algorithm Full-text reorganize

The second text indexing algorithm that is going to be used is FT-Reorg, an algorithm presented in [72]. FT-Reorg is a derivative of FT-Index that aims to reorganize badly clustered text queries. FT-Reorg monitors the number of poorly clustered queries in the system (i.e., queries with only a few words clustered in the trie). The reorganization process is triggered when a certain threshold Q of poorly clustered queries is reached. During the reorganization phase FT-Reorg examines all text queries q that their node reusability is lower than the given threshold nr_{min}. If the currently examined profile q_c has a node reusability ratio $nr(q_c, T_c) < nr_{min}$ the algorithm tries to reposition q_c under a new Trie T_n. The new Trie T_n must provide a higher nr than the current one thus $nr(q_c, T_c) < nr(q_c, T_n)$.

FT-Reorg operates on the same data structures as FT-Index, which where described above. For FT-Reorg to facilitate it's reorganization capabilities uses an additional table NR where it stores a reference to every text profile in the *text forest* with it's current $nr(q, T)$. The algorithm maintains updated the table NR in all cases. More specially when a new profile is indexed by the forest FT-Reorg creates an new entry into the NR. While when a list *uexp* of q is expanded in order to index a new profile all queries that involve in the indexing are updated accordingly in NR.

		Textual indexing algorithm	
		FT-Index	FT-Reorg
Text forests	One	RTFi-O	RTFr-O
	Multiple	RTFi-M	RTFr-M

Table 3.4: Variations for Algorithm RTF.

As described above the full-text indexing can be handled by two different text indexing algorithms. Resulting to two variations of RTF, namely RTFi and RTFr. Algorithm RTFi utilizes the textual indexing algorithm FT-Index, and algorithm RTFr for the textual indexing algorithm. An other approach to indexing the textual information is either dedicate one text indexing forest for the indexing of every textual constraint, or utilize a text indexing forest for every unique attribute that contains a full-text operator. Thus creating four variations of RTF, algorithms RTFi-O and RTFi-M that utilize FT-Index with one and multiple textual indexing forests accordingly. Algorithm RTFr-O and RTFr-M that utilize FT-Reorg with one and multiple textual indexing forests accordingly. This four approaches will be examined in detail in Chapter 4. The four variations are presented in Table 3.4.

Figure 3.10 presents an overview of all the data structures of algorithm RTFi-O after indexing the profiles of Table 3.3.

Algorithm index text

```
BEGIN
currentNR ← 0
position ← NULL

foreach trie $T$ with $root(T) = k \in q$ do                ▷ For all candidate tries

  foreach node $n \in T$ such as $kwd(n) \in q$              ▷ For all possible storage positions in examining tries perform a DFS


        calculate ($nr(q, T)$)
        if $currentNR < nr(q, T)$ then                        ▷ If a better position is found store it

          $currentNR \leftarrow nr(q, T)$
          $position \leftarrow n$
        end if
  end for
end for

if $position = NULL$ then                                     ▷ If $q$ cannot be indexed in any existing trie

  create trie $T'$ with $root(T')$ such as $kwrd(T') \in q$
  $id(root(T')) \leftarrow q$                                 ▷ Index $q$ in $root(T')$
  $uexp(T', q) \leftarrow q \setminus kwrd(T')$               ▷ Put the rest in $uexp(T', q)$
else
  if $uexp(position, p) \cap q = \emptyset$ then             ▷ If there are not common keywords

    $id(position) \leftarrow id(position) \cup q$            ▷ Index $q$ in $position$
    $uexp(position, q) \leftarrow q \setminus \{k_0, ..., k_y\}$   ▷ Put the rest in $uexp(position, q)$

  else
    expand $uexp(position, p) \cap q$                        ▷ Else expand the common keywords

    $id(m) \leftarrow q \cup p$                              ▷ Index $q$ and $p$ at the leaf node

    $id(position) \leftarrow id(position) \setminus p$       ▷ Remove $p$ from $id(position)$
    $uexp(m, q) \leftarrow q \setminus \{k_0, ..., k_x\}$    ▷ Put the rest in two new uexp lists

    $uexp(m, p) \leftarrow p \setminus \{k_0, ..., k_x\}$
  end if
end if
END
```

Figure 3.9: Pseudocode for text indexing.

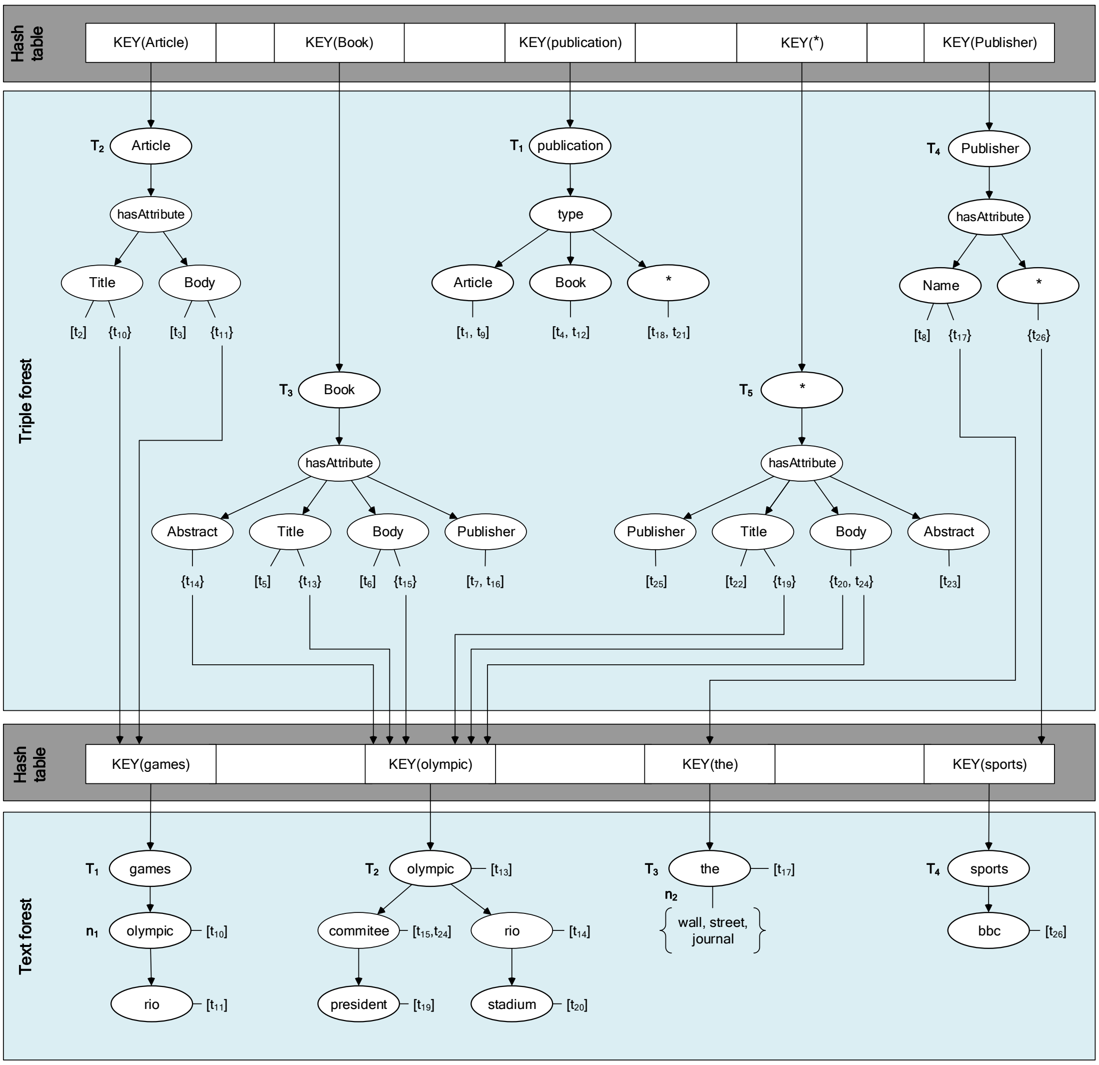


Figure 3.10: The data structures triple and text after the indexing phase of RTFi-O.

3.2.3 Matching algorithm

In Sections 3.2.1 and 3.2.2 we presented how RTF indexes RDF-triples and quadruples along with their full-text restrictions. We now proceed to demonstrate how our algorithm filters the incoming RDF publications during the filtering phase.

When an RDF publication *pub* arrives in the system the filtering mechanism is initialized as follows. The RDF publication is parsed and every triple $pt_1, ..., pt_i$ that represents *pub* is returned accompanied with it's text field if this is available. After the completion of parsing *pub*, every triple and quadruple $pt_1, ..., pt_i$ that represents *pub* is processed independently by the filtering algorithm.

At first for every triple pt_c, RTF examines the *triple forest* and tries to locate a tree T with a root i.e., having $atrb(root(T)) = subject(dt_c)$ with the same field as pt_c's subject . If such tree T is located, RTF begins traversing T in a depth first manner and examines all predicate and object nodes of T in order to determine if there is a matching subscription triple. In order to reach from the root of T to a leaf node every node n_s, n_p and n_o in the route must fulfil the following requirements: $atrb(n_s) = subject(pt_c)$, $atrb(n_p) = predicate(pt_c)$ and $atrb(n_o) = object(pt_c)$. At any point of the node traversal if a node n is visited with a wildcard field i.e. $atrb(n) = *$ the filtering algorithm considers the node to satisfy the current field of pt_c and continues to examine the next field of pt_c by iterating through the child list $chld(n)$ of node n. At the arrival of RTF in a leaf node n_l the algorithm has matched all three field of pt_c with a tree T. Subsequently the Algorithm RTF examines the two lists that the leaf node maintains, i.e. $tripIds(n_l)$ and $quadIds(n_l)$. The triples identifiers that are located in list $tripIds(n_l)$ are recognized as a direct match and are passed into a hash table named $matchedTriplets$. While the quadruples' identifiers that are located in list $quadIds(n_l)$ are triples that contain a full-text operator. The list $quadIds(n_l)$ should be further investigated in order to determine if the text field of pt_c fulfils their set criteria.

Having matched the structure of the incoming publication RTF proceeds to examine the text field of the publication triple pt_c by visiting the *text forest*. When a text field is present the filtering procedure for Algorithm RTF proceeds as follows. For each *distinct* keyword k_j of $text(pt_c)$ (maintained in a linked list created at the preprocessing step of pt_c), the trie of Text Forest that has keyword k_j as root is traversed in a depth-first manner. Notice that only subtrees having as root the keyword k_j contained in the text field of pt_c are examined (since only these may contain potentially matching queries), and a hash table (also created at the preprocessing step of pt_c) that indexes all distinct keywords of $text(pt_c)$ 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 text field $text(pt_c)$. To identify all qualifying queries, this procedure is repeated for all the keywords of $text(pt_c)$. The resulting triples identifiers are returned in a list named $matchedTextIds$. The Algorithm RTF must determine which $quadIds(n_l)$ have met their full-text restrictions. This is achieved by calculating the intersection of the lists $quadIds(n_l)$ and $matchedTextIds$ i.e., $quadIds(n_l) \cap matchedTextIds$. The resulting triple identifiers from the intersection are considered a positive match as both their RDF restrictions and full-text requirements are met by pt_c. The intersection results are also added to the hash table $matchedTriplets$.

The filtering process described above is repeated by Algorithm RTF for all triples $pt_1, ..., pt_i$ of *pub* and the matched triples are stored in the common hash table $matchedTriplets$.

When the filtering of all triples is completed RTF proceeds by examining the *Profile Table*. The Profile Table as discussed earlier is a table that represents all queries $q_1, ..., q_i$ and their triples that form them. All triples $t_1, ..., t_i$ that form a profile q_j must match with a publication *pub* in order to identify q_j as a match. The algorithm RTF iterates through every cell of the Profile Table that represents a profile q_j and examines it. For every q_j the list of triples $t_1, ..., t_i$ that represent are in turn iterated. The hash table *matchedTriplets* is used for faster access to the matched triples identifiers as calculated above during the filtering of the triple forest and text forest. If every $t_1, ..., t_i$ of q_j is detected in *matchedTriplets* then q_j is satisfied by *pub*. During the examination of $t_1, ..., t_i$ if one triple identifier is not located into the table *matchedTriplets* RTF stops the iteration of $t_1, ..., t_i$ as

q_i will not match *pub* and proceeds to examine the next profile q_{j+1} of the Profile Table. Finally all users that have a subscription that matched with p are notified accordingly.

Figure 3.11 presents the pseudocode for filtering RDF publications, while for space consideration Figure 3.12 presents the pseudocode for filtering the text parts of an RDF publication.

Algorithm filter

```
BEGIN
$matchedTriplets \leftarrow$ Null

foreach triplet $pt_i$ in publication $p$ do

    foreach tree $T$ with $atrb(root(T)) = subject(pt_i)$ || $atrb(root(T)) == *$ do

      foreach node $n \in T$ do
        if $\exists n_p \in T$ such as $atrb(n_p) = predicate(n_p)$ $\wedge$ $\exists n_o \in T$ such as $atrb(n_o) = object(n_p)$ then
        $matchedTriplets \leftarrow matchedTriplets \cup tripIds(n_o)$
        $matchedTextIds \leftarrow text_filtering(text(pt_i))$
        $matchedTriplets \leftarrow matchedTriplets \cup (quadIds(n_o) \cap matchedTextIds)$
        end if
      end for
    end for
end for

foreach  query $q_i$ in Profile Table do
  $matched \leftarrow$ TRUE
  foreach triplet identifier $t_i$ of $q_i$ do
    if $t_i \notin matchedTriplets$ do
      $matched \leftarrow$ FALSE
      break
    end if
  end for

  if $matched =$ TRUE do
    notify
   end if
end for
END
```

Figure 3.11: Pseudocode for publication filtering.

Algorithm filter text

```
BEGIN
$match \leftarrow$ Null

foreach distinct keyword $k \in d$ do                    ▷ Use a linked list for
                                                           distinct keywords of d
  foreach trie $T$ with $root(T) = k \in d$ do

    foreach node $n \in T$ do

      if $kwrd(n) \in d$ then                              ▷ Use a hash table
                                                           representation of $d$ to check
                                                           this
        if $uexp(n, q) \subseteq d$ then
          $match \leftarrow match \cup id(n)$              ▷ The queries stored here
                                                           match $d$
          $n \leftarrow children(n)$                       ▷ Traverse trie in DFS
        end if
      else
        prune $n$                                          ▷ Else no need to search
                                                           in sub-tries
      end if
    end for
  end for
end for

return match

END
```

Figure 3.12: Pseudocode for text filtering.

3.2.4 Competitor

To evaluate the efficiency of RTF we have chosen to implement the only state-of-the-art algorithm iBroker [57] that is capable of handling RDF profiles with both structure and text. Algorithm iBroker is a broker for publish/subscribe systems that makes use of SPARQL queries subscription with phrase support. In this section we give the basic functionality of iBroker and data structures upon which it operates and show hot to extend iBroker's functionality to support full-text constraints.

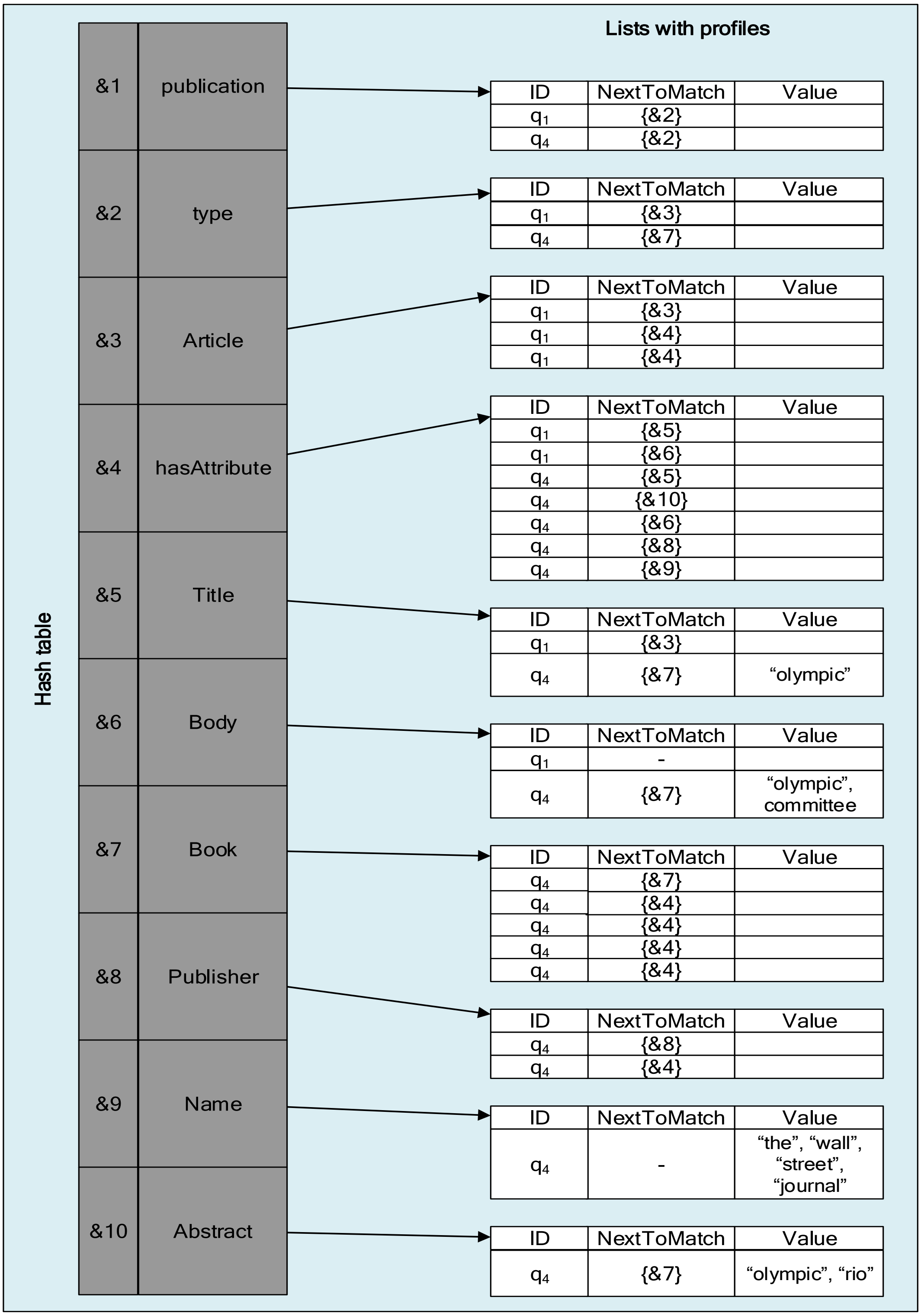
Algorithm iBroker utilizes a two-level hash table in order to index the users subscriptions into the system. The indexing structure that is used consists at the first level by a hash table. The hash table indexes the unique fields of all triples that are present into the system. iBroker uses the unique fields' names as hash keys to access the cells of the table, while in the cells of the hash table references to lists of users profiles are stored. The lists in the second level of the indexing structure maintain three fields of information: the first entry of the lists maintains the unique identifier ID of the profile q, the second entry, named $NextToMatch$, is a reference to an entry of the first-level of the two-level hash, that contain the next triple field in a chain that forms the profile q, and the last field of the list is parameter $Value$ that stores the text keywords the user has set.

The algorithm iBroker uses the two-level hash table in order to store the queries in a chain-like manner. Every profile is formed by following the $NextToMatch$ references to the first level of the hash table 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 match any of it's fields in the first level of the hash table. The result is that iBroker must examine all the profiles in the corresponding list of the first match. After iBroker examines all the profiles, it will proceed to examine their $NextToMatch$ entries and likewise the algorithm will continue to examine every $NextToMatch$ entry until there is none left. 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 profile p can not be determined as positive, resulting to false positives for iBroker. This problem can be solved by utilizing the Profile Table as described in a previous section. As the RDF triples that form a profile p are assigned different identifiers and stored into the Profile Table, thus creating a guided chain of match for every triple identifier which must match in order to satisfy a profile. This approach eliminates the majority of false positives but extreme cases where the first triple of a profile matches partially can not be solved efficiently.

iBroker implements a text value matching system for filtering the text fields of the publications on the system. The model that iBroker supports for the indexing and filtering is *text phrase matching* but not *full-text filtering*. In order to evaluate iBroker against RTF we have extended the ability of iBroker to index full-text subscriptions. This was achieved by replacing the $Value$ field with a list of words. The implemented list of words may support both phrase and full-text matching as described in Section 3.2.2. Finally we have extended the functionality of iBroker to index and filter SPARQL queries that contain wild-card operators.

Example 16 *Consider the queries q_1 and q_4 of Table 3.3. The result of indexing those queries in* iBroker*'s data structures is presented in Figure 3.13. Due to space consideration we don't present the rest of the queries, as done earlier for* RTF*. Additionally the Profile Table presented in Figure 3.5 is the same for both algorithms.*

In this chapter we have presented the Algorithm RTF, an algorithm developed to index SPARQL-based subscriptions extended with full-text operators. Subsequently we have given the data structures and methodologies for indexing RDF-based profiles along with their text constraints. Finally we outlined our competitor, the state-of-the-art Algorithm iBroker, and described how we modified and extended it to support the profile language described in Section 3.1.1. In the next chapter we proceed by presenting the evaluation of the Algorithm RTF with a diversity of experiments.



ID	NextToMatch	Value
q_1	{&2}	
q_4	{&2}	

ID	NextToMatch	Value
q_1	{&3}	
q_4	{&7}	

ID	NextToMatch	Value
q_1	{&3}	
q_1	{&4}	
q_1	{&4}	

ID	NextToMatch	Value
q_1	{&5}	
q_1	{&6}	
q_4	{&5}	
q_4	{&10}	
q_4	{&6}	
q_4	{&8}	
q_4	{&9}	

ID	NextToMatch	Value
q_1	{&3}	
q_4	{&7}	“olympic”

ID	NextToMatch	Value
q_1	-	
q_4	{&7}	“olympic”, committee

ID	NextToMatch	Value
q_4	{&7}	
q_4	{&4}	
q_4	{&4}	
q_4	{&4}	
q_4	{&4}	

ID	NextToMatch	Value
q_4	{&8}	
q_4	{&4}	

ID	NextToMatch	Value
q_4	-	“the”, “wall”, “street”, “journal”

ID	NextToMatch	Value
q_4	{&7}	“olympic”, “rio”

Figure 3.13: The data structure two-level hash table after the indexing phase of iBroker.

Chapter 4

Experimental evaluation

In this chapter we discuss the experimental evaluation of the algorithms presented in Chapter 3.2. At first we demonstrate the data, profile and publication sets used in our evaluation process. We proceed outlining the configuration parameters selected for the algorithms set-up. Thereafter we give the metrics used in our evaluation and the technical configuration of our algorithms and experiments implementation. Finally we present and analyse the results obtained from the process of evaluating RTF and iBroker.

4.1 Data and profile set

In order to evaluate the developed algorithms we used the set of publications DBpedia[1]. The DBpedia corpus is a collection of data that contains *structured* information. The information that form the corpus of DBpedia are extracted from the Wikipedia domain. The informations gathered in DBpedia concern the English part of the Wikipedia domain and form a knowledge database that describes 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 different properties. It is understood that the data collected in the DBpedia collection cover a wide variety of topics. The diversity in

[1] http://www.dbpedia.org

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 4.1: Characteristics of the DBpedia corpus.

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 utilize DBpedias' textual information. The majority of the publications in the DBpedia corpus carry both structural and textual information. The textual information of every publication are representations of human generated content and published at the Wikipedia domain. This characteristics gives an additional advantage to the DBpedia corpus as it provides a solid set 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 information available in a publication is 53 words.

The diversity in the content of the DBpedia corpus accompanied with the information on structural and textual level, renders it as the perfect candidate for evaluating our algorithms indexing and filtering efficiency. Table 4.1 summarises some key characteristics of the DBpedia corpus.

4.1.1 Profile set

Since we are examining an information filtering scenario it is important to investigate the behaviour of the algorithms when varying the profile data set. As no database of real-life profiles was available except by obtaining proprietary data (e.g., by a news alerting system) we designed and simulated three different experimental setups. The aim of these profile sets is to evaluate the performance of our algorithms under three different filtering scenarios. By capturing and studying the most common scenarios that can emerge in the publish/subscribe paradigm, we provide a comprehensive understanding of the performance that the algorithms can achieve.

The main profile collection The main collection of profiles that is going to be used in our evaluation is formed by conjunctions of RDF-triples and quadruplets baring full-text operators. The RDF-triples were constructed by utilizing the 529 classes and $2,333$ properties that were extracted by the Wikipedia domain and are available in the DBpedia corpus. By utilizing structural and textual data from the DBpedia corpus we can simulate the profile creation with great precision to real RDF-based profiles. Each profile that was generated for the main profile collection may contain at most 4 RDF-triples. Each RDF-triple, in a profile, can contain a full-text operator with a probability of 50%. The full-text operators contain conjunctive terms that are selected equiprobably among the multi-set of words that form the DBpedia corpus textual vocabulary. While the full-text operators do not contain more than 3 textual terms. The main profile collection aims at evaluating the algorithms under an average filtering scenario where the database is filled with profiles consisting more from structural restrictions than textual. A subscription usually bears more structural constraints in order to select a specific type of publication with specific properties and classes while the textual constraints refine the publication selection and using a small amount of keywords the thematic matching can be achieved.

The second profile collection The second profile collection is formed by conjunctions of RDF-triples baring no full-text operators. The lack of full-text operators aims to studying the structural filtering of the algorithms. Omitting to include

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

Table 4.2: Parameters' description and baseline values.

full-text operators into the profiles will stress-test the algorithms and focus on the part of structural filtering. The presence of full-text operators into the main profile collection restricts the matching of a major percentage of the RDF-triples. In this filtering scenario the RDF-triples can match at a higher percentage with an incoming publication thus giving a closer look at the structural matching capabilities of the algorithms. The algorithms that can leverage their filtering capabilities at any type of database should be able to perform equally or better compared to the main profile collection.

The third profile collection The third profile collection aims to stress-test the algorithms and focus on the textual part filtering. While all RDF-triples contain full-text operators the algorithms that we evaluate will demonstrate their scaling capabilities when they index databases that contain a high percentage of full-text restrictions.

4.1.2 Publication set

In order to evaluate the three data sets described above, we selected 5000 publications from the DBpedia corpus. The publications selected had both their structural and textual information as extracted and processed from the Wikipedia domain. The selected publications are used in every filtering event during the algorithm evaluation. By maintaining the same publication through the evaluation of our algorithms we assert that the algorithms are evaluated against the nature of the profiles that are inserted for indexing. While the publications represent human generated, real-life, data giving an accurate predication of how the algorithms perform.

4.2 Metrics employed

In our evaluation, we present, compare and discuss a series of metrics in order to determine and understand the algorithms performance. We present and compare the memory requirements of each algorithm under the varying data sets. As all algorithms index the same profile databases, a lower memory requirement indicates a more compact clustering of data while a higher memory footprint a less compact database. We give the insertion time of each algorithm i.e., the amount of time needed to index a set of profiles into the database in order to determine which algorithm can be faster at the indexing phase. Additionally, we demonstrate the filtering time to measure the filtering performance of each algorithm. i.e., the amount of time needed to locate all continues profiles satisfied by a publication. We juxtapose the insertion and filtering time results and discuss which metric presents a more clear picture of the algorithms performance when it is needed. Finally we present the algorithms differences in both insertion and filtering times and demonstrate how they differentiated during the size increase of the profile database.

Table [Reference] summarises the parameters examined in our experimental evaluation with their baseline values.

4.3 Technical configuration

All the algorithms shown in the experiments of this section were implemented in C++, and an of-the-shelf PC with a Core $i7$ $3.6GHz$ processor and $8GB$ RAM running Ubuntu Linux 12.04 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.

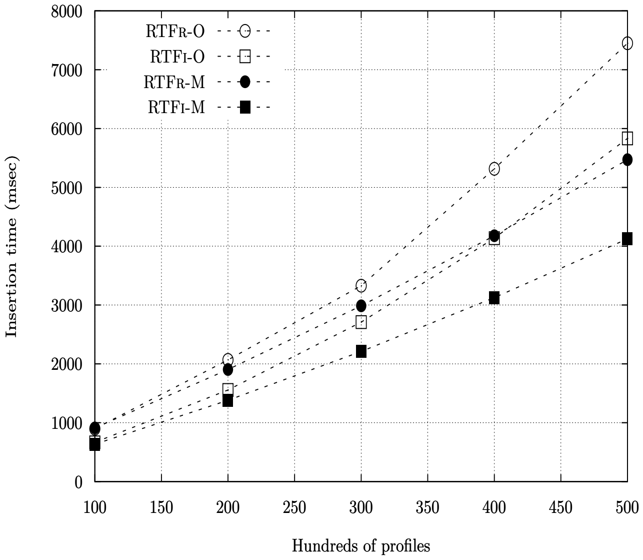
4.4 Algorithm configuration

In this section we discuss the algorithms configuration. There is a number of parameters that need to be considered in order to tune the RTF's performance. We proceed presenting the selected configuration values for RTF.

4.4.1 Text indexing configuration

As demonstrated in Chapter 3.2 RTF utilizes the usage of two algorithms for the indexing of the textual part of the full-text operators. The first algorithm described is FT-Index which indexes the terms that are present into full-text operators by using forests of tries. The algorithm FT-Index doesn't need any special configuration as it indexes the textual parts with the order they are inserted. As discussed in Chapter 3.2 FT-Index utilizes a greedy way to index the terms, thus selecting a node in a forest of tries where the node re usability is maximized.

The second algorithm that is described is a variation of FT-Index namely FT-Reorg which utilizes a reorganization mechanism in order to maximize the trie compactness. FT-Reorg tries to overcome the problem of insertion order and aims to reposition the textual parts of profiles in a better position when their node re usability is at a low percent. FT-Reorg exploits newly created position into the forest of tries that were not available during the indexing phase of a specific textual part. The reorganization mechanism is triggered after a specific number of queries has been indexed into the forest. When this number of text parts is reached the reorganization process is triggered as some clustering opportunities are likely to have arisen. While the text-parts are reorganized only after they fall under a predetermined threshold of node re-usability. After carefully evaluating FT-Reorg we concluded that the reorganization process must be triggered after 500 thousands new text parts have been indexed, this guaranties us that the new positions in the forest are created. While the threshold that a text part must has fallen under is determined at 0.8. The above parameters ensure that the trie forest has an optimal node re-usability level that can be achieved given these sets of words.



Parameter	Value
I_p	$100K$
I_{pub}	–
DB	$100K - 500K$
FT_{pr}	50%

Figure 4.1: Insertion time of the variations of RTF.

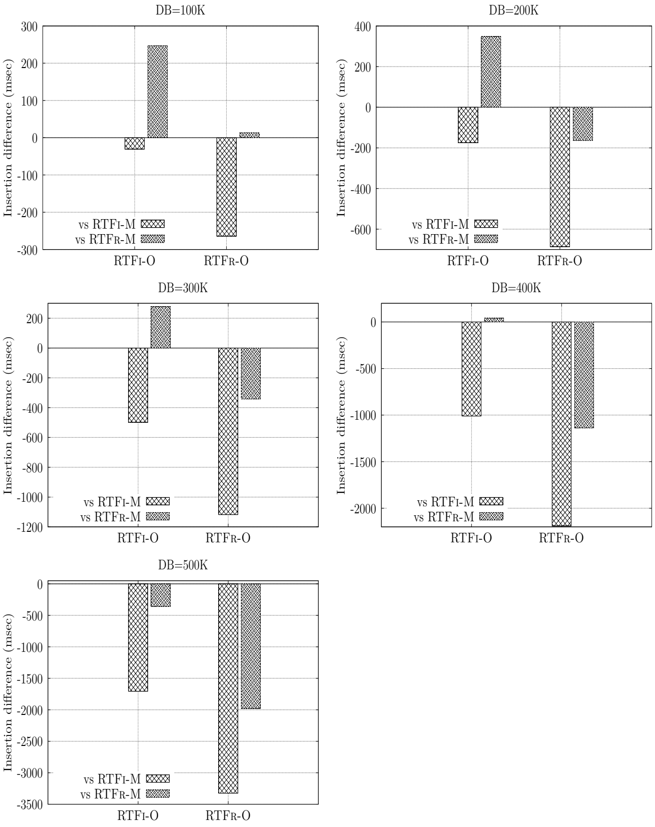
4.4.2 RTF configuration

As discussed in Chapter 3.2 RTF can uses a single or multiple forests of tries in order to facilitate the indexing of the text part of the profiles. We have conducted experiments in order to determine the best set-up for RTFṪhe evaluation of the variations of RTF is conducted with the main profile collection as it is the closest to a real-life scenario of a profile database. As the RDF-part indexing is the same for all RTF's variations, the main purpose of this comparison is to determine the the performance of the text indexing and text filtering of the algorithms.

Comparing indexing time

Figure 4.1 presents the time that the four variations of RTF consume in order to insert $I_p = 100K$ new profiles in a database of varying sizes. The insertion times represent both the indexing phase of the structural restrictions of the profiles as well as the indexing of textual restrictions in the corresponding structures. While the measurements for the RTFR-M and RTFR-O include the reorganization time the algorithms use in order to reposition the text parts of the profiles in better positions. We observe that the insertion time of all algorithms increases as the size of the database increases. This is a logical consequence of the manner that the four variations of RTF operate during the indexing phase. As the database size increases RTFr-M, RTFr-O, RTFi-O and RTFi-Mmust consider more positions to index the structural and textual parts of the profiles. The algorithms RTFr-O and RTFi-O perform slower during the indexing phase while the database size increases. RTFr-O and RTFi-O consume more time as they maintain a single text indexing structure that offers more clustering opportunities, thus a high percentage of the time is spend looking for the position that maximizes the node re-usability. RTFr-O is the slowest of both as it utilizes textual reorganization techniques thus spending more time repositioning badly clustered text parts. On the other hand RTFi-Odoes not implement any reorganization technique, which gives it a lower insertion time compared to RTFr-O. Algorithms RTFr-M and RTFi-M spend less time in the insertion as they employ the usage of multiple textual indexing forest. RTFr-M is slower compared to RTFi-M as the maintenance of multiple forests forces algorithm RTFr-M to explore a higher number of possible positions to index the text parts of a profile. While the reorganization for multiple forests consumes a high percent of the time spent in the indexing phase. RTFi-M utilizes no reorganization technique, thus spending less time in the indexing phase while the available positions for indexing a text part of a profile are reduced due to the existence of multiple text indexing opportunities speeding up the insertion procedure.

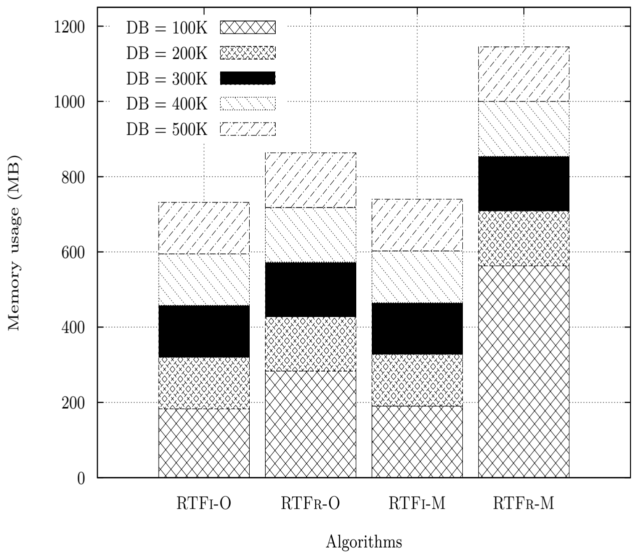
Figure 4.2 presents the differences in insertion times of the four variations of the algorithm RTF in absolute numbers. The presented difference concern the variations that use one text indexing forest against the variations that use multiple text forests, thus RTFi-O and RTFr-O are compared against RTFi-M and RTFr-M. In more detail, algorithm RTFi-M spends less time to index an input of profiles $I_p = 100K$ to an empty database compared to RTFi-O. Namely RTFi-M spends 4.6% less time than RTFi-O and it's lead is maintained, while their difference increases to 29.25% for every new $I_p = 100K$ profiles and the database size increases from $DB = 400K$ to $DB = 500K$. Algorithms RTFi-M also spends less time in the indexing phase compared to RTFr-O. Their difference when filling an empty



Parameter	Value
I_p	$100K$
I_{pub}	–
DB	$100K - 500K$
FT_{pr}	50%

Figure 4.2: Insertion time difference between the variations of RTF.

database with $I_p = 100K$, thus going from $DB = 0K$ to $DB = 100K$, is at 29.4% percent. While the lead of RTFi-M is maintained through out the indexing phase against RTFr-O, it reaches 44.62% as the database size increases from $DB = 400K$ to $DB = 500K$. This behaviour can be explained as follows: As the text forests of RTFi-M increase in size better clustering opportunities arise for every text part of a profile thus RTFi-M places the candidate text parts into better indexing positions into it's forests. On the other hand we can see that RTFr-M needs more indexing time to index the first $I_p = 100K$, thus going from $DB = 0K$ to $DB = 100K$, compared to it's competitors. Namely RTFr-M needs 37.1% more time to index $I_p = 100K$ to an empty database compared to RTFi-O. The difference decreases as the database size increases, thus when the database increase from $DB = 400K$ to $DB = 500K$ and $100K$ new profiles are inserted RTFr-M consumes 6.17% less time compared to RTFi-O. The same behaviour we can observe comparing RTFr-M to



Parameter	Value
I_p	$100K$
I_{pub}	–
DB	$100K - 500K$
FT_{pr}	50%

Figure 4.3: Memory consumption of the variations of RTF.

RTFr-O. The starting difference in insertion time is at 37.12%, when the database size increases form $DB = 0K$ to $DB = 100K$. While the difference of RTFr-M against RTFr-O reverses, thus RTFr-M needs 6.17% less time compared to RTFi-O when the database increases from $DB = 400K$ to $DB = 500K$ and $100K$ new profiles are indexed. This behaviour is attributed to the following: As the textual database increases in size the reorganization procedure is triggered multiple times to reposition and cluster the textual parts into better positions. The reorganization process by clustering similar text parts together reduces the available positions into the text forest thus reducing the search time during the indexing phase of RTFr-M.

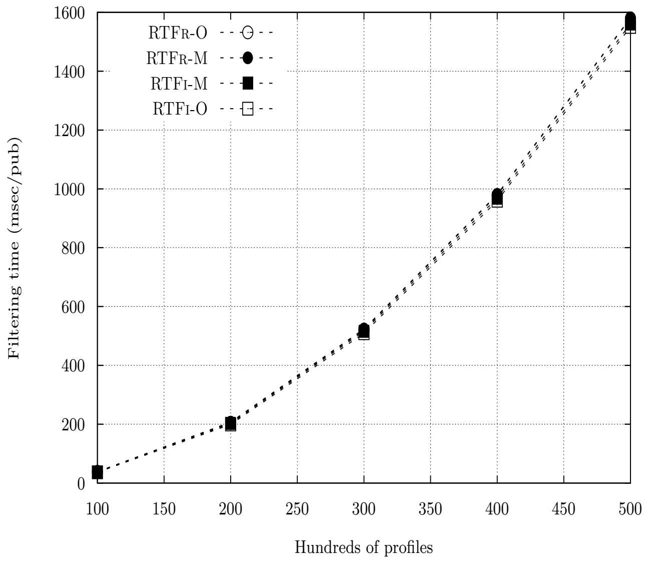
Comparing memory requirements

Figure 4.3 presents the memory requirements of the four variations of RTF. We can observe the memory that the algorithms RTFi-O, RTFr-O, RTFi-Mand RTFr-M use in every step of the insertion process for a database that starts empty ($DB = 0K$) and finally indexed $500K$ profiles ($DB = 500K$). We can observe that all algorithms increase their memory usage as the database size increases. The algorithms that utilize a reorganization technique for the textual part occupy more space in the main memory compared to the variants that do not utilize a reorganization technique, namely RTFr-M and RTFr-O. Algorithm RTFr-M occupies $1145MB$ when the database size $DB = 500K$ and algorithm RTFr-O uses $863MB$ from the main memory. We can observe that RTFr-M and RTFr-O occupy the majority of their memory requirements when indexing the first $100K$ of profiles. Algorithm RTFr-M reserves $563MB$ when indexing the first $100K$ databases while RTFr-O occupies $283MB$. This can be explained as follows: The algorithms during the initialization phase reserves the memory that is required for them to operate. As RTFr-M and RTFr-O utilize reorganization techniques for repositioning badly clustered text parts they keep an index of all the profiles that must be repositioned during the reorganization phase, thus requiring more memory compared to RTFi-M and RTFi-O. RTFr-M needs more memory to operate, as opposed to RTFr-O, as the maintenance of multiple forest requires more memory for the indexing and reorganization structures that are used. Comparing the rest for the memory requirements when the initialization phase has been completed we can see that RTFr-M and RTFr-O do not require more than $150MB$ to index every $I_p = 100K$ new profiles. Algorithms RTFi-M and RTFi-O also reserve the majority of their memory requirements during the indexing phase. Namely RTFi-M requires $190MB$ when indexing $I_p = 100K$ in an empty database, in the same setting RTFi-O requires $183MB$. While both RTFi-M and RTFi-O do not require more than $150MB$ to index every $I_p = 100K$ new profiles. Finally RTFi-M has bigger memory requirements when indexing a database $DB = 500K$ compared to RTFi-O, where RTFi-M uses $740MB$ and RTFi-O reserves $731MB$. Their difference can attributed to the following: Algorithm RTFi-M maintains more text indexing structures thus requiring more memory for the text indexing structures while RTFi-M maintains a single text indexing structures.

Comparing filtering time

Figure 4.4 presents the filtering time that the four variations of RTF consume in order to filter an incoming publication both structurally and textually for varying sizes of the database. The filtering time of a publication is the average time the algorithms the algorithms needed to filter the selected 5000 RDF-based publications as described in Section 4.1.2. We observe that the filtering time that all variation of RTF consume, increases as the size of the database increases. As the database size increases RTFi-M, RTFi-O, RTFr-Mand RTFr-Oneed to spend more time searching in the indexing structures for profiles that match the incoming publication. We can observe that the filtering performance is similar for all the algorithms with small differences. Algorithms RTFr-O and RTFr-M take the most time to filter an incoming publication with the database of profiles they index. Although they implement textual reorganization techniques the do not exceed the performance of RTFi-M and RTFi-O. This can be attributed to the nature of the textual parts of the profiles as they do not contain a great amount of textual information thus not maximizing the re organization capabilities of RTFr-O and RTFr-M. While the algorithms RTFi-M and RTFi-O perform better during the filtering phase. The algorithm RTFi-O has the lower filtering time per publication as it utilizes a single text indexing structure for the textual parts of the profiles. On the other hand RTFi-M by maintaining multiple indexing forests loses in performance as it must check more indexing structures for every publication in order to determine the matching profiles.

Figure 4.5 presents the differences in filtering time of all the possible configurations of algorithm RTF in absolute numbers. The presented difference study the variations that use one text indexing forest against the variations that use multiple text forests, thus RTFi-O and RTFr-O are compared against RTFi-M and RTFi-O. We can see that the differences between the algorithms are minimal. More specifically algorithm RTFi-O spends more time to filter index the database when first filled $DB = 100K$ profiles compared to RTFi-M and RTFr-M. However their differences are little, namely RTFi-O performs 2.58% faster compared to RTFi-M while RTFi-O delivers 5% faster the filtering results for a publication compared to RTFr-M. As the database size increases for every $I_p = 100K$ new profiles we can see that RTFi-O keeps performing better compared to RTFi-M and RTFr-M. Algorithm RTFi-O delivers faster matching results when the final database size is



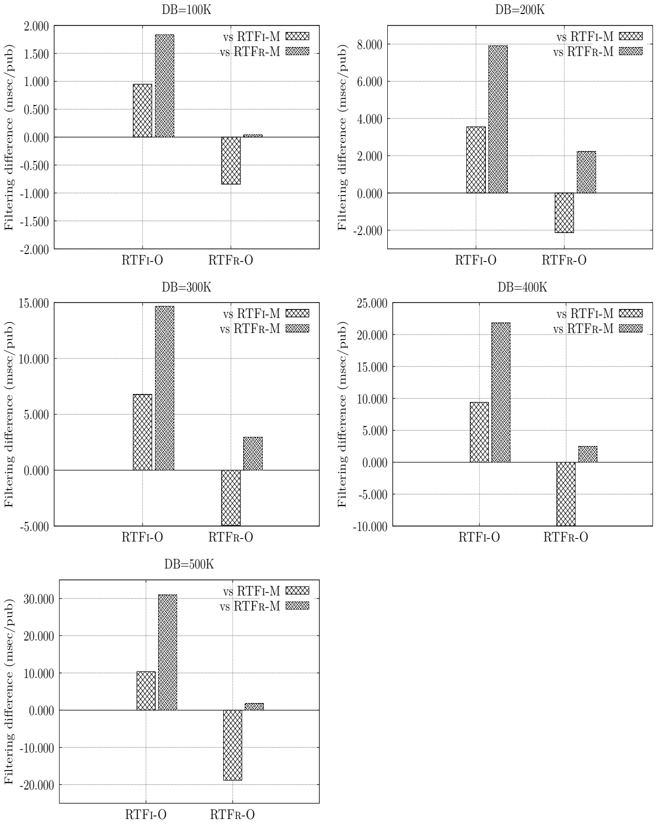
Parameter	Value
I_p	–
I_{pub}	5000
DB	$100K - 500K$
FT_{pr}	50%

Figure 4.4: Filtering time of the variations of RTF.

$DB = 500K$ and it's differences although minimal between RTFi-M and RTFr-Mare 0.66% and 1.99% accordingly. Comparing RTFr-O we can observe that it is performing higher filtering times compared to RTFi-M and RTFr-M when filtering $DB = 100K$ profiles. Namely RTFi-M filters 0.84% faster an incoming publication compared to RTFr-O. While RTFr-M filters 0.11% slower an incoming publication compared to RTFr-O. We observe that while the database size increases there are some differentiations in the filtering times when comparing RTFr-O and RTFr-M. Finally when the database size reaches $DB = 500K$ there is a slight change in the hierarchy, RTFi-M is faster by 1.19% against RTFr-O and RTFr-M is slower by 0.11% against RTFr-O.

Conclusions

We studied the four possible variations of RTF, presented insertion and filtering times as well as the memory requirements of all the algorithms. As we study a publish/subscribe system we focus on the filtering performance of the algorithms, the filtering of incoming publications is resource intensive. Profile indexing can be handled by the server at any given time, a server can implement buffering techniques



Parameter	Value
I_p	–
I_{pub}	5000
DB	$100K - 500K$
FT_{pr}	50%

Figure 4.5: Filtering time difference between the variations of RTF.

when indexing a high number of profiles, although a high number of subscriptions rarely occurs simultaneously. On the other hand publication events can flood the server at any give moment, publish/subscribe systems usually have a high amount of publishers that can publish at any given moment thus rendering the publication filtering crucial. For the above reasons we select two variations of RTF as optimal, although their filtering performance is equivalent. Although algorithms RTFR-O and RTFR-M implement textual reorganization techniques, they seem to perform slightly slower compared to algorithms RTFI-O and RTFI-M. This can be explained as follows: The profiles indexed contain at most 3 text terms in every full-text operators thus not allowing RTFR-O and RTFR-M to perform a better reorganization as the indexing opportunities are limited resulting to not differentiate from the algorithms that implement any database optimization technique. Algorithms RTFI-O and RTFI-Mare selected and in the rest of the experimental evaluations will be used comparing against iBroker.

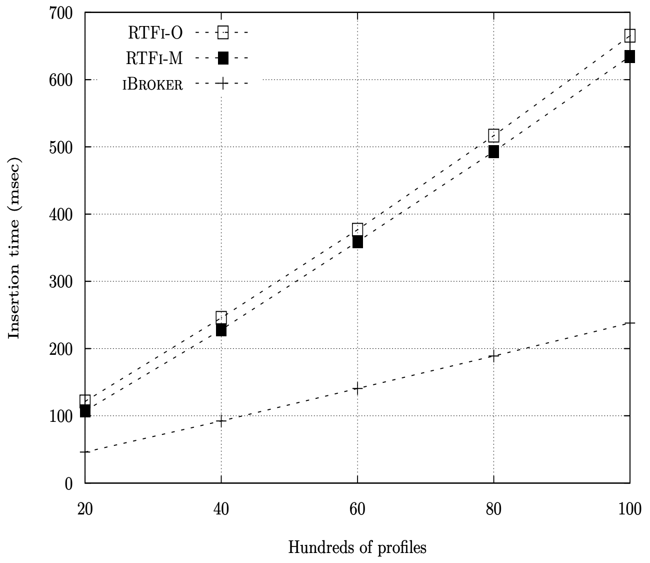
4.5 Results for the general profile collection

In this section, we present the experiment conducted in order to evaluate the variants of algorithm RTF against the algorithm iBroker as presented in the literature [57]. In more detail we evaluate algorithms RTFi-O, RTFi-M and iBroker using the general profile as described in Section 4.1.1, while we present the memory requirements, the indexing and filtering times of the algorithms. In Section 4.4.2 we presented the four variations of RTF indexing in a database of $DB = 500K$ size. Although we conducted experiments for large databases for the variations of RTF we can not present them against iBroker, where it is needed we are going to present the most important of these measurements.

Comparing insertion time

This section presents the time every algorithm spends in order to index a standard input of $I_p = 20K$ new profiles in a database with varying sizes.

Figure 4.6 presents the insertion time that algorithms RTFi-O, RTFi-M and iBroker spend to index $I_p = 20K$ new profiles into their database. We observe that the algorithms increase their time needed to index the new queries as the database size increases. The algorithms RTFi-O and RTFi-M need more time to index the same amount of profiles $I_p = 20K$ compared to iBroker. Additionally the two variations of RTF increase their time requirements faster compared to iBroker. This performance in the insertion phase can be explained as follows: The algorithms RTFi-O and RTFi-M utilize trie-based data structures in order to index the structural restrictions of the profiles while capturing the common textual restrictions of all profiles. Additionally RTFi-O and RTFi-M utilize trie-based structures to index the full-text requirements of the profiles and capture the common textual constraints that form them. The search for clustering common structural and textual restrictions into trie based structures result to slower indexing performance as opposed to iBroker. On the other hand iBroker does not utilize any clustering

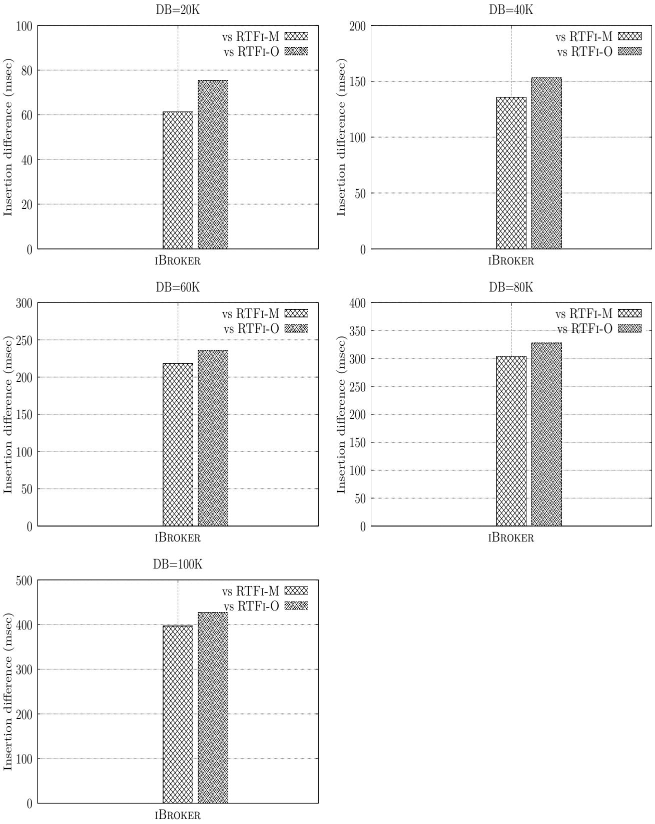


Parameter	Value
I_p	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	50%

Figure 4.6: Insertion time of algorithms iBroker and RTF.

technique during the indexing phase, thus resulting to lower insertion times.

To better demonstrate the differences in insertion increase between the proposed algorithms and their competitors Figure 4.7 summarises the absolute differences of iBroker compared against the competitor algorithms RTFi-O and RTFi-M. We present the differences in insertion time when varying the database sizes when inserting $I_p = 20K$ new profiles. We can observe that iBroker spends lees time during the indexing phase and the difference against RTFi-O and RTFi-M increases. More specifically when the algorithms index the $I_p = 20K$ to an empty database, iBroker spends 133% less time compared to RTFi-M. The differences between iBroker and RTFi-M are maintained during the increase of the database size. When iBroker indexes the final $I_p = 20K$, thus going from $DB = 80K$ to $DB = 100K$ the difference reaches 166%. Likewise iBroker is faster compared against RTFi-O, as it needs 163.7% less time filling the first $I_p = 20K$ to an empty database. When iBroker indexes the final $I_p = 20K$ it needs 179% less time.

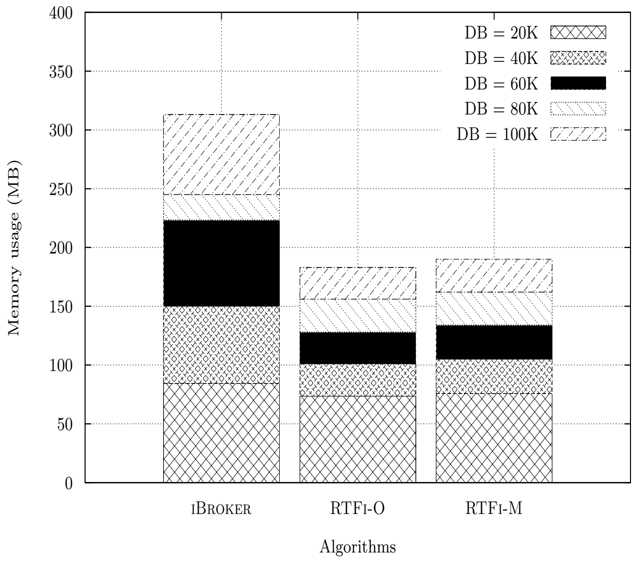


Parameter	Value
I_p	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	50%

Figure 4.7: Insertion time difference of algorithm iBroker against RTF.

Comparing memory usage

We have also executed experiments to specify the memory requirements for each of the presented algorithms. Figure 4.8 exhibits a good overview of the results for varying DB sizes. Algorithm RTFi-O has the lowest memory requirements using $183MB$ for storing the whole profile database $DB = 100K$ and all indexing components. Algorithm RTFi-M memory usage is at $190MB$ for storing the same profile database. Algorithm iBroker occupies the higher amount of memory requiring $313MB$ to index a database of $DB = 100K$ profiles. The variations of RTF have low memory requirements as the utilizes clustering techniques to capture the common elements of the profile bot on structural restrictions and textual. RTFi-O has lower memory requirements compared to RTFi-M as it utilizes a single textual indexing structure thus capturing more common elements achieving a more compact



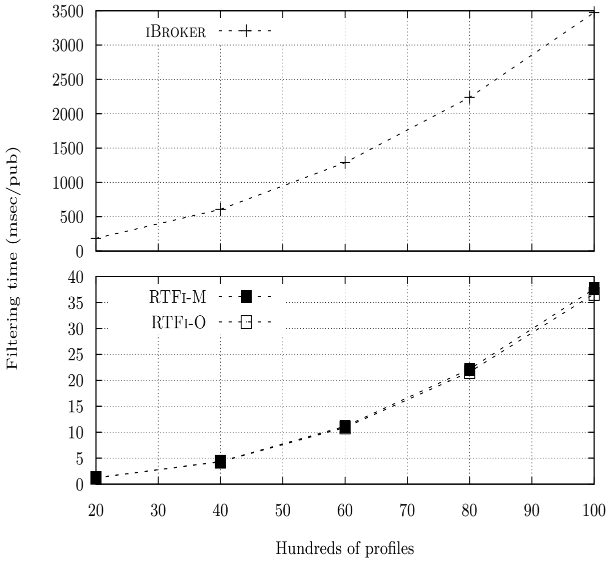
Parameter	Value
I_p	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	50%

Figure 4.8: Memory consumption of algorithms RTF and iBroker.

forest. While RTFi-M by utilizing multiple forests misses clustering opportunities requiring more memory to index the profile set. iBroker does not utilize any clustering techniques thus needing more memory to store all the elements of the profiles. Finally we observe that RTFi-O and RTFi-M reserve the majority of their memory when indexing the first $I_p = 20K$ to an empty database. This behaviour can be attributed to the initialization of the indexing structures that the algorithms use. Namely RTFi-O reserves $73MB$ when indexing the first $I_p = 20K$ profiles and RTFi-M reserves $75MB$. For every new $I_p = 20K$ inserted into the database RTFi-O and RTFi-M do not require more than $28MB$ to facilitate the indexing of the new profiles. On the other hand iBroker reserves $84MB$ of memory to index the first $I_p = 20K$ profiles into an empty database while it requires approximately more than $60MB$ of memory to index every set of $I_p = 20K$ new profiles.

Comparing filtering time

This section discusses the results concerning the filtering time required to match an incoming publication against a database of profiles. The time shown in the graphs represents the average time spend to filter a collection of $I_D = 5K$ publications.

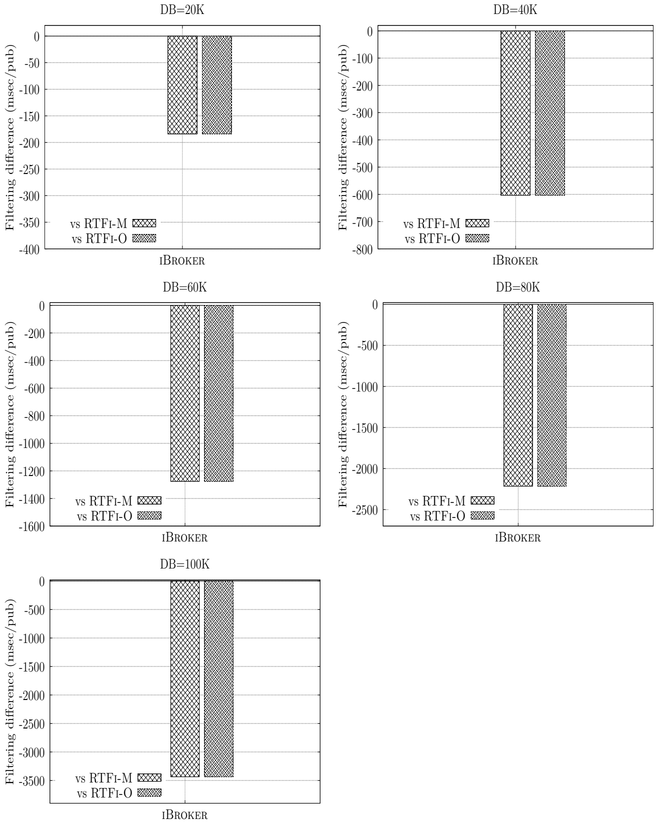


Parameter	Value
I_p	–
I_{pub}	5000
DB	$20K - 100K$
FT_{pr}	50%

Figure 4.9: Filtering time of algorithms iBroker and RTF.

Figure 4.9 shows the time in milliseconds needed to filter an incoming publication against a profile database of different sizes. Observe that filtering time increase for all algorithms as the profile database size increases. Algorithms RTFi-O and RTFi-M, that utilize clustering techniques for the storage of the profiles achieve the lowest filtering time, suggesting better performance than their competitor iBroker. Algorithms RTFi-O and RTFi-M are less sensitive to the profile database size increase compared to iBroker. RTFi-O and RTFi-M can match against a database faster an incoming publication searching the trie based forests and match publications in bulk. While iBroker searches and matched every profile independently when filtering publication.

Figure 4.10 summarises absolute differences in filtering time for iBrokeragainst it's competitors. All differences are computed for varying database sizes. Note that negative numbers in differences indicate that less time is required for RTFi-O and RTFi-M for filtering incoming publications. When algorithm RTFi-M indexes a database size of $DB = 20K$ it performs 99.3% less time compared to iBroker. Their difference is maintained for all database sizes studied and when the database size reaches $DB = 100K$ their difference is at 98.94%. When algorithm RTFi-

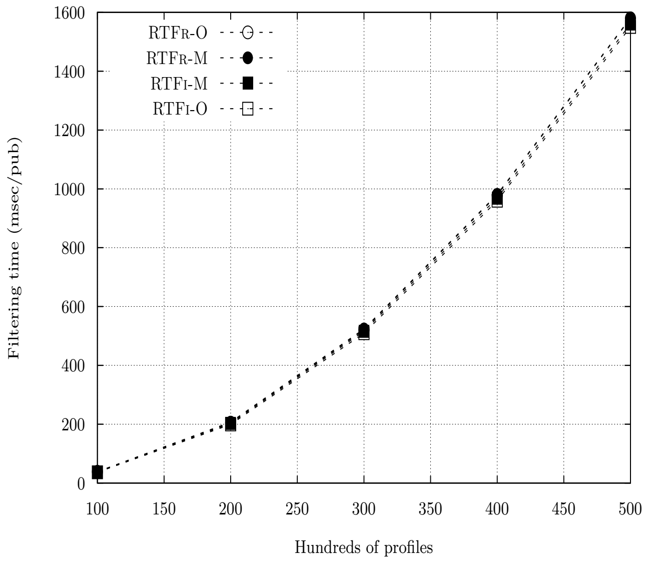


Parameter	Value
I_p	–
I_{pub}	5000
DB	$20K - 100K$
FT_{pr}	50%

Figure 4.10: Filtering time difference between RTF and iBroker.

O indexes a database size of $DB = 20K$ it performs the matching process of a publication 99.34% faster compared to iBroker. Finally when algorithms RTFi-O and iBroker index $DB = 100K$, RTFi-O is 98.91% faster.

Figure 4.11 shows the time in milliseconds needed to filter an incoming publication against a profile database of different sizes for all the variations of RTF. The results demonstrate the performance of the algorithms for database sizes greater than $DB = 100K$. We can see that the time increases for all algorithms as the profile database size increases. The algorithms exhibit a similar performance as discussed in Section 4.4.2. Their clustering and reorganization techniques gives them the ability to easily match incoming publications into large databases spending few milliseconds. The ranking of the algorithms is slightly altered during the database increase placing RTFi-O at the top of the performance scale for the maximum database size $DB = 500K$.



Parameter	Value
I_p	–
I_{pub}	5000
DB	$100K - 500K$
FT_{pr}	50%

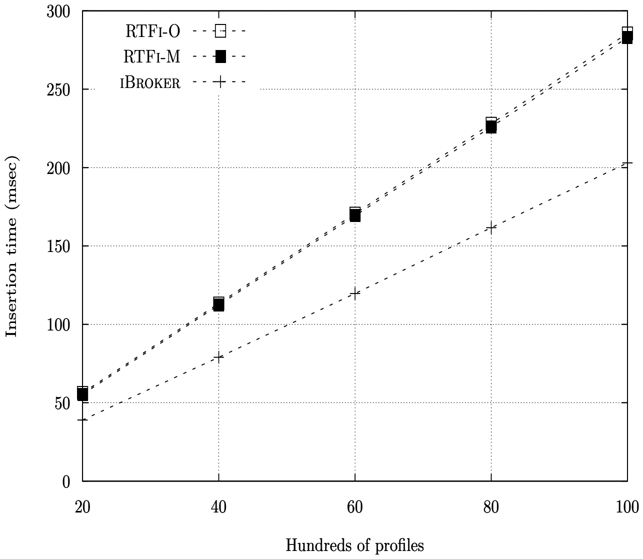
Figure 4.11: Filtering time for all the variations of RTF.

4.6 Results for the second profile collection

In this section, we present the experiment conducted in order to evaluate the algorithms RTFi-O and RTFi-M against the algorithm iBroker. In more detail we evaluate algorithms RTFi-O, RTFi-M and iBroker using the second profile collection as described in Section 4.1.1. The second profile collection is formed by RDF-based profiles that do not bare any full-text operators, making this collection a perfect candidate to evaluate the structural matching capabilities of the proposed algorithms. We present diagrams concerning memory requirements, the indexing and filtering times of all the algorithms for database sizes up to $DB = 100K$. Finally we give the most important results for larger databases for the variations of RTF.

Comparing insertion time

In this section we give the time requirement of algorithms RTFi-O, RTFi-M and iBrokerfor indexing $I_p = 20K$ new profiles in a database of profile with varying sizes.

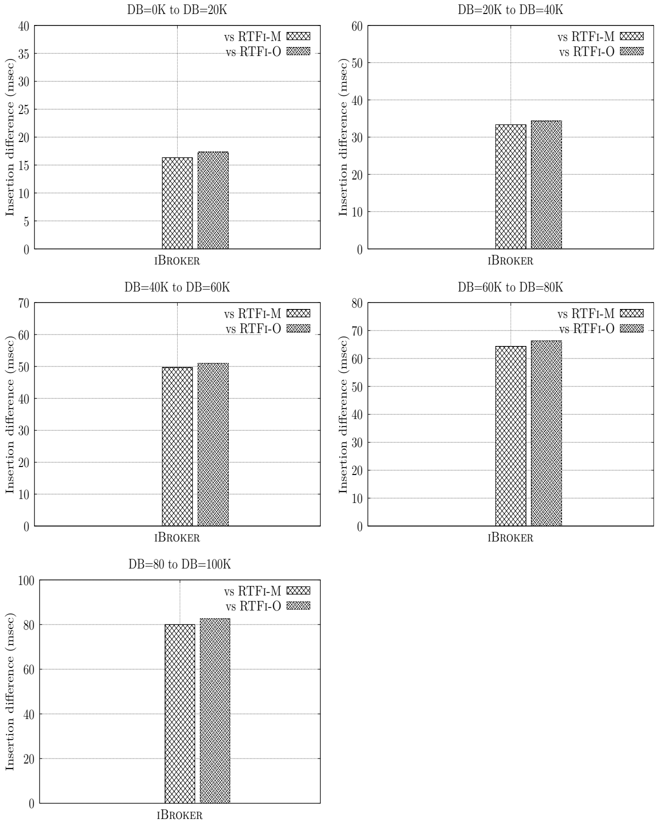


Parameter	Value
I_p	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	0%

Figure 4.12: Insertion time of iBroker and RTF for the second profile collection.

Figure 4.12 presents the insertion time that algorithms RTFi-O, RTFi-M and iBroker spend to index $I_p = 20K$ new profiles of the second profile collection into their databases. We observe that the algorithms increase their time needed to index the new profiles as the database size increases. The algorithms RTFi-O and RTFi-M need more time to index the same amount of profiles $I_p = 20K$ compared to iBroker. Additionally the two variations of RTF increase their time requirements faster compared to iBroker. This performance in the insertion phase can be explained as follows: The algorithms RTFi-O and RTFi-M utilize trie-based data structures in order to index the structural restrictions of the profiles and capture their common elements. The search for clustering common structural restrictions into the trie based structures results to slower indexing performance as opposed to iBroker. Comparing the numbers of Figure 4.6 that demonstrates the insertion times for the main profile collection against Figure 4.12 we can see that the algorithms RTFi-M and RTFi-O double down their time requirements when their are no textual restrictions to be indexed. While iBroker has little time difference when there are no textual restrictions to index.

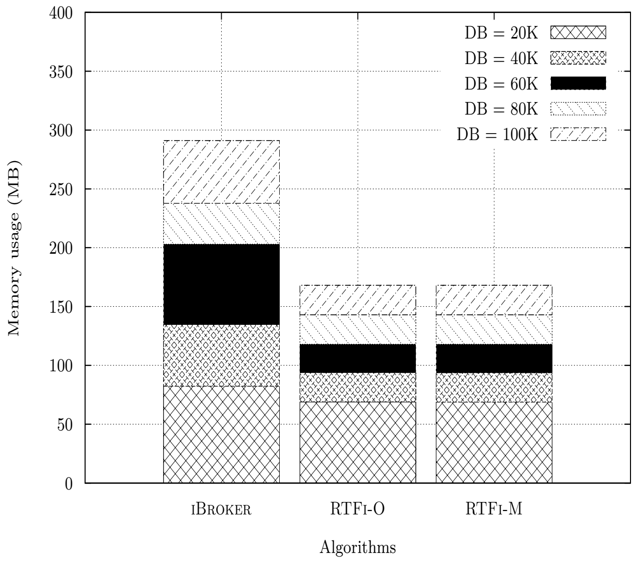
To better demonstrate the difference in insertion increase between the proposed



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

Figure 4.13: Insertion time difference of iBroker and RTF for the second profile collection.

algorithms an their competitors Figure 4.13 summarises the absolute differences of iBroker compared against the competitor algorithms RTFi-O and RTFi-M. We present the differences in insertion time when varying the database sizes when insertion $I_p = 20K$ new profiles. We can observe that iBroker spends less time during the indexing phase and the difference against RTFi-O and RTFi-M increases. Namely when the algorithms index the first $I_p = 20K$ to an empty database, iBroker spends 41.8% less time compared to RTFi-M. The speed of iBroker is maintained during different database sizes. When iBroker indexes the final $I_p = 20K$, thus going from $DB = 80K$ to $DB = 100K$ the difference falls at 39.4%. Likewise iBroker is faster compared against RTFi-O, as it needs 44.4% less time inserting the first $I_p = 20K$ to an empty database. Finally, when iBroker indexes the last $I_p = 20K$ it needs 40.7% less time compared to RTFi-O.

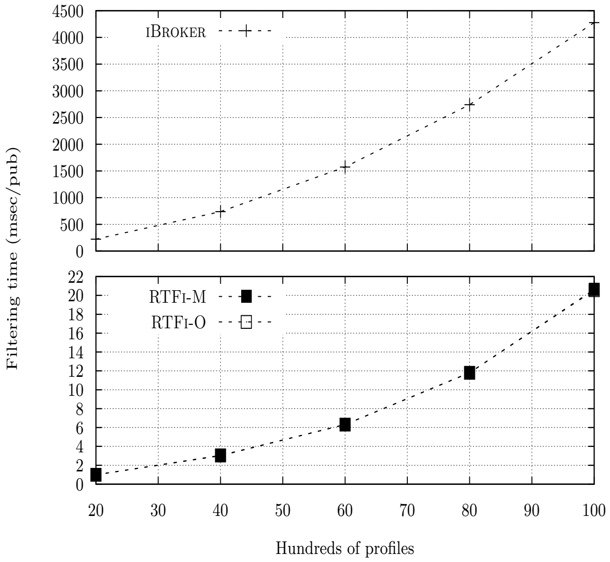


Parameter	Value
I_p	$20K$
I_{pub}	–
DB	$20K - 100K$
FT_{pr}	0%

Figure 4.14: Memory consumption of RTF and iBroker for the second profile collection.

Comparing memory usage

We have also executed experiments to specify the memory requirements for each of the presented algorithms. Figure 4.14 exhibits a good overview of the results for varying database sizes. Algorithms RTFi-O and RTFi-M have the lowest and same memory requirements thus $168MB$. Algorithms iBroker occupies the highest amount of memory requiring $291MB$ to index a database of $DB = 100K$ profiles. The variations of RTF have low memory requirements as they utilize clustering techniques to capture the common structural elements of the profiles. Both present the same memory usage as indexing this profile collection does not require any text indexing structures. Thus creating identical forests for the structural restrictions indexing. iBroker does not utilize any clustering technique for the structural restrictions indexing thus needing more memory to store all the elements of the profiles. Finally we observe that the variations of RTF reserve the majority of their memory when indexing the first $I_p = 20K$ to an empty database. This behaviour can be attributed to the initialization of the indexing structures that the algorithms use. Namely bot RTFi-O and RTFi-M reserve $68MB$ of memory during the first



Parameter	Value
I_p	–
I_{pub}	5000
DB	$20K - 100K$
FT_{pr}	0%

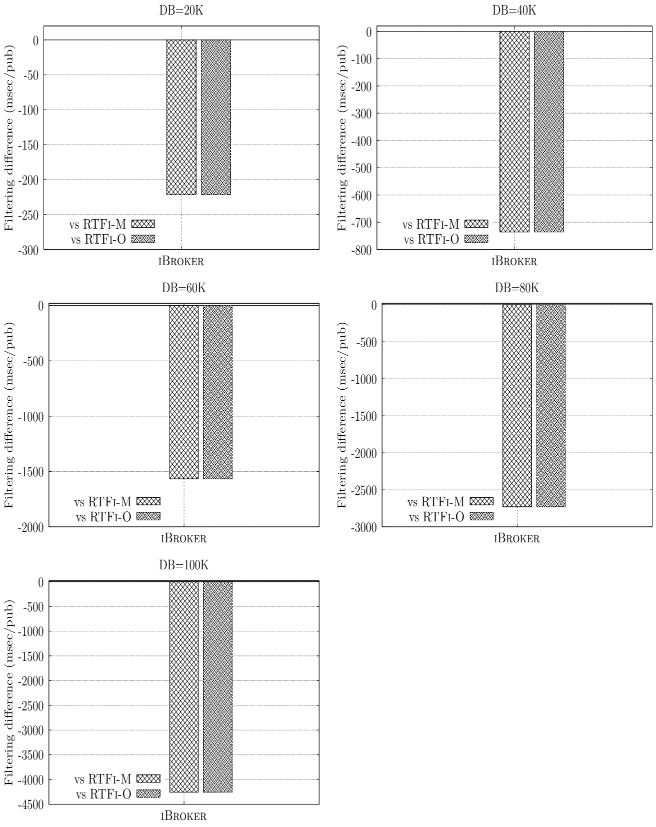
Figure 4.15: Filtering time of iBroker and RTF for the second profile collection.

indexing phase and do not require more than $25MB$ to facilitate the indexing of the new profiles. On the other hand iBroker reserves $82MB$ of memory to index the first $I_p = 20K$ profiles into an empty database while it requires at most $68MB$ of memory to index every set of $I_p = 20K$ new profiles. By comparing the results against the main profile collection (Figure 4.8) results we observe that algorithms RTFi-O and RTFi-M decrease significantly their memory requirements, thus showing a good scalability

Comparing filtering time

This section discusses the results concerning the filtering time required to match an incoming publication against a database of profiles. The time shown in the graphs represent the average spend to filter a collection of $I_D = 5K$ publications.

Figure 4.15 shows the time in milliseconds needed to filter an incoming publication against a profile database of different sizes. Observe that the filtering time increases for all algorithms as the profile database size increases. Algorithms RTFi-O and RTFi-M that utilize clustering techniques to store the profiles achieve the lowest filtering time, suggesting better performance than their competitor iBroker.

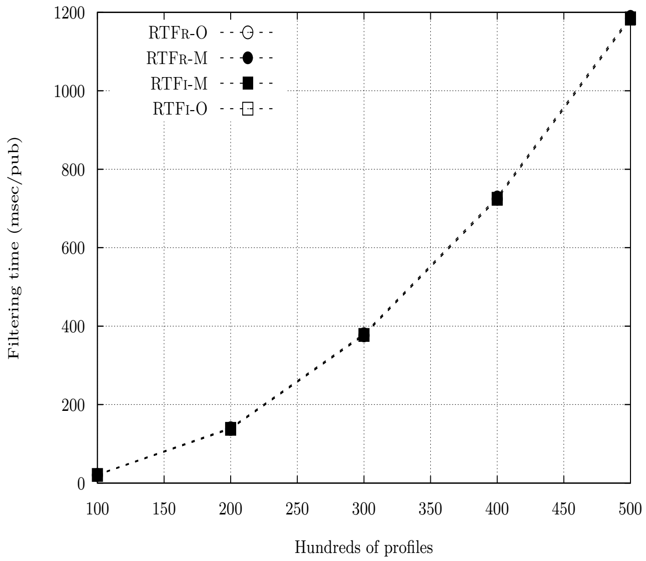


Parameter	Value
I_p	–
I_{pub}	5000
DB	$20K - 100K$
FT_{pr}	0%

Figure 4.16: Filtering time difference between RTF and iBroker for the second profile collection.

Algorithms RTFi-O and RTFi-M are less sensitive to the profile database increase compared to iBroker. RTFi-O and RTFi-M can match against a database faster an incoming publication searching the trie based forests and match profiles in bulk. While iBroker searches and matches every profile independently when filtering a publication.

Figure 4.16 summarises absolute differences in filtering time for iBroker against it's competitors. All differences are computed for varying database sizes. Note that negative numbers in differences indicate that less time is required for RTFi-O and RTFi-M for filtering incoming publications. Algorithms RTFi-O and RTFi-M utilize the same indexing structure to store the structural restrictions of the profile also as there are no full-text restrictions to the second profile collection algorithms perform in the same manner. Namely, algorithms RTFi-M and RTFi-O when



Parameter	Value
I_p	–
I_{pub}	5000
DB	$100K - 500K$
FT_{pr}	0%

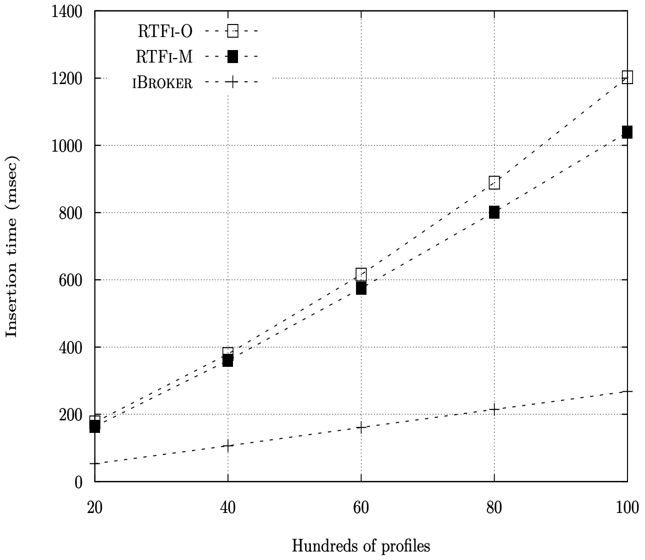
Figure 4.17: Filtering time of the variations of RTF for the second profile collection.

indexing a database size of $DB = 20K$ perform the process of matching a publication 99.55% faster compared to iBroker. Their difference is maintained for all database sizes studied and when the database size reaches $DB = 100K$ their difference is at 99.51%.

Figure 4.17 shows the time in milliseconds needed to filter an incoming publication against a profile database of difference sizes for all the variations of RTF. The results demonstrate the performance of the algorithms for database sizes greater than $DB = 100K$. We can see that the time increases for all algorithms as the profile database size increases. The algorithms exhibit similar performance as all variations of RTF utilize the same indexing structures to capture and store the structural restrictions of the profiles. Their clustering techniques gives them the ability easily match incoming publications into large databases spending few milliseconds.

4.7 Results for the third profile collection

In this section we present the experiments conducted in order to evaluate the algorithms RTFi-O and RTFi-M against then algorithm iBroker. In more detail we evaluate algorithms RTFi-O, RTFi-M and iBroker using the third profile



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

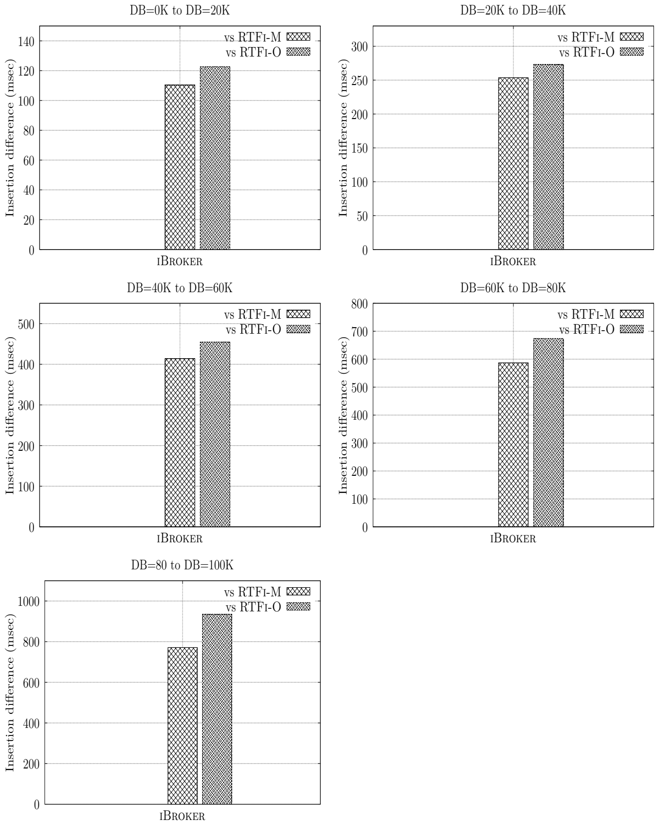
Figure 4.18: Insertion time of iBroker and RTF for the third profile collection.

collection as presented in Section 4.1.1. The third profile collection is formed by RDF-based profiles that bare full-text operators on all triples that form them, making this collection a perfect candidate to evaluate the textual matching capabilities of the proposed algorithms. We present diagrams concerning the indexing times, filtering times and memory requirements of all the algorithms for database sizes up to $DB = 100K$. Finally we present the most important results for larger databases for all the variations of RTF.

Comparing insertion time

This section present the time every algorithm spends in order to index a standard input of $I_p = 20K$ new profiles in a database with varying sizes.

Figure 4.18 presents the insertion time that the algorithms RTFi-O, RTFi-M and iBroker spend to index $I_p = 20K$ new profiles into their database. We observe that the algorithms increase their time needed to index the new queries as the database size increases. The algorithms RTFi-O and RTFi-M need more time to index the same amount of profile compared to iBroker. Additionally the two variants of RTF increase their time requirements faster compared to iBroker. This



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

Figure 4.19: Insertion time difference of iBroker and RTF for the third profile collection.

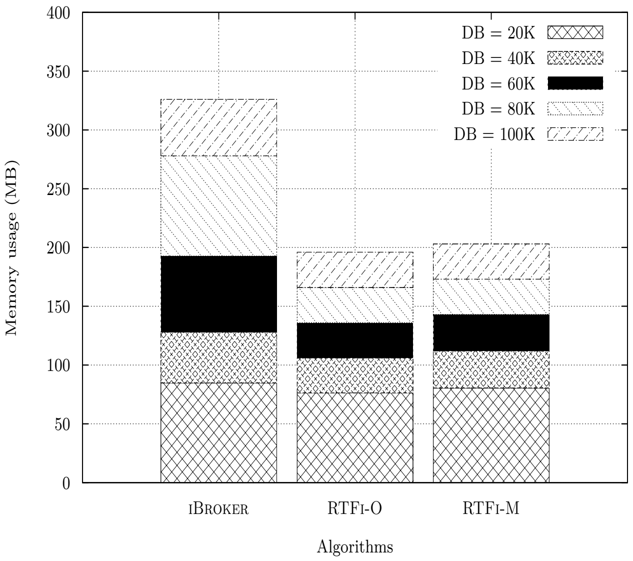
performance in the insertion phase can be explained as follows: The algorithms RTFi-O and RTFi-M as discussed and in previous sections utilize trie-based structures to index and capture the common element of the structural and textual parts of the profiles. The search for existing places to index a profile in the trie based structures results to slower indexing performance compared to iBroker. On the other hand iBroker does not utilize any clustering technique during the indexing phase, thus resulting to low insertion times.

To better demonstrate the differences in insertion increase between the proposed algorithms and their competitors Figure 4.19 summarises the absolute differences of iBroker compared against the competitors algorithms RTFi-O and RTFi-M. We present the differences in insertion time when varying the database sizes when inserting $I_p = 20K$ new profiles. Note that negative numbers in difference indicate that less time is required for RTFi-O and RTFi-M indexing the incoming profiles, while the positive numbers indicate that more time is required compared to iBroker. We can observe that iBroker spends less time during the indexing phase and the difference against RTFi-O and RTFi-M changes slightly. Namely when the algorithms index the first $I_p = 20K$ to an empty database, iBroker spends 206% less time compared to algorithm RTFi-M$\dot{T}$he lead of iBroker is maintained during the increases in database size against RTFi-M. When iBroker indexes the final $I_p = 20K$, thus increasing the database size from $DB = 80K$ to $DB = 100K$ the difference increases at 287%. Likewise iBroker is faster compared against RTFi-O, as it needs 230% less time inserting the first $I_p = 20K$ to an empty database. Finally when iBroker indexes the last $I_p = 20K$ it increase the difference with RTFi-O up to 348%.

This differences great differences in the insertion performance of RTFi-O and RTFi-M can be explained as follows: The third profile collections contains a great amount of full-text operators. Algorithms RTFi-O and RTFi-M try to capture the common elements of the textual restrictions into their indexing structures thus consuming a great amount of time searching for the best available position. In contrast iBroker stores the textual restrictions into lists of string and does not implement any clustering technique, resulting to low insertion times.

Comparing memory usage

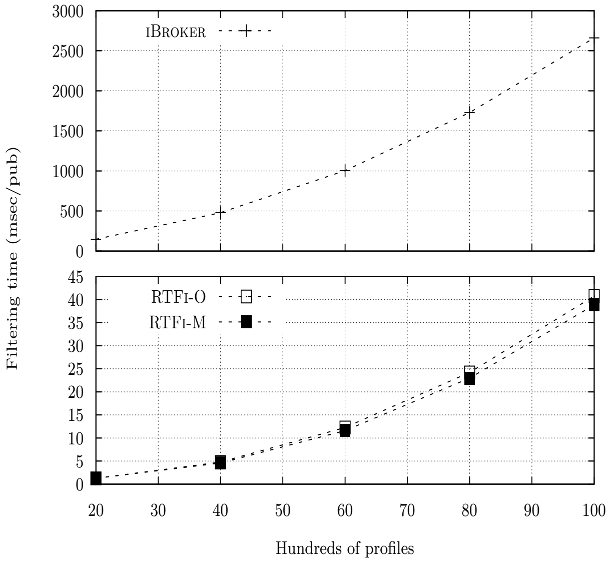
We also executed experiment to specify the memory requirements for each of the presented algorithms. Figure 4.20 exhibits a good overview of the result for varying DB sizes. Algorithm RTFi-O has the lowest memory requirements using $196MB$ for storing the whole profile database $DB = 100K$ and all indexing components. Algorithm RTFi-M's memory usage is at at $203MB$ for storing the same profile database. Algorithm iBroker occupies the highest amount of memory requiring $326MB$ to index a database of $DB = 100K$ profiles. The variations of RTF have low memory requirements as they utilize clustering techniques to capture the common elements of the profiles both structural and textual restrictions. RTFi-O has



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

Figure 4.20: Memory consumption of RTF and iBroker for the third profile collection.

lower memory requirements compared to RTFi-M as it utilizes a single textual indexing structure thus capturing more common elements achieving a more compact forest. While RTFi-M by utilizing multiple forests misses important clustering opportunities requiring more memory to index the same profile set. iBroker does not implements any clustering techniques thus needing more memory to store all the element of the profiles. Finally we observe that RTFi-O and RTFi-M reserve the majority of their memory when indexing the first $I_p = 20K$ profiles to an empty database. This behaviour can be attributed to the initialization of the indexing structures that the algorithms use. Specifically RTFi-O reserves $76MB$ when indexing the first $I_p = 20K$ profiles and RTFi-M reserves $80MB$. For every new $I_p = 20K$ profiles inserted into the database RTFi-O and RTFi-M do not require more than $30MB$ and $31MB$ to facilitate the indexing of the new profiles accordingly. On the other hand iBroker reserves $84MB$ of memory to index the first $I_p = 20K$ profiles into an empty database while it requires more than $40MB$ of memory to index every set of $I_p = 20K$ new profiles.



Parameter	Value
I_p	–
I_{pub}	5000
DB	$20K - 100K$
FT_{pr}	100%

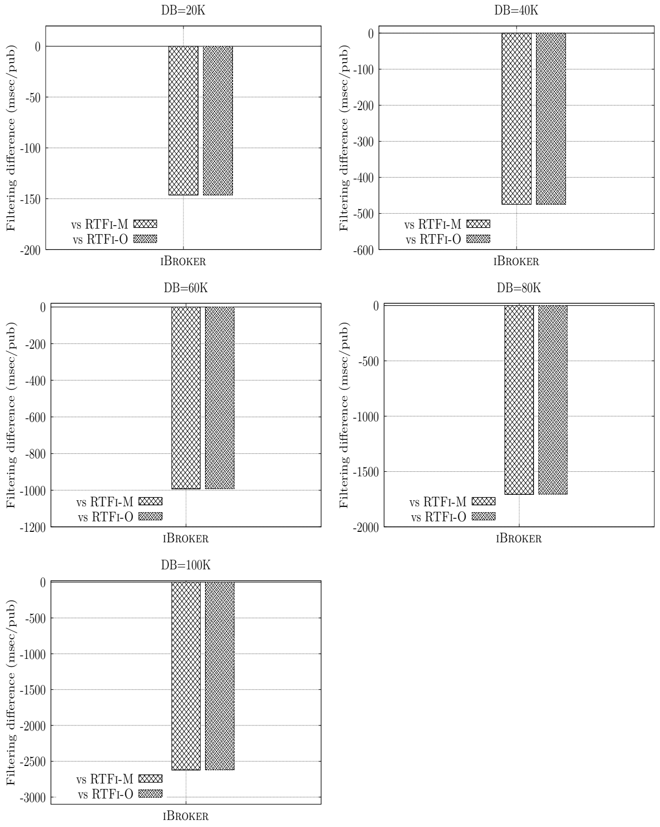
Figure 4.21: Filtering time of iBroker and RTF for the third profile collection.

4.7.1 Comparing filtering time

This section discusses the results concerning the filtering time required to match an incoming publication against a database of profiles. The time shown in the graphs represent the average time spend to filter a collections of $I_D = 5K$ publications.

Figure 4.21 shows the time in milliseconds needed to filter an incoming publication against a profile database of different sizes. Observe that filtering time increase for all algorithms as the profile database size increases. Algorithms RTFi-O and RTFi-M, that utilize clustering techniques for the storage of the profiles achieve the lowest filtering time, suggesting better performance than their competitor iBroker. Algorithms RTFi-O are less sensitive to the profile database size increase compared to iBroker. RTFi-O and RTFi-M can match against to the profile database faster an incoming publication searching the trie based forests and match publications in bulk. While iBroker searches and matches every profile in the database independently when filtering a publication.

To better demonstrate the differences in filtering decrease between the proposed algorithms and their competitors Figure 4.22 summarises the absolute differences of iBroker compared against the competitors algorithms RTFi-O and RTFi-

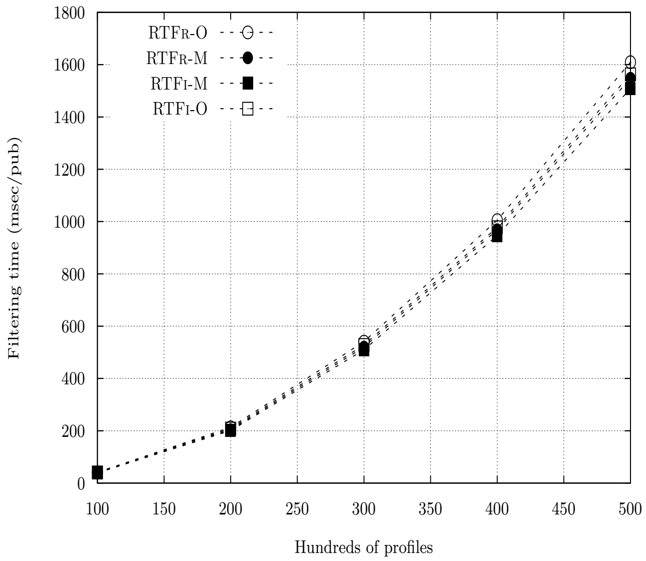


Parameter	Value
I_p	–
I_{pub}	5000
DB	$20K - 100K$
FT_{pr}	100%

Figure 4.22: Filtering time difference between RTF and iBroker for the third profile collection.

M. All differences are computed for varying database sizes. Note that negative numbers in differences indicate that less time is required for RTFi-O and RTFi-M for filtering incoming publications. When algorithm RTFi-M indexes a database size of $DB = 20K$ it uses 99.1% less time compared to iBroker. The performance of RTFi-M is maintained for all database sizes studied and when the database size reaches $DB = 100K$ their difference is at 98.54%. When algorithm RTFi-O indexes a database size of $DB = 20K$ it performs the matching process of a publication 99.1% faster compared to iBroker. Finally when algorithms RTFi-O and iBroker index a $DB = 100K$, RTFi-O performs 98.46% faster compared to iBroker.

Figure 4.23 shows the time in milliseconds needed to filter an incoming publication against a profile database of different sizes for all the variations of RTF. The



Parameter	Value
I_p	–
I_{pub}	5000
DB	$100K - 500K$
FT_{pr}	100%

Figure 4.23: Filtering time of the variations of RTF for the third profile collection.

results demonstrate the performance of the algorithms for database sizes greater than $DB = 100K$. We observe that the time increases for all algorithms as the profile database increases. All variations of RTF exhibit similar performance as all of them utilize the indexing structures to capture and store the structural and textual restrictions of the profiles. Additionally if we compare the filtering performance of the variations of RTF when the database size is $DB = 500K$ against iBroker (Figure 4.21), we see that iBroker spends approximately the same filtering time for a publication when the database is at $DB = 80K$. The algorithms exhibit similar performance as discussed in Section 4.4.2. The ranking of the algorithms is slightly altered during the database increase placing RTFI-O at the top of the performance scale for the maximum database size $DB = 500K$.

4.8 Conclusions

In this section we discuss the findings we received from the experimental evaluation of our proposed algorithm RTF and it's competitor iBroker. We designed three different profile datasets and evaluated our algorithms under three different filtering scenarios.

We presented and discusses the insertion time the algorithms require in order to index the same profile volume in their database. We have shown that the four variations of RTF consume more time in order to index the same profiles in their database compared to iBroker. This can be attributed to the nature of data structures the four variations of RTF utilize. As all variations of RTF utilize clustering techniques to index the structural and textual restrictions of the profiles more time is required in order to locate a position to insert a profile into their data structures. While iBroker by not utilizing any clustering techniques it performs better in the indexing phase.

By comparing the memory requirements of the algorithms we have shown that the four variations of RTF reserve less memory to index the same profile input. This can be attributed to the clustering techniques used as more profiles are indexed under same positions thus creating more compact databases. On the other hand iBroker due to the nature of it's profile indexing structures requires more memory to store the same profile input.

We presented the filtering times and their differences of the four variations of RTF and iBroker for different database sizes. The results suggest that all four variations of RTF outperform the filtering capabilities of iBroker. This excellence is attributed to the nature of the data structures used by RTF, by clustering similar profiles together the filtering process is sped up as more profiles can be matched in groups.

Finally, we conducted all the experimental evaluations for three different profile datasets in order to determine the performance of our algorithms for different scenarios. The main profile collection aimed for an average case scenario where user profiles where filled with 50% of full text restrictions. The second profile collection aimed for a case scenario where there were no full-text restrictions and we studied the performance of the structural matching of the algorithms. The third profile collection aimed to stress test text filtering capabilities of the algorithms. In all three case scenarios all four variations of RTF outperformed the competitor algorithm iBroker.

Chapter 5

Conclusions

In this final chapter of our thesis we will present an overview of the research conducted, we will highlight our main contributions and provide possible directions for future research.

5.1 Summary

In this thesis we studied the problem of full-text support on ontology based publish/subscribe systems. These systems can facilitate the users needs in exploring new information, and may be applied in many domains, such as news alerting systems, RSS feeds and digital libraries. The users by utilizing these systems can achieve great benefits in their need of information delivery, (a) they receive personalized and filtered results that match their interests, and (b) they may stay informed by resorting in a timely information delivery guaranteed by the publish/subscribe system.

In order to facilitate user needs we studied the SPARQL query language and proposed an extension for full-text subscriptions. The developed extension supports full-text subscriptions with capabilities that can easily support all typical Boolean operations applied on text, such as conjunctions, disjunctions, negations proximity and others.

Furthermore we developed the Algorithm RTF that can index hundreds of thousands of user profiles expressed in SPARQL with full-text operators. Additionally Algorithm RTF is able to filter incoming publications more than 98% faster than existing state-of-the-art solutions in the bibliography.

The developed algorithm RTF was tested against the typical data structures of a two-level hash table and more specifically against algorithm iBroker [57]. We designed and conducted three different experimental evaluations. The goal of our experiments was to approximate real-life scenarios and test our algorithm under regular and extreme cases. In this way we demonstrate the efficiency of algorithm RTF against the current state-of-the-art.

We attribute the filtering performance of RTF to the clustering techniques it utilizes in organizing the user profiles into trie-based data structures. RTF separates the user profiles in two parts: one where the RDF restrictions are described and one where the full-text restrictions are defined. Thus RTF indexes the two separated parts of the profiles under two similar data structures each one with it's own unique characteristics. The RDF-part of every profile is indexed under trie-based data structures that can capture and cluster the common parts of the profiles that form the database. Furthermore the textual part of every profile is indexed under trie-based structures. In their turn the trie-based structures are more adequate and specialised to index the text terms that form a textual restriction. In this manner RTF clusters the most common structural and textual parts of the profiles thus providing excellent performance during publication filtering. In contrast, current state-of-the-art solutions focus only on the semantic matching part of the profiles, thus neglecting the optimization of the filtering process. This may lead to systems that do not scale and are unable to support growing number of profiles together with efficient information delivery.

5.2 Contributions

In this section we summarise the contributions of this thesis. We presented a SPARQL query language extension in order to support full-text operators. We aimed to increase the flexibility in profile definition and supply the users with expressive tools in order to better define their interests. Additionally the proposed full-text extension increases the precision of the delivered information to the user as a publication must match to higher number of restrictions.

We presented a novel algorithm coined RTF (acronym for RDF Text Filtering), indexes subscriptions defined in the SPARQL query language. We provided users with better expressivity, by extending the SPARQL query language with full-text operators. The utilization of full-text subscriptions enhances the flexibility and expressiveness of the subscription language. Additionally, we identified and developed four different indexing algorithms to facilitate the structural and textual restriction indexing of the profiles. Namely the four variations of algorithm RTF that were presented are RTFi-O, RTFr-O, RTFi-M and RTFr-M.

Furthermore we modified the and extended the competitor algorithm. We extended the state-of-the-art publish/subscribe solution iBroker [57] algorithm. iBroker is able to support the full-text extension of SPARQL we had introduced. In this way, extending the support of not only text containment but also full-text keyword matching.

Finally, we designed and conducted experimental evaluations with real-world data comparing a state-of-the-art algorithm against our proposed solutions, and concluded that our proposed algorithm outperforms the state-of-the-art in terms of filtering efficiency by at least 98%.

5.3 Future directions

In this section we discuss the open problems of Information Filtering in ontology based publish/subscribe systems and the directions we will focus on in our future work.

With the increased information availability, information filtering systems must be able to deliver highly personalized results to the user and reduce the irrelevant publications. In order to achieve more accurate results, we would like to extend the support of the proposed data models to all typical Boolean operations applied on RDF and text representations. Additionally we intend on supporting semantic matching and study it's impact in the information delivery accuracy.

Additionally, the constant increase of publication volume drives the need for developing highly scalable publish/subscribe systems. Lately there has been a thrive towards cloud-based computational solutions, thus creating opportunities to develop algorithms that are able to utilize them. The proposed algorithm RTF utilizes trie-based structures that render it a proper candidate for exploiting either shared memory or cloud-based architectures parallelization. We would like to study the aspects of this problem and supply a solution for multi-processor parallel, publish/subscribe systems.

Finally, the last years the research concerning graphs is becoming important as there is a plethora of applications in linked data, social networks and bioinformatics (e.g. protein-to-protein interactions). We would like to study the problem of Information Filtering over graph models. Imagine users that have the ability to pose queries expressed in sub-graphs and graph data that are constantly updated with new edges and vertices. The contributions of such research can be applied to linked data, social networks and many more.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] A. Algarni, Y. Li, Y. Xu, and R. Y. K. Lau. An effective model of using negative relevance feedback for information filtering. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, pages 1605–1608, 2009.

[4] M. Altinel, D. Aksoy, T. Baby, M. Franklin, W. Shapiro, and S. Zdonik. DBIS-toolkit: Adaptable Middleware for Large-scale Data Delivery. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, USA*, 1999.

[5] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. Flexpath: Flexible structure and full-text querying for xml. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France*, pages 83–94, 2004.

[6] J.-I. Aoe, K. Morimoto, and T. Sato. An Efficient Implementation of Trie Structures. *SOFTPREX: Software–Practice and Experience*, 22(9):695–721, 1992.

[7] R. Baeza-Yates and G. Gonnet. Fast Text Searching for Regular Expressions or Automaton Simulation on Tries. *Journal of the ACM*, 43(6):915–936, 1996.

[8] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. W3C recommendation, February 2004. URL http://www.w3.org/TR/owl-ref/.

[9] N. Belkin and W. Croft. Information Filtering and Information Retrieval: Two Sides of the Same Coin? *Communications of the ACM (CACM)*, 35(12):29–38, 1992.

[10] T. Bell and A. 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, Zurich, Switzerland*, pages 12–20, 1996.

[11] T. Bell, J. Cleary, and I. Witten. Text Compression. *Prentice-Hall publishers*, 1990.

[12] A. Berglund, S. Boag, D. Chamberlin, M. F. FernG•ndez, M. Kay, J. Robie, and J. SimG•on. XML Path Language (XPath) 2.0, W3C Recommendation, December 2010. URL http://www.w3.org/TR/xpath20/.

[13] J. Callan. Document Filtering With Inference Networks. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Zurich, Switzerland*, 1996.

[14] J. Callan. Learning While Filtering Focuments. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Melbourne, Australia*, pages 224–231, 1998.

[15] A. Carzaniga, D.-S. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, August 2001.

[16] P. Case, M. Dyck, M. Holstege, S. Amer-Yahia, C. Botev, S. Buxton, J. Doerre, J. Melton, M. Rys, and J. Shanmugasundaram. XQuery and XPath Full Text. W3C Recommendation., March 2011. URL http://http://www.w3.org/TR/xpath-full-text-10/.

[17] Y.-I. Chang, J.-H. Shen, and T.-I. Chen. A data mining-based method for the incremental update of supporting personalized information filtering. *Journal of Information Science and Engineering*, 24(1):129–142, 2008.

[18] P.-A. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Publish/Subscribe for RDF-based P2P Networks. In *Proceedings of the First European Semantic Web Symposium (ESWS), Heraklion, Crete, Greece*, pages 182–197, 2004.

[19] D. Comer. Analysis of a Heuristic for Trie Minimization. *ACM Transactions on Database Systems (TODS)*, 6(3):513–537, Sept. 1981.

[20] D. Comer and R. Sethi. The Complexity of Trie Index Construction. *Journal of the ACM*, 24(3):428–440, 1977.

[21] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria*, 2002.

[22] R. de la Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, pages 295–298, 1959.

[23] P. Denning. Electronic Junk. *Communications of the ACM (CACM)*, 25(3): 163–165, 1982.

[24] L. Devroye. A study of trie-like structures under the density model. *The Annals of Applied Probability*, 2(2):402–434, 1992.

[25] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Electronic Proceedings of the ACM SIGMOD Conference, Santa Barbara, CA, USA*, 2001.

[26] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (DEBS), Nashville, Tennessee, USA*, 2009.

[27] P. Flajolet. On the Performance Evaluation of Extendible Hashing and Trie Searching. *Acta Informatica*, 20:345–369, 1983.

[28] P. Flajolet and C. Puech. Partial match retrieval of multidimensional data. *J. ACM*, 33(2):371–407, 1986.

[29] P. Foltz and S. Dumais. Personalized Information Delivery: An Analysis of Information Filtering Methods. *Communications of the ACM (CACM)*, 35 (12):51–60, 1992.

[30] M. Franklin and S. Zdonik. "Data in Your Face": Push Technology in Perspective. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):516–519, June 1998. ISSN 0163-5808.

[31] E. Fredkin. Trie Memory. *Communications of the ACM (CACM)*, 3(9):490–499, 1960.

[32] C. Grün, S. Gath, A. Holupirek, and M. H. Scholl. XQuery Full Text Implementation in BaseX. In *Proceedings of the 6th International XML Database Symposium on Database and XML Technologies (XSym), Lyon, France*, pages 114–128, 2009.

[33] D. Hull, J. Pedersen, and H. Schütze. Method Combination For Document Filtering. In *Proceedings of the ACM SIGIR*, pages 279–287, 1996.

[34] S. Idreos, M. Koubarakis, and C. Tryfonopoulos. P2P-DIET: One-Time and Continuous Queries in Super-Peer Networks. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT), Heraklion, Crete, Greece*, pages 851–853, Heraklion, Greece, March 2004.

[35] P. Jacquet and W. Szpankowski. Analysis of digital tries with Markovian dependency. *IEEE Transactions on Information Theory*, 37(5):1470–1475, 1991.

[36] D. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1973.

[37] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, Reading, Massachusetts, 1973.

[38] M. Koubarakis, T. Koutris, C. Tryfonopoulos, and P. Raftopoulou. Information Alert in Distributed Digital Libraries: The Models, Languages, and Architecture of DIAS. In *Proceedings of the 6th European Conference on Research and*

Advanced Technology for Digital Libraries (ECDL), Rome, Italy, pages 527–542, Rome, Italy, September 2002.

[39] M. Koubarakis, C. Tryfonopoulos, S. Idreos, and Y. Drougas. Selective Information Dissemination in P2P Networks: Problems and Solutions. *SIGMOD Record, Special Issue on Peer-to-Peer Data Management*, 32(3):71–76, 2003.

[40] Y. Li, X. Zhou, P. Bruza, Y. Xu, and R. 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), Napa Valley, California, USA*, pages 1023–1032, 2008.

[41] Y. Li, A. Algarni, S.-T. Wu, and Y. Xue. Mining negative relevance feedback for information filtering. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI), Milan, Italy*, pages 606–613, 2009.

[42] Y. Li, A. Algarni, and Y. Xu. A pattern mining approach for information filtering systems. *Information Retrieval*, 14(3):237–256, 2011.

[43] E. Liarou, S. Idreos, and M. Koubarakis. Publish/subscribe with rdf data over large structured overlay networks. In *Proceedings of the Databases, Information Systems, and Peer-to-Peer Computing, International Workshops (DBISP2P), Trondheim, Norway*, pages 135–146, 2005.

[44] H. Luhn. A Business Intelligence System. *IBM Journal of Reasearch and Development*, 2(4):314–319, 1958.

[45] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

[46] A. Mishra, S. Gurajada, and M. Theobald. Running sparql-fulltext queries inside a relational dbms. In *Proceedings of Evaluation Labs and Workshop (Online Working Notes) (CLEF), Rome, Italy*, 2012.

[47] M. Morita and Y. 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, pages 272–281, 1994.

[48] N. Nanas and M. Vavalis. A "bag" or a "window" of words for information filtering? In *Proceedings of the 5th Hellenic Conference on Artificial Intelligence: Theories, Models and Applications (SETN), Syros, Greece*, pages 182–193, 2008.

[49] N. Nanas, S. Kodovas, and M. Vavalis. Revisiting evolutionary information filtering. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC), Barcelona, Spain*, pages 1–8, 2010.

[50] N. Nanas, M. Vavalis, and A. N. D. 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), Geneva, Switzerland*, pages 202–209, 2010.

[51] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. Edutella: a p2p networking infrastructure based on rdf. In *Proceedings of the 11th International World Wide Web Conference (WWW), Honolulu, Hawaii*, pages 604–615, 2002.

[52] B. Nguyen, S. Abiteboul, G.Cobena, and M. Preda. Monitoring XML Data on the Web. In *Proceedings of the ACM SIGMOD Conference, Santa Barbara, CA, USA*, Santa Barbara, CA, USA, 2001.

[53] S. Nilsson and G. Karlsson. IP-Address Lookup Using LC-Tries. *IEEE Journal of Selected Areas in Communications*, 17(6):1083–1092, 1999.

[54] E. Panzeri and G. Pasi. An approach to define flexible structural constraints in xquery. In *Proceedings of the 8th International Conference on Active Media Technology (AMT), Macau, China*, pages 307–317, 2012.

[55] E. Panzeri and G. Pasi. A flexible extension of xquery full-text. In *Proceedings of the 4th Italian Information Retrieval Workshop (IIR), Pisa, Italy*, pages 29–32, 2013.

[56] E. Panzeri and G. Pasi. Flexible structural constraints in xquery full-text. In *OAIR*, 2013.

[57] M.-J. Park and C.-W. Chung. ibroker: An intelligent broker for ontology based publish/subscribe systems. In *Proceedings of the 25th International Conference on Data Engineering (ICDE), Shanghai, China*, pages 1255–1258, 2009.

[58] L. Pellegrino, F. Huet, F. Baude, and A. Alshabani. A distributed publish/subscribe system for rdf data. In *Proceedings of the 6th International Conference on Data Management in Cloud, Grid and P2P Systems (GLOBE), Prague, Czech Republic*, pages 39–50, 2013.

[59] J. Peterson. Computer Programs for Detecting and Correcting Spelling Errors. *Communications of the ACM (CACM)*, 23(12):676–686, 1980.

[60] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-topss: Semantic toronto publish/subscribe system. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), Berlin, Germany*, pages 1101–1104, 2003.

[61] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-topss: fast filtering of graph-based metadata. In *Proceedings of the 14th International Conference on World Wide Web (WWW), Chiba, Japan*, pages 539–547, 2005.

[62] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, W3C Recommendation, January 2008. URL `http://www.w3.org/TR/rdf-sparql-query/`.

[63] J. Qian, J. Yin, and J. Dong. Parallel matching algorithms of publish/subscribe system. In *Proceedings of the 8th International Conference on Information Technology: New Generations (ITNG), Las Vegas, Nevada, USA*, pages 638–643, 2011.

[64] P. Raftopoulou, E. G. Petrakis, C. Tryfonopoulos, and G. Weikum. Information Retrieval and Filtering over Self-Organising Digital Libraries. In *Proceedings of the 12th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*, Aarhus, Denmark, September 2008.

[65] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM Conference on*

Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), San Diego, CA, USA, pages 161–172, 2001.

[66] M. Regnier and P. Jacquet. New Results on the Size of Tries. *IEEE Transactions on Information Theory*, 35(1):203–205, 1989.

[67] R. L. Rivest. Partial-Match Retrieval Algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976.

[68] G. Schreiber and Y. Raimond. Rdf 1.1 primer., February 2014. URL `http://http://www.w3.org/TR/rdf11-primer/`.

[69] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), San Diego, CA, USA*, pages 149–160, 2001.

[70] E. Sussenguth. Use of Tree Structures for Processing Files. *Communications of the ACM (CACM)*, 6(5):272–279, 1963.

[71] C. Tryfonopoulos, M. Koubarakis, and Y. Drougas. Filtering Algorithms for Information Retrieval Models with Named Attributes and Proximity Operators. In *Proceedings of the 27th Annual International ACM SIGIR Conference*, pages 313–320, Sheffield, UK, July 2004.

[72] C. Tryfonopoulos, M. Koubarakis, and Y. Drougas. Information filtering and query indexing for an information retrieval model. *ACM Transactions on Information Systems (TOIS)*, 27(2), 2009.

[73] W. Vanderbauwhede, L. Azzopardi, and M. Moadeli. Fpga-accelerated information retrieval: High-efficiency document filtering. In *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL), Prague, Czech Republic*, pages 417–422, 2009.

[74] J. Wang, B. Jin, and J. Li. An ontology-based publish/subscribe system. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada*, pages 232–253, 2004.

[75] T. Yan and H. Garcia-Molina. Index Structures for Selective Dissemination of Information Under the Boolean Model. *ACM Transactions on Database Systems (TODS)*, 19(2):332–364, 1994.

[76] T. Yan and H. Garcia-Molina. Index Structures for Information Filtering under the Vector Space Model. *Proceedings of the 10th International Conference on Data Engineering, Houston, Texas, USA*, pages 337–347, 1994.

[77] T. Yan and H. Garcia-Molina. The SIFT Information Dissemination System. *ACM Transactions on Database Systems (TODS)*, 24(4):529–565, 1999.

[78] B. Yang and H. Garcia-Molina. Designing a Super-peer Network. In *Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, India*, March 5–8 2003.

[79] J. A. Yochum. A High-Speed Text Scanning Algorithm Utilising Least Frequent Trigraphs. In *IEEE Symposium on New Directions in Computing*, 1985.

[80] Y. Zhang and J. Callan. Maximum likelihood estimation for filtering thresholds. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, New Orleans, Louisiana, USA*, 2001.

[81] X. Zhou, Y. Li, P. Bruza, Y. Xu, and R. Y. K. Lau. Rough sets based reasoning and pattern mining for a two-stage information filtering system. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management (CIKM), Toronto, Ontario, Canada*, pages 1429–1432, 2010.

[82] X. Zhou, Y. Li, P. Bruza, Y. Xu, and R. Y. K. Lau. Pattern mining for a two-stage information filtering system. In *Proceedings of the 15th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD), Shenzhen, China*, pages 363–374, 2011.