Full-Text Support for Publish/Subscribe Ontology Systems

Lefteris Zervakis[1(✉)], Christos Tryfonopoulos[1(✉)], Spiros Skiadopoulos[1], and Manolis Koubarakis[2]

[1] Department of Informatics and Telecommunications, University of Peloponnese, Tripolis, Greece
{zervakis,trifon,spiros}@uop.gr
[2] Department of Informatics and Telecommunications, Univeristy of Athens, Athens, Greece
koubarak@di.uoa.gr

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

1 Introduction

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

In the early days of content-based pub/sub the structure of a publication was nothing more than a (usually static) collection of named attributes with

L. Zervakis and S. Skiadopoulos have been partly supported by Greek national funds through the EICOS project (Research Funding Program Thales, Operational Program "Education and Lifelong Learning", National Strategic Reference Framework.

H. Sack et al. (Eds.): ESWC 2016, LNCS 9678, pp. 233–249, 2016.
DOI: 10.1007/978-3-319-34129-3_15

values of different types (e.g., text) [19,21]. As XML gained popularity and started becoming the standard for data/information representation and exchange on the web, various XML-based pub/sub systems have, naturally, arised [3,6–8,14]. In those systems, publications were expressed in XML and extensions of XPath/XQuery were used to express continuous queries. All research in the field focused mainly on the structural/value matching between (indexed) continuous queries and incoming publications, but has largely ignored semantics. This gave rise to ontology-based pub/sub systems [12,15,16,20] that typically used RDF [18] for representing publications and SPARQL [17] extensions/modifications for expressing user interests through continuous queries.

Ontology-based pub/sub systems research [12,15,16,20] has naturally focused more on semantics and has delivered interesting results. What it currently lacks, though, compared to the technological arsenal of the traditional pub/sub research is the support of a complete full-text retrieval mechanism, beyond existing regular expression and equality support, with sophisticated algorithms and data structures to minimise processing and memory requirements.

In this work, we initially propose an *extension* of SPARQL with full-text operators, aiming at more expressive continuous queries that are able to support versatile user needs in applications like digital libraries or news filtering. To preserve the expressivity of SPARQL, we view the full-text operations as an additional filter of the query variables. In our setup, publications are ontology data that contain RDF literals in their property elements. A full-text expression is evaluated against a literal, and supported expressions involve the usual Boolean operators (i.e., conjunction, disjunction, negation), as well as word proximity and phrase matching. To efficiently filter the incoming publications against the stored queries, we present RTF (acronym for RDF Text Filtering), a family of trie-based, main-memory, (continuous) query indexing algorithms that support SPARQL queries with full-text constraints and are able to filter incoming publications in a few milliseconds. We propose indexing methods that exploit the commonalities between continuous queries at indexing time and leverage on the natural properties of RDF during the filtering procedure. To the best of our knowledge, our family of algorithms is the first in the literature that is able to support SPARQL queries with full-text constraints. To demonstrate the efficiency of our approach we extend iBroker [12], a state-of-the-art query indexing and RDF publication filtering algorithm, with full-text capabilities and compare it against our approach both on structural and full-text filtering tasks; our approach proves more than two orders of magnitude faster for the structural and more than one order of magnitude faster for the full-text filtering tasks.

In the light of the above, our contributions are:

- We *extend* SPARQL with full-text operators and support Boolean, word proximity, and phrase matching operators.
- We *develop* a family of continuous query indexing algorithms that support full-text SPARQL queries and are able to filter the incoming RDF publications efficiently.

- We *extend* iBroker [12], a third party algorithm for ontology pub/sub, to offer full-text support and use it as a state-of-the-art competitor.
- We *identify* algorithmic alternatives for query indexing and assess their performance with a real-word data set against the extended version of iBroker.

The rest of the work is organised as follows. Section 2 presents an overview of our data and query model, while Sect. 3 introduces the RTF family of algorithms and outlines the competitor extensions. Subsequently, Sect. 4 presents the experimental evaluation of the developed algorithms with a real-world data set. Finally, Sect. 5 presents related research in ontology pub/sub systems, and Sect. 6 concludes the paper.

2 Query and Data Model

RDF constitutes a conceptual model and a formal language for representing resources in the Semantic Web; it is the building block of a metadata layer on top of the current structured information layer of the *World Wide Web*, which enables interoperability between different systems and facilitates the exchange of machine-understandable information. The SPARQL query language is currently the W3C recommendation for querying the Semantic Web; the graph model, over which it operates, naturally joins data together and supports several query forms for querying RDF datasets. However, it still lacks the support of a complete full-text mechanism for filtering purposes. Since we focus our attention on full-text filtering of ontology data we are interested only in property elements with a plain RDF literal as their content. In this context, the subject of an RDF triple is always a node element and the predicate denotes the relation to the literal. The object is the literal, which is expressed as a string.

In the spirit of [2], we propose an extension to the SPARQL syntax to support full-text continuous queries in RDF datasets. To preserve SPARQL expressibility we view the full-text operations as an additional filter of the (continuous) query variables. In this context, we define a new binary operator *ftcontains* (full-text contains), that takes as input a *variable* of the continuous SPARQL query and a *full-text expression* that operates on the values of this variable. The query signature of the operator is expressed as the function $xsd : boolean : ftcontains(var, ft_{expression})$. A full-text expression is evaluated only against a literal, so *var* is always the object of the SPARQL tuple pattern; the subject and/or predicate of the tuple pattern may be constants. The expressions supported involve the usual Boolean operators (denoted by ftAND, ftOR, etc.), as well as proximity (denoted by ftNEAR) and phrase matching as in [4]. To this end, we carefully designed a new set of full-text queries which currently can not be efficiently evaluated by existing pub/sub ontology systems.

The example SPARQL continuous query in Fig. 1 will match all publications that are of type *article* and have an attribute *title* with a string literal. The title of the publications must contain the terms “olympic” and “games”. Additionally, the publications that match must have an attribute *body* that contains the terms

```
1 SELECT ?publication
2 WHERE {?publication type article.
3         ?publication title ?title.
4         ?publication body ?body.
5 FILTER ftcontains(?title, "olympic" ftAND "games")
6 FILTER ftcontains(?body, "olympic" ftAND "games" ftNEAR[0,2] "rio")
7 }
```

Fig. 1. An example SPARQL query with the proposed extended syntax.

"olympic", "games" and "rio", and the term "rio" is at least 0 and at most 2 words after the term "games" (due to the word proximity constraint).

In addition to the full-text extension of SPARQL we also support the *wildcard (*)* operator applied in RDF triples, i.e., queries where the subject, predicate and/or object of a triple may match any value of the publication. Such a combination of full-text and wildcard operations allows us to offer to users a rich set of tools that allow them to specify expressive continuous queries that will match their information needs. An example query of this type could be derived by substituting line 2 of Fig. 1 with "`WHERE {? publication type *.`".

A publication, in this context, is represented as a set of RDF triples containing additional fields, where needed, to store the text parts. 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 and are connected via properties that are expressed as the predicate.

3 Query Indexing Algorithms

In this section, we present RTF, a family of query indexing algorithms that utilise trie structures to exploit commonalities between continuous queries to achieve faster filtering times. Initially, we elaborate on the indexing algorithm RTFM[1], discuss its variation RTFS, and provide details for the common filtering procedure. Finally, we briefly discuss IBROKER, a state-of-the-art competitor that uses an inverted index to store submitted SPARQL queries.

3.1 Algorithm RTFM

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

1. Transforming the continuous query to conjunctions of tuples (quadruples or triples depending on the existence of a text constraint or not) and assigning a unique identifier to each tuple.
2. Registering all the discrete tuples produced from the previous step in a table that associates each continuous query with the tuple identifiers it contains.
3. Indexing of all the query tuples at the trie structure described below.

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

In the following, we analyse each step and provide details on the data structures and algorithms utilised.

Step 1: Tuple Representation. Algorithm RTFM operates on tuples; in this section we define continuous queries as conjunctions of tuples, and, in the following sections, we illustrate how we exploit commonalities between those tuples to achieve better query indexing and thus faster filtering performance.

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

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

Example 1. By applying Definition 1 to the continuous query q in the example of Fig. 1 we receive the following set of tuples:

```
(?publication, type, article) ∧
(?publication, title, ?title, ftcontains("olympic" ftAND "games")) ∧
(?publication, body, ?body,
                    ftcontains("olympic" ftAND "games" ftNEAR[0,2] "rio"))
```

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

Step 2: Associating Queries with Tuple Identifiers. Following Step 1, RTFM receives a query q that consists of two fields, a unique *query identifier* and a set of tuples also associated with their unique tuple ids. RTFM proceeds by storing each continuous query along with the tuple identifiers into the Query Table (QT). QT is comprised of two fields: the unique identifier of each query q and a linked list that stores the unique identifiers of the continuous query tuples. For instance, for the continuous query of Fig. 1, RTFM will add three tuple identifiers into QT (as they are shown in the previous step). RTFM proceeds in a similar way to insert every new continuous query that is submitted in QT.

Step 3: Indexing Tuples in the Trie Forest. The *trie forest* is populated in order to store the tuples compactly by exploiting their common elements. Thus, every trie forest consists of a collection of tries, which in turn contain a number of trie nodes; in each node N the following information is stored:

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

[2] http://rdf4j.org.

When a new query q arrives RTFM iterates through the set of all query tuples and indexes every tuple in the trie forest. During the indexing phase RTFM searches the trie forest for a suitable place to index each tuple as follows.

The first tuple of the first continuous query that is submitted will naturally arrive in an empty trie forest and will create a number of nodes that depend on the form of the tuple. Specifically, for the structural constraints of the tuple, RTFM creates three new trie nodes one for each attribute specified. If the tuple contains also a full-text constraint with k distinct words, RTFM will create k more nodes (one for each distinct word). For illustration purposes, we use a pseudo-node "FT" to separate the structural from the word constraints and highlight the difference between the different RTF variants.

In general, when inserting a new tuple, RTFM considers storing it at an existing trie or creating a new trie. To insert a new tuple $t(\mathcal{S}, \mathcal{P}, \mathcal{O})$, RTFM examines the subject $\mathcal{S}$ of the tuple and utilises the trie structure to find if there is a candidate trie which has a root node R such that $content(R) = \mathcal{S}$. If such a trie is found, the indexing algorithm proceeds to examine $children(R)$ in order to determine if there is a child C such that $content(C) = \mathcal{P}$. The same applies for the object $\mathcal{O}$ of the tuple. Notice that variables (in subject/predicate/object) and wildcards in tuples are mapped onto the corresponding variable/wildcard nodes. If the new tuple contains full-text constraints, the trie is expanded with the distinct words contained in these tuple constraints in a similar manner.

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

Indexing of proximity formulas and phrases in the trie forest of RTFM is performed in the same way as described above, since proximity is a more constrained case of conjunction. To accommodate the word distance in the proximity/phrase expression, we use an extra data structure that stores the proximity constraints in the spirit of [19]. Disjunctions are handled by creating separate queries (that have the same user as the notification recipient) for the different word operands.

Figure 2 shows the resulting trie after inserting three continuous queries, including the query q of Fig. 1. Additionally, the three tuples of q (shown in Example 1 above) are assigned ids $q.t_1, q.t_2, q.t_3$ respectively. From the indexing performed by RTFM in these queries notice that:

- Query q shares the same tuple (`(?publication, type, article)`) with query v, as two different tuple identifiers (namely $q.t_1$ and $v.t_1$) are stored in the same leaf node. Moreover, this tuple contains only structural constraints.
- Query q contains tuple $q.t_3$ that specifies both structural and full-text constraints.

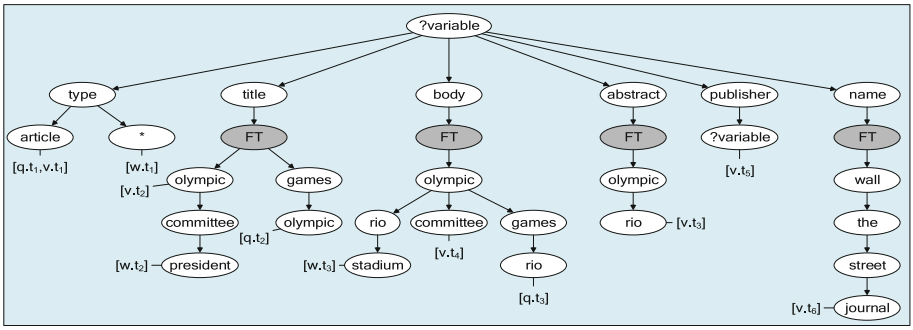

Fig. 2. Trie forest during the indexing phase of RTFM.

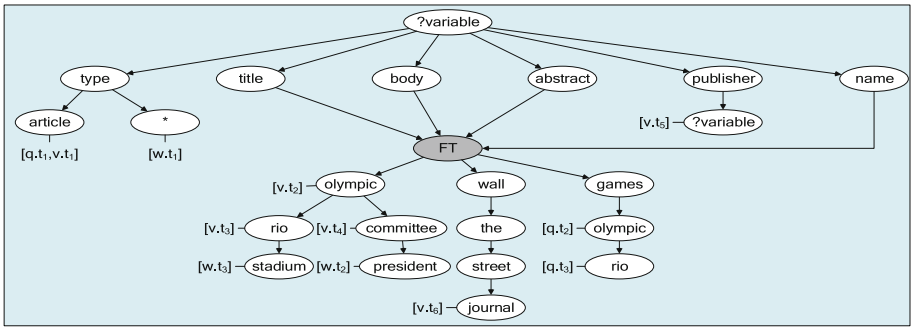

Fig. 3. Trie forest during the indexing phase of RTFS.

- Query q (with tuple $q.t_3$) shares the same structural constraints and also has the word `"olympic"` in common for the textual constraints part with query v (with tuple $v.t_4$).

Finally, note that Fig. 2 shows just one of the tries that would be created; typically, because of different query structure the resulting indexing structure is a *forest of tries*. Thus, a hash table (not shown in Fig. 2 to avoid cluttering) is used to provide fast access to trie roots.

Algorithm RTFs. Indexing of word constraints in the context of RTF may be performed in two different ways: (i) using *multiple* tries (hence the name RTFM) for indexing the word constraints depending on the structural part of the continuous queries as described in the previous section (and shown in Fig. 2), and (ii) using a *single* trie forest (hence the name RTFS) that is dedicated to all text components, regardless of the structural part of the continuous queries (shown in Fig. 3 – not described in detail due to space considerations).

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

together regardless of the structural constraints of the query. Algorithm RTFs is a variation that allows us to construct a more compact[3] forest of tries since this organisation creates more clustering opportunities for the words. As we will demonstrate in Sect. 4, RTFs is better suited for cases where queries with text constraints are relatively sparse, whereas RTFm is better suited for cases where many queries contain text constraints.

3.2 Filtering Algorithm

The filtering algorithm is common for the two variants (RTFm and RTFs) of RTF. In this section, we present the filtering algorithm that allows RTF to filter incoming RDF publications and issue notifications to subscribed users.

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

Definition 2. *We define a publication p as a series of $i, i \in 1 \ldots, n$ conjuncts of RDF triples. Each triple has three attributes, namely* subject *($\mathcal{S}_i$),* predicate *($\mathcal{P}_i$) and* object *($\mathcal{O}_i$) or* text field *($\mathcal{T}_i$) that represents the textual content of an attribute. Thus, a publication may be represented as:*

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

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

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

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

[3] Notice that the trie of Fig. 3 has less nodes compared to that of Fig. 2 for the same queries and the same query insertion order.

```
Algorithm: FILTER
1  Function traverseTrie(node N, tuple t)
2      if content(N) is satisfied by t then
3          matchedTuples ← matchedTuples ∪ tIDs(N)
4          traverseTrie(C,t), where C ∈ children(N)
5  Function filterPublication(p)
6      matchedTuples ← Null
7      foreach tuple t in publication p do
8          foreach trie root R do
9              traverseTrie(R,t)
10     isSatisfied ← TRUE
11     foreach query q in QT do
12         foreach tuple t ∈ q do
13             if t is not marked as matched then
14                 isSatisfied ← FALSE
15                 break
16     if isSatisfied==TRUE then
17         notify subscriber
```

Fig. 4. Pseudocode for publication filtering.

processing of publication p (i.e., after processing all its tuples), a scan of QT allows us to determine the queries that have matched the incoming publication. The pseudocode of the filtering process for RTF variants is given in Fig. 4.

3.3 Algorithm iBroker

To evaluate the efficiency of RTF we have also implemented iBroker [12] as a *baseline competitor*. iBroker is a continuous query indexing algorithm that supports SPARQL queries with structural and string matching constraints, and is currently *the only* state-of-the-art algorithm that is able to handle RDF queries with both structure and (some form of) text. In this section, we outline the basic idea behind iBroker and the data structures upon which it operates and show how we extended its functionality to support full-text constraints.

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

This inverted index stores the queries in a chain-like manner. Every query may be recomposed by following the *NextToMatch* references to hash buckets until an empty *NextToMatch* field is visited. This procedure is applied by the algorithm iBroker during the filtering of a publication event. As there is no defined hierarchy that outlines the filtering sequence, an incoming publication may need to examine many hash buckets looking for the beginning of a query. The result is that iBroker must in this case examine all the continuous queries

in the corresponding bucket list, and then proceed to examine their *NextToMatch* entries until there is none left to match.

iBroker implements string matching, but has no support for full-text operators. To evaluate it against RTF we have extended iBroker with full-text subscriptions by replacing the *Value* field with the list of words that appear in a full-text constraint. This modification enables iBroker to support both string and full-text constraints. Finally, for comparison purposes, we have also extended the functionality of iBroker to index and filter SPARQL queries that contain wildcard operators. For a more detailed description of iBroker and the specifics of its algorithms the interested reader is referred to [12].

4 Experimental Evaluation

In this section, we present a series of experiments that compare RTF against iBroker under a series of different scenarios.

Data and Query Set. For the experimental evaluation we utilised, the *DBpedia* corpus (http://dbpedia.org/Downloads2015-04) extracted from the *Wikipedia* domain that forms a structured knowledge database of more than 4 million items. A major part, namely 3.22 millions publications, of the *DBpedia* corpus, has been classified into an ontology resulting in 529 different classes which are described by 2.3 thousand properties. Additionally, publications bear textual information that originates from human generated content published at the *Wikipedia* domain. The vocabulary extracted from the *DBpedia* publications consists of 3.14 millions unique words. The maximum textual information present in a publication is 14,254 words, while, the average is 53 words. The diversity in content of the *DBpedia* corpus accompanied with the information on structural and textual level, renders it as the perfect candidate for evaluating our algorithms indexing and filtering efficiency.

Query Set. The queries were constructed by utilising classes and properties extracted from the *DBpedia* corpus. Each query, contains at most 4 tuples. The query set, is formed by sets of tuples containing full-text operators with probabilities of $FT_{pr} = 0\,\%$, $50\,\%$ or $100\,\%$. The full-text operators contain conjunctive terms that are selected equiprobably among the multi-set of words from the *DBpedia* vocabulary. Additionally, the full-text operators contain at most 3 terms. Queries with $FT_{pr} = 0\,\%$, examine the performance of the algorithms for structural matching only. For queries, with $FT_{pr} = 50\,\%$, we examine a mixed filtering scenario where half of the queries contain also textual constraints apart from the structural ones. Finally, queries with $FT_{pr} = 100\,\%$ demonstrate the scaling capabilities of the algorithms as they all contain full-text constraints.

Publication Set. In order to evaluate the query collections, described above, we selected $I_{pub} = 5\,\text{K}$ publications from the *DBpedia* corpus. The set selected, had structural and textual information as extracted and processed from the *Wikipedia* domain. The publications contain human-generated, real-life data, thus providing a realistic overview of the performance of the algorithm. We

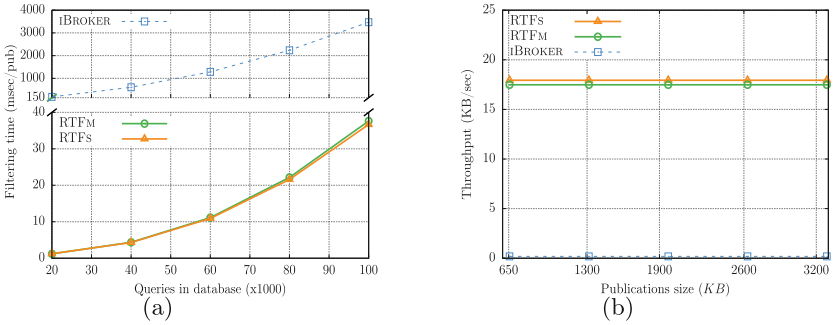

Fig. 5. Comparing (a) filtering time when varying DB and (b) filtering throughput when $DB = 100\,\text{K}$, for queries of $FT_{pr} = 50\,\%$.

maintain the same publication set through the evaluation process against different query collections. Thus, it is asserted that the algorithms are evaluated based on their indexing capabilities and the nature of queries they index.

Metrics Employed. In our evaluation, we present and discuss the filtering time and throughput of each algorithm, i.e., the amount of time needed to locate all continuous queries satisfied by an incoming publication. We present and compare the memory requirements of the algorithm. As all algorithms index the same query databases, a lower memory requirement indicates a more compact clustering of data while a higher memory footprint a less compact database. Finally, we present the insertion time of each algorithm, i.e., the amount of time needed to index a set of queries $I_p = 20\,\text{K}$ into the database.

Technical Configuration. All algorithms were implemented in C++, and an off-the-shelf PC (Core *i*7 3.6 GHz, 8 GB RAM, Ubuntu Linux 14.04) was used. The time shown in the graphs is wall-clock time and the results of each experiment are averaged over 10 runs to eliminate fluctuations in time measurements.

4.1 Results When Varying the Query Database Size

In this section, we present the most significant findings for the proposed algorithms, when, varying the query database size DB for queries of $FT_{pr} = 50\,\%$.

Comparing Filtering Time. Figure 5(a) presents the time in milliseconds needed to filter an incoming publication for $I_{pub} = 5\,\text{K}$ publications, when the DB size is increasing. Notice that the y-axis is split into two parts due to high differences in the performance of RTF variants and iBroker. We observe that filtering time increases for all algorithms as the DB size grows. Algorithms RTFs and RTFm achieve the lowest filtering times, suggesting better performance. Algorithm iBroker, is more sensitive to DB size changes compared to RTF due to its query indexing structures, i.e., an inverted index that does not implement any clustering techniques. More specifically, the results indicate that algorithm RTFm

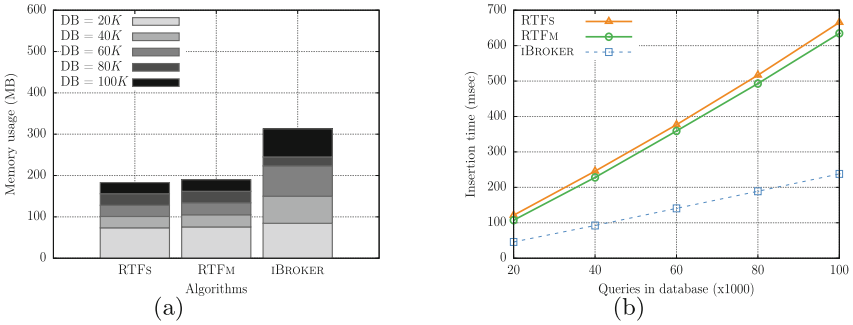

Fig. 6. Comparing (a) memory usage and (b) insertion time when varying DB for $FT_{pr} = 50\,\%$.

filters incoming publications 92 times faster compared to iBroker. Finally, algorithm RTFs achieves the lowest filtering time (2.5 % faster than RTFm and 94 times faster than iBroker), i.e., 36 ms/publication.

Comparing Filtering Throughput. We present the results concerning the algorithms' filtering throughput, when, indexing $DB = 100\,\text{K}$ queries for $I_{pub} = 5\,\text{K}$ incoming publications. Figure 5(b) presents the throughput all algorithms achieve during the filtering of $I_{pub} = 5\,\text{K}$ incoming publications. We observe that the throughput remains steady throughout the publication events. This is attributed to the nature of the algorithms, as their filtering capability is not affected by the publications size but from the indexing structures that store the queries. Algorithms RTFs and RTFm achieve the highest throughput, thus the best performance, compared to algorithm iBroker. More specifically algorithms RTFs and RTFm achieve a throughput of more than 17 KB/s that corresponds to more than 27 publications/s. Contrary, iBroker accomplishes a throughput of 0.18 KB/s that corresponds to 0.28 publications/s.

Comparing Memory Usage. In Fig. 6(a), we present the results for the memory requirements of each algorithm when increasing the query database DB by $I_p = 20\,\text{K}$ new continuous queries in each iteration. Algorithm RTFs has the lowest memory requirements using 183 MB for storing the whole query database $DB = 100\,\text{K}$. Algorithm's RTFm memory usage is 190 MB for the same $DB = 100\,\text{K}$, as it maintains multiple forests of tries for the indexing of textual constraints. We observe that RTF's variations reserve the majority of their memory when indexing the first $I_p = 20\,\text{K}$ to an empty database. This is due to the creation of many new tries at index structure initialisation. Namely, RTFs reserves 73 MB when indexing the first $I_p = 20\,\text{K}$ queries and RTFm reserves 75 MB. For every new $I_p = 20\,\text{K}$ inserted into the database RTFs and RTFm do not require more than 28 MB to facilitate the indexing of new queries due to the accommodation of new queries mostly in existing tries. Finally, algorithm iBroker occupies 313 MB of memory to index a database of $DB = 100\,\text{K}$ queries. iBroker reserves 84 MB for the first $I_p = 20\,\text{K}$ queries, while it requires more than 60 MB of memory to index every set of $I_p = 20\,\text{K}$ new queries.

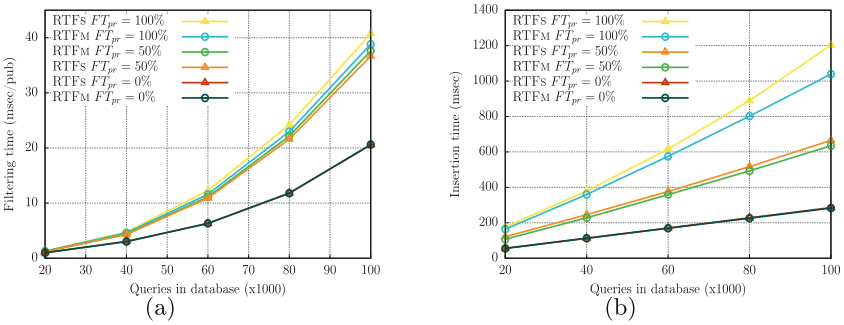

Fig. 7. Comparing RTF's (a) filtering and (b) insertion time when varying DB size and FT_{pr}.

Comparing Insertion Time. In this section, we discuss the query indexing time of all algorithms. Figure 6(b) shows the insertion time in milliseconds required to insert $I_p = 20\,\text{K}$ queries when the DB size increases. We observe that the algorithms require more time to index new queries as the database size increases. Algorithms RTFs and RTFM need more time to index the same number of queries $I_p = 20\,\text{K}$ compared to iBROKER. This can be explained as follows. The variations of RTF utilise trie-based data structures to capture and index the common structural and textual constraints of the queries. Trie traversal results to high insertion time during the indexing phase. On the other hand, insertion in an inverted index (as done by iBROKER) is faster. Notice that insertion time is not critical in a pub/sub scenario; the most important dimension is filtering time/throughput that defines the processing rate of publications.

4.2 Results When Varying the Full-Text Percentage

This section presents the most interesting results concerning the RTF variants when varying the percentage of full-text constraints in the tuples. Notice that we do not show iBROKER (that has a significantly worse performance than the RTF variants as demonstrated in the previous sections) to avoid cluttering the graphs. We evaluate the structural matching performance of RTFM and RTFs when $FT_{pr} = 0\,\%$, and stress-test the algorithms when the query database contains the highest number of full-text constraints possible, i.e., when $FT_{pr} = 100\,\%$.

Comparing Filtering Time. Figure 7(a) shows the time needed to filter an incoming publication against full-text constraints with $FT_{pr} = 0\,\%, 50\,\%$ and $100\,\%$, when increasing the DB size. As expected, RTFs and RTFM achieve the lowest filtering times when $FT_{pr} = 0\,\%$, and exhibit the same performance as they utilise the same indexing structure for the structural constraints of the queries. Finally, RTFs and RTFM increase their filtering times when $FT_{pr} = 100\,\%$ with RTFM achieving better performance.

Comparing Insertion Time. Figure 7(b) shows the time required to insert $I_p = 20\,\text{K}$ queries when DB size is increasing and varying $FT_{pr} = 0\,\%$, $50\,\%$

and 100 %. As expected, algorithms RTFs and RTFM increase their time needs, when more textual constraints ($FT_{pr} = 100\,\%$) are included in the continues queries, and reduce them when no textual constraints ($FT_{pr} = 0\,\%$) are present.

Comparing Memory Usage. We give an outline of RTFs's and RTFM's memory requirements, when $DB = 100\,\text{K}$, for queries of $FT_{pr} = 0\,\%$ and 100 %. Both, RTFs and RTFM have the lowest memory requirements when $FT_{pr} = 0\,\%$, namely 168 MB. Finally, for $FT_{pr} = 100\,\%$, 196 MB and 203 MB are required for RTFs and RTFM respectively.

4.3 Summary of Results

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

5 Related Work

In this section we discuss pub/sub approaches in centralised and distributed environments and contrast them to our approach.

Centralised Ontology Pub/Sub Systems. The S-ToPSS [15] system was among the first designs that supported pub/sub functionality in an ontological context. S-ToPSS was designed to enhance the matching process aiming at semantically similar but syntactically different information present in publications and user subscriptions. This was achieved by identifying synonyms and utilising concept taxonomies and hierarchies. Its successor, G-ToPSS [16], focused on information dissemination of RDF data on ontologies, emphasising on scalability and fast filtering of RDF data. G-ToPSS represented publications as directed labelled graphs, while a two-level hash table was used for the subscriptions; the matching algorithm traversed the publication and subscription graphs. In the same spirit, the Ontology-based Pub/Sub (OPS) system [20] supported events with complex data structures and aimed for a uniform representation. Subsequently, user subscriptions and publication events were processed into RDF graphs and thereafter indexed or filtered respectively by utilising graph matching algorithms. Finally, OPS examined the matching trees that emerged from the graph traversal to determine the matching subscriptions. The Sparkwave [10] system was built to perform continuous pattern matching over RDF streams by supporting expressive pattern definitions, sliding windows and schema-entailed knowledge. The C-SPARQL [1] extension enabled the registration of continuous SPARQL queries over RDF streams, thus, bridging data streams with knowledge bases and enabling stream reasoning.

Although all the aforementioned works focus on supporting pub/sub functionality in ontology systems, none has considered supporting any form of text extension. The work closest to ours is iBroker [12], an OWL-based pub/sub

mediator focused in filtering publications from OWL ontologies against a set of stored SPARQL queries. iBroker matched incoming events generated from an ontology against user queries by resorting on an inverted index to represent the graph that indexed user subscriptions. Although there is no text support, iBroker is able to perform string matching using the inverted index mentioned above. In this work, we extended iBroker with full-text support and used it as a baseline competitor for our algorithms to showcase the performance gains.

Distributed Ontology Pub/Sub Systems. With the advent of distributed and P2P computing, decentralised ontology pub/sub systems naturally emerged. The first P2P pub/sub system based on RDF data was build by Chirita et al. [5]; the system utilised a super-peer architecture where super-peers were responsible for the routing of the content determined by the RDF schema, property or value, while peers were responsible for specific schemas and properties. At publication time, super peers routed the data to the responsible peers for the filtering process; the performance gain was achieved by utilising the content similarities present in subscriptions. Similarly, Liarou et al. [11] studied the problem of evaluating multi-predicate conjunctive queries in pub/sub systems; the aim of the system was to distribute the load of the matching process into a P2P network. In the same spirit, an RDF-based pub/sub P2P network was build by Pelegrino et al. [13] to study the messaging paradigm. The system supported the creation of queries by making use of SPARQL and publications by using RDF data. Users' subscriptions were indexed into a peer, determined by the CAN protocol. Data from a publication event that concerned a peer was stored while the event was forwarded to other peers. Finally, Kaoudi et al. [9] presented a study for distributed RDF reasoning and query answering. The work in [9] focused on implementing, optimising, and evaluating forward and backward chaining over a distributed hash-table for a subset of SPARQL.

None of the works mentioned above supports full-text in their data/query model. Finally notice that our solution can be extended in a decentralised environment by distributing the triple forest to different nodes and modifying the filtering process to visit only nodes that may contain matching queries.

Connection to Other Technologies. The vast majority of RDF stores (Apache Jena[4] Text module, Virtuoso[5], Allegrograph[6], OntoText GraphDB[7]) offers text indexing and full-text retrieval combined with SPARQL. Our solution shares ideas with these technologies, but pub/sub copes with different problems and challenges compared to traditional retrieval. Finally, pub/sub functionality on ontologies may complement many applications including LOD platforms such as Lotus[8] by enabling users to get notified for information of interest, or by providing a useful moderation/monitoring tool for curators/editors of such systems.

[4] http://jena.apache.org/.
[5] http://virtuoso.openlinksw.com/.
[6] http://franz.com/agraph/allegrograph/.
[7] http://ontotext.com/products/graphdb/.
[8] http://lotus.lodlaundromat.org/.

6 Conclusions and Outlook

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

Currently, we are working on supporting VSM queries/text representation in SPARQL, and adapting our algorithms to multi-processor environments.

References

1. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: C-SPARQL: a continuous query language for RDF data streams. IJSC **4**, 3–25 (2010)
2. Case, P., Dyck, M., Holstege, M., Amer-Yahia, S., Botev, C., Buxton, S., Doerre, J., Melton, J., Rys, M., Shanmugasundaram, J.: XQuery and XPath Full Text. (2011). http://www.w3.org/TR/xpath-full-text-10/
3. Chan, C.Y., Felber, P., Garofalakis, M.N., Rastogi, R.: Efficient filtering of XML documents with XPath expressions. In: ICDE (2002)
4. Chang, C.C.K., Garcia-Molina, H., Paepcke, A.: Predicate rewriting for translating Boolean queries in a heterogeneous information system. ACM TOIS **17**, 1–39 (1999)
5. Chirita, P.-A., Idreos, S., Koubarakis, M., Nejdl, W.: Publish/subscribe for RDF-based P2P networks. In: Bussler, C.J., Davies, J., Fensel, D., Studer, R. (eds.) ESWS 2004. LNCS, vol. 3053, pp. 182–197. Springer, Heidelberg (2004)
6. Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P.M.: Path sharing and predicate evaluation for high-performance XML filtering. ACM TODS **28**, 467–516 (2003)
7. Green, T.J., Gupta, A., Miklau, G., Onizuka, M., Suciu, D.: Processing XML streams with deterministic automata and stream indexes. ACM TODS **29**, 752–788 (2004)
8. Hou, S., Jacobsen, H.: Predicate-based filtering of XPath expressions. In: ICDE (2006)
9. Kaoudi, Z., Miliaraki, I., Koubarakis, M.: RDFS reasoning and query answering on top of DHTs. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 499–516. Springer, Heidelberg (2008)
10. Komazec, S., Cerri, D., Fensel, D.: Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In: DEBS (2012)
11. Liarou, E., Idreos, S., Koubarakis, M.: Publish/subscribe with RDF data over large structured overlay networks. In: DBISP2P (2005)
12. Park, M.J., Chung, C.W.: iBroker: an intelligent broker for ontology based publish/subscribe systems. In: ICDE (2009)
13. Pellegrino, L., Huet, F., Baude, F., Alshabani, A.: A distributed publish/subscribe system for RDF data. In: GLOBE (2013)
14. Peng, F., Chawathe, S.S.: XPath queries on streaming data. In: SIGMOD (2003)
15. Petrovic, M., Burcea, I., Jacobsen, H.A.: S-ToPSS: semantic toronto publish/subscribe system. In: VLDB (2003)
16. Petrovic, M., Liu, H., Jacobsen, H.A.: G-ToPSS: fast filtering of graph-based metadata. In: WWW (2005)

17. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF (2008). http://www.w3.org/TR/rdf-sparql-query/
18. Schreiber, G., Raimond, Y.: RDF 1.1 Primer (2014). http://www.w3.org/TR/rdf11-primer/
19. Tryfonopoulos, C., Koubarakis, M., Drougas, Y.: Information filtering and query indexing for an information retrieval model. TOIS **27**, 1–47 (2009)
20. Wang, J., Jin, B., Li, J.: An ontology-based publish/subscribe system. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 232–253. Springer, Heidelberg (2004)
21. Yan, T.W., Garcia-Molina, H.: The SIFT information dissemination system. ACM TODS **24**, 529–565 (1999)