Query Reorganisation Algorithms for Efficient Boolean Information Filtering

Lefteris Zervakis, Christos Tryfonopoulos, Spiros Skiadopoulos, Manolis Koubarakis

Abstract—In the information filtering paradigm, clients subscribe to a server with continuous queries that express their information needs and get notified every time appropriate information is published. To perform this task in an efficient way, servers employ indexing schemes that support fast matches of the incoming information with the query database. Such indexing schemes involve *(i)* main-memory trie-based data structures that cluster similar queries by capturing common elements between them and *(ii)* efficient filtering mechanisms that exploit this clustering to achieve high throughput and low filtering times. However, state-of-the-art indexing schemes are sensitive to the query insertion order and cannot adopt to an evolving query workload, degrading the filtering performance over time. In this paper, we present an adaptive trie-based algorithm that outperforms current methods by relying on query statistics to reorganise the query database. Contrary to previous approaches, we show that the nature of the constructed tries, rather than their compactness, is the determining factor for efficient filtering performance. Our algorithm does not depend on the order of insertion of queries in the database, manages to cluster queries even when clustering possibilities are limited, and achieves more than 96% filtering time improvement over its state-of-the-art competitors. Finally, we demonstrate that our solution is easily extensible to multi-core machines.

Index Terms—Information filtering, user profiles and alert services, indexing methods, dissemination.

1 Introduction

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

In more detail, the problem of information filtering may be defined as follows: given a database DB of continuous queries that reside on a server and an incoming document d, retrieve all queries $q \in DB$ that match d. The filtering problem is of high importance and needs to be solved efficiently, since servers are expected to handle millions of user queries and high rates of published documents. Efficiency issues were identified by many researchers that proposed tree and trie-based algorithms for supporting fast filtering under various data models (e.g., flat attribute-based, semi-structured XML) and query languages (e.g., Boolean, VSM), both for main-memory [2]–[4] and secondary storage [5]. However, all these approaches use a greedy clustering method that is sensitive to the insertion order of submitted queries and do not consider that an evolving query workload might require the reorganisation of the query database to achieve efficient filtering performance.

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

Our algorithm, coined STAR (acronym for STAtistical Reorganisation), is the first in the literature to consider *database reorganisation* (through appropriate word/query statistics) to achieve efficient textual IF under the Boolean model. The

- *L. Zervakis, C. Tryfonopoulos and S. Skiadopoulos are with the Department of Informatics and Telecommunications, University of the Peloponnese, Tripolis, Greece. e-mail: {zervakis, trifon, spiros}@uop.gr*
- *M. Koubarakis is with the Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Athens, Greece. e-mail: koubarak@di.uoa.gr*

main idea behind the proposed algorithm is to use tries to capture common elements of queries, similarly to [3]–[5]. However, the key differences with these approaches lie in *(i)* the collection and utilisation of statistics on the importance of keywords in the indexed queries, *(ii)* the reorganisation of the query database according both to *word* and *query importance*, and *(iii)* the demonstration that the nature of the trie forest is more important than its compactness when it comes to filtering efficiency. Interestingly, all previous works [3]–[5] were aiming at *minimising the size* of the trie forest, since there was an implicit conjecture that a small forest would result in lower filtering times due to less node visits. In this work, we demonstrate that forest size is not the dominating optimisation factor when it comes to filtering efficiency; contrary, the focus should be put on the nature of the tries and on qualitative characteristics (expressed through heuristics).

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

- We present a *novel* query indexing and reorganisation algorithm that supports Boolean IF up to 96% faster than state-of-the-art competitors. Query reorganisation is also efficient; 500K queries (in a database of millions) can be reorganised in only a few seconds on a commodity PC.
- We identify different *reorganisation options* for the trie indexes and demonstrate the importance of query insertion order in the construction of the indexing structure. We also show that constructing tries with *rare* words at the higher level of the trie leads to improved filtering performance due to early pruning at filtering time.
- We demonstrate, contrary to previous works, that the *nature* of the constructed tries, rather than their *compactness*, is the determining factor for efficient filtering performance, especially in datasets with rare clustering opportunities.
- We experimentally evaluate different reorganisation strategies and showcase their effect in filtering efficiency using two different *real-world datasets* and both *synthetic* and *real* query sets.
- We extend the presented algorithm implementation by parallelising the filtering process to suit modern multi-core processors. We identify two different parallelisation options and experimentally evaluate their performance.

The rest of the paper is organised as follows. Section 2 surveys related work. Section 3 presents Algorithm STAR and discusses possible extensions, while Section 4 presents our experimental evaluation and comparison against existing approaches. Finally, Section 5 provides future directions.

2 RELATED WORK

One of the first filtering systems based on the Boolean model to address performance was LMDS [6] that relied on least frequent trigrams for query indexing. In LMDS, profiles are indexed under the least frequent trigram, whereas documents are represented as a sequence of trigrams. At filtering time a table lookup determines which profiles match the incoming document. Since false positives may incur, a second stage is necessary to determine the actual matches. In a similar spirit, [7] presents a scalable IF system where a server receives documents at a high rate and the proposed algorithms support VSM queries by improving Algorithm SQI of [8]. InRoute [9] was another influential system based on inference networks and belief propagation techniques with emphasis on filtering efficiency. Recently, [10] utilised graph structures to locate and index the subsumption relationships between continuous VSM queries and deliver the top-k most relevant notifications. Although [10] aims at filtering efficiency, its approach and index structures are not appropriate for Boolean IF as their focus is on the aggregation function needed to cover VSM semantics.

Most of the work on IF efficiency in the database literature has its origins in [11] and the DBIS system [12]. Publish/subscribe (pub/sub) in the database field focused on the performance of systems with data models based on attribute-value pairs and query languages based on attributes with arithmetic/string comparison operators (e.g., [2], [13]–[15]). BE-Tree [16], [17] is the most recent work in the area that utilises space partitioning techniques and adaptive tree structures to create subsumption hierarchies of continuous boolean expressions and filter them against incoming events. However, this data structure is designed for arithmetic and string operations and is not applicable in textual IF. Other works concentrated on XML and XPath/XQuery-based query models (e.g., [18]–[21]).

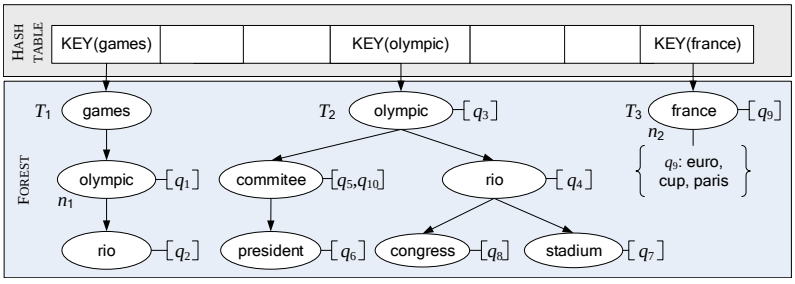
In our approach, the concept of list tries [22] is used and extended for constructing the data structures described in the following sections. Our work is heavily influenced by IF systems, such as SIFT [5], [23] and DIAS [24], and the algorithms closest to STAR are TREE [5], BESTFITTRIE [3], and RETRIE [4], which are trie-based indexing algorithms for textual Boolean queries. The main idea behind these algorithms was to cluster queries by exploiting their common elements. In Algorithm TREE [5], a query was considered as a sequence of words sorted in lexicographic order and a trie was used to index queries by exploiting common prefixes. Algorithm BESTFITTRIE [3] improves on TREE [5] by handling query words as a set rather than as a sorted sequence and by exhaustively searching the forest of tries to discover the best place to store a new query. These algorithms are heavily influenced by query insertion order.

The first algorithm to identify the importance of query insertion order and its influence in the filtering time was Algorithm RETRIE [4]. Algorithm RETRIE introduced the concept of query relocation; identified poorly indexed queries and re-indexed them in better positions, achieving a limited form of re-organisation in the query database. The downside in this approach was twofold: *(i)* although it was beneficial for a part of the indexed queries, it degraded clustering in other parts of the query database (e.g., in the tries initially indexing the relocated queries), and *(ii)* it was still heavily dependent on the initial creation of the tries, which in turn was influenced by query insertion order. Contrary to the aforementioned approaches, our proposal is the first in the literature that emphasises on the reorganisation of the query database and addresses the issue of query insertion order.

Finally, many IF efforts focused more on appropriate representations of user interests [25], [26] and on improving filtering effectiveness [27], [28]. In [25], behaviour monitoring and substring indexing are used to decide which documents

$q_1 = \{\text{olympic, games}\}$
$q_2 = \{\text{olympic, games, rio}\}$
$q_3 = \{\text{olympic}\}$
$q_4 = \{\text{olympic, rio}\}$
$q_5 = \{\text{olympic, committee}\}$
$q_6 = \{\text{olympic, committee, president}\}$
$q_7 = \{\text{olympic, rio, stadium}\}$
$q_8 = \{\text{olympic, congress, rio}\}$
$q_9 = \{\text{euro, cup, france, paris}\}$
$q_{10} = \{\text{olympic, committee}\}$

(a)



(b)

Fig. 1. (a) Queries and (b) their organisation in the indexing phase of Algorithm STAR

match user interests. Moreover, [27], [28] explores an ensemble of methods from machine learning, aiming at increasing filtering effectiveness in an IF setup. Other approaches included statistical filtering systems, such as [29] that uses Latent Semantic Indexing to filter incoming documents and [30] that utilises network-based profile representations to better identify user interests and cope with the curse of dimensionality in VSM. Adaptive filtering [31], [32] focuses also on profile effectiveness and considers the adaptation of VSM queries and their dissemination thresholds. In order to enhance user information discovery, [33] developed a novel statistical latent class model that applies user/item grouping to deliver better content recommendations/predictions. Moreover, sophisticated user profiling has also been used to promote personalised IR systems (e.g., [34]) that focus on improving retrieval effectiveness.

Our work relates (at a higher level) to the studies presented since users' information needs are expressed through profiles. However, these works' focus is on the effectiveness of the user profiles/modelling to facilitate the creation of more relevant notifications, while in our setup the focus is on efficiency: i.e., given a set of (automatically/manually generated) user profiles determine which match an incoming event quickly. Notice that the semantics and filtering effectiveness of our approach are given by the Boolean model and are well studied in the literature (e.g., [36]).

3 THE ALGORITHM STAR

In this section, we consider queries that comprise of conjunctions (i.e., sets) of keywords, as in Fig. 1(a), and present Algorithm STAR that relies on the reorganisation of the query database to achieve low filtering times. Subsequently, we outline existing trie-based solutions and discuss extensions of our algorithm to support more expressive queries.

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

- The indexing phase (Section 3.1), where (initial) query indexing takes place. The pseudocode for the indexing phase of Algorithm STAR is performed by INDEX and is shown below.
- The reorganisation phase (Section 3.2), where *only newly indexed* queries in the database are reorganised by utilising the *statistics* collected during the indexing phase.

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

3.1 Query indexing phase

To index queries, Algorithm STAR 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 Fig. 1(a) are organised in the structures of Fig. 1(b). Each trie node n:

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

For instance consider Fig. 1(b), where $kwrd(n_1) = \text{olympic}$ and node n_1 stores one reference to query q_1. Consider also node n_2 of trie T_3 that stores query $q_9 = \{\text{euro, cup, france, paris}\}$. Since $kwrd(n_2) = \{\text{france}\}$, we have that $uexp(n_2, q_9) = \{\text{euro, cup, paris}\}$. Finally, note that n_2 contains all keywords of q_9 (since $q_9 = kwrd(n_2) \cup uexp(n_2, q_9)$), thus, it also maintains a reference to q_9. 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 trie nodes.

The forest of tries is populated in order to store queries compactly by exploiting their *common keywords*. When a new query q arrives, Algorithm STAR considers its keywords and inserts them in a (new or existing) trie in the forest. For this task, STAR selects the best trie T in the forest and the best node n in that trie to insert q (the insertion process is described later in the section). To this end, STAR uses

```
Algorithm: INDEX
Input: A query q = {k_1, ..., k_t}
Result: Store q in FOREST
1  currentNR ← 0;
2  position ← Null;
   // For all candidate tries T
3  foreach trie T with root(T) = k ∈ q do
       // DFS traversal for all possible storage positions
4      foreach node n ∈ T such as kwd(n) ∈ q do
5          calculate (nr(q, T));
           // If a better position is found store it
6          if currentNR < nr(q, T) then
7              currentNR ← nr(q, T);
8              position ← n;
   // If q cannot be indexed in any existing trie
9  if position = Null then
10     create trie T' with root(T') such as kwrd(T') ∈ q;
11     id(root(T')) ← q ; // Index q in root(T')
12     uexp(T', q) ← q \ kwrd(T') ; // Put the rest in uexp(T', q)
13 else
       // If there are not common keywords
14     if uexp(position, p) ∩ q = ∅ then
15         id(position) ← id(position) ∪ q; // Index q in position
           // Put the rest in uexp(position, q)
16         uexp(position, q) ← q \ {k_1, ..., k_y};
17     else // Else expand the common keywords
18         expand uexp(position, p) ∩ q;
19         id(m) ← q ∪ p;   // Index q and p at the leaf node
           // Remove p from id(position)
20         id(position) ← id(position) \ p;
           // Put the rest in two new uexp lists
21         uexp(m, q) ← q \ {k_1, ..., k_x};
22 gatherStats(q); // Gather statistics for query reorganisation
```

the concept of *node reusability*, denoted by $nr(q,T)$, that quantifies the percentage of q's keywords that are stored in a path starting from the root of T and also used by other queries. More formally, $nr(q,T) = \frac{|path|}{|q|}$, where $|path|$ is the size of the longest path from the root of trie T that contains only keywords of q participating to other queries and $|q|$ is the number of keywords in q. It follows that $0 \leq nr(q,T) \leq 1$, and generally when $nr(q,T)$ is close to 0, trie T is considered as a poor candidate for q because only a small fraction of terms in q will be stored in existing nodes of T. Contrary when $nr(q,T)$ is close to 1, trie T is considered as a good candidate, because a large fraction of terms in q will be stored in existing nodes of T. Node reusability extends the clustering ratio concept [4] with the constraint that keywords should be present in other queries and promotes frequent/rare keywords towards trie roots.

Example 1. Let us consider the queries and their organisation illustrated in Fig. 1. We have $nr(q_1,T_1) = \frac{2}{2}$, since the 2 keywords of q_1 are both stored in a path starting from the root of T_1 and also used in a different query (i.e., q_2). We also have $nr(q_2,T_1) = \frac{2}{3}$, since only 2 keywords of q_2 (out of 3) are stored in a path starting form the root of T_1 and also used in a different query (i.e., q_1), as keyword rio is used solely for q_2.

The algorithm for inserting a new query proceeds as follows. The first query that arrives, creates a trie with a randomly chosen keyword as the root; the remaining keywords are stored at the $uexp$ list of the root. The second query will consider being stored at the existing trie or create a new trie. In general, to insert a new query q, STAR 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, STAR then chooses the trie T among the candidates for which q insertion maximises $nr(q,T)$. To compute $nr(q,T)$, STAR performs a depth-limited search with depth limit $|q|-1$ in *all* candidate tries. This search finds node n in T where q should be inserted. Note that the chosen path from the root to n is the longest path in T that exclusively contains keywords of q. If more than one tries maximise $nr(q,T)$, STAR randomly chooses one.

To complete insertion, the path from the root of trie T to node n, that already stores the identifier of a query p and the set of keywords K, is then extended with new child nodes having as keywords the intersection of $uexp(n,p)$ and $q \setminus K$. 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 query 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, query q is indexed in this subtrie under all its keywords, and node m now indexes two query identifiers, namely q and p. Otherwise, if some keywords of q are not contained in $K \cup uexp(n,p)$, then the common keywords are expanded to trie nodes to create a path from node n to node m, and node m will store two new $uexp$ lists, namely $uexp(m,p)$ and $uexp(m,q)$. 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 $id(n)$.

Example 2. Fig. 1(b) shows the forest of tries created when inserting the queries $q_1, \ldots, q_{10}$ (shown in Fig. 1(a)) in that order. The first query q_1 creates trie T_1 and is indexed under the (randomly chosen) keyword games. The second query q_2 does not create a new trie, but, is indexed under T_1, since this maximises its node reusability $nr(q_2,T_1)$. The third query q_3 cannot be indexed in T_1, since it does not contain the keyword games, thus, a new trie T_2 is created and q_3 is indexed under the keyword olympic. Similarly, STAR inserts the remaining queries.

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

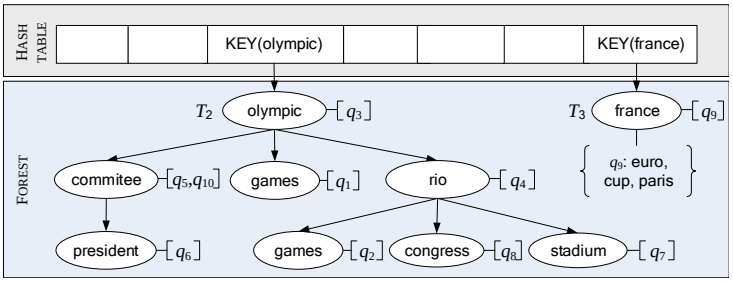
During the indexing phase, Algorithm STAR collects statistics about the frequency of occurrence of keywords in queries, which are then utilised in the reorganisation phase (described in the next section).

3.2 Query reorganisation phase

Reorganisation is a periodic procedure that initiates at given time intervals, after a given number of query insertions, or when a criterion is met (e.g., when a certain percentage of queries have low node utilisation). Any of the above options may be implemented in the context of Algorithm STAR; for simplicity we have selected to reorganise the query database after the insertion of Q new queries. It should be noted that, contrary to Algorithm RETRIE [4] which relocates only

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

(a)



(b)

Fig. 2. (a) Query scores and (b) the data structure after the reorganisation phase of Algorithm STAR-LR for the queries of Fig. 1(a)

TABLE 1
Statistics of keywords gathered from the keywords of queries in Fig. 1

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

poorly indexed queries, STAR reorganises only those queries inserted since the last database reorganisation.

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

Example 3. Let us consider the queries and the index of Fig. 1, and assume that Algorithm STAR collected the frequencies illustrated in Table 1. According to these frequencies, we have $score(q_1) = sprt(\mathsf{olympic}) + sprt(\mathsf{games}) = 9 + 2 = 11$. Fig. 2(a) shows the total score of queries $q_1, ..., q_{10}$.

To maintain $sprt$ (resp. $score$), Algorithm STAR utilises a hash table, denoted by $statSprt$ (resp. $statSco$), that contains keywords k (resp. queries q) as keys and support of keywords $sprt(k)$ (resp. scores of queries $score(q)$) as values (similarly to Table 1). STAR employs these statistics to reorganise newly inserted queries in the query database as follows. STAR re-indexes the newly inserted queries $\{q_1, \ldots, q_s\}$ from the existing forest by sorting them in descending order according to $score(q_i)$, where $score(q_1) \geq score(q_i) \geq score(q_s)$. Thus, queries with the highest score are inserted first in the forest; this variation of STAR is identified as STAR-H. Respectively, Algorithm STAR could re-index all the newly inserted queries by sorting them in ascending order according to $score(q_i)$, where $score(q_1) \leq score(q_i) \leq score(q_s)$. Thus, queries with the lowest score are inserted first in the forest; this variation of STAR is identified as STAR-L. According to STAR-H, the new order of insertion will be $q_2, q_7, q_8, q_4, q_6, q_5, q_{10}, q_1, q_3, q_9$, while STAR-L uses the inverse order. As we will show in Section 4, the problem of query insertion order is important; the first queries to be indexed define the *clustering opportunities* for the subsequent ones.

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

The above options that define the indexing order of the queries and influence the construction of tries by using keyword frequency, provide four distinct variations for Algorithm STAR, identified as STAR-HF, STAR-HR, STAR-LF, and STAR-LR, each with its own characteristics. All these options, along with the filtering performance of each variation are discussed in Section 4. Fig. 2(b) shows the FOREST data structure after the reorganisation phase of Algorithm STAR-LR is executed. Notice that compared to the previous forest of Fig. 1, all queries are now indexed under two tries and utilise one trie node less (10 trie nodes in the initial forest vs. 9 trie nodes in the reorganised one).

Finally, notice that node reusability of q_1 is reduced from $nr(q_1, T_1) = \frac{2}{2}$ (Fig. 1) to $nr(q_1, T_2) = \frac{1}{2}$ (Fig. 2), while it remained the same for the rest of the queries. As we will

```
Algorithm: FILTER
Input: A document d = {k_1, ..., k_t}
Output: A list of queries match = {q_i, ..., q_j}
1  match ← Null;
   // Use a linked list for distinct keywords of d
2  foreach distinct keyword k ∈ d do
3      foreach trie T with root(T) = k ∈ d do
4          foreach node n ∈ T do
5              if kwrd(n) ∈ d then // Use a hash table
                 representation of d to check this
6                  if uexp(n, q) ⊆ d then
                       // The queries stored here match d
7                      match ← match ∪ id(n);
8                      n ← children(n); // Traverse trie in DFS
9              else
10                 prune n; // Else do not search in sub-tries
11 return match
```

demonstrate in Section 4, the most important factor for the filtering performance of the algorithms is the nature of the created forest and the way it is constructed, rather than trie compactness and high node reusability values.

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

3.3 Filtering incoming documents

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

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

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

3.4 Parallelisation of the filtering process

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

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

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

Both approaches extend STAR to multi-core environments and allow it to exploit shared memory capabilities of modern hardware to perform faster filtering.

3.5 Competitors

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

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

The main differences between Algorithms STAR and RETRIE are as follows. Algorithm STAR *(i)* uses a query indexing mechanism that builds tries based on statistical information about query words, *(ii)* destroys poorly performing tries and creates new ones based on query scores, *(iii)* repositions newly inserted queries only, and *(iv)* emphasises trie shape rather than trie compactness. Following our running example, Algorithm RETRIE would not reposition any query in the forest of Fig. 1, although better alternatives are available (see Fig. 2).

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

3.6 Supporting richer query languages

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

Attributes may be introduced by creating a hash table (that will use the attribute name as key) and a forest of tries (like the ones presented in Figs. 1 and 2) for each attribute A in the data model. Similarly, we may use one table per attribute to maintain statistics of words in each attribute. Reorganisation will then be executed independently for each attribute; when a document d is published, the filtering procedure for Algorithm STAR is modified so that for each attribute A in the document and for each keyword k in A, the trie in the forest of A that has the word k as root is traversed using the filtering algorithm of Section 3.3. Finally, list $id(n)$ in each trie node n gives implicitly all queries that match d for attribute A; thus, a query q will match a document d if q matches all attributes of d.

Different types of proximity formulas (e.g., the '*' operator of Google or proximity formulas of arbitrary distance intervals as in [35]) may also be easily supported by STAR. The words that are operands in proximity formulas are stored in the forest of tries (since proximity is a stricter form of conjunctive queries), while the distance intervals are stored in a separate data structure. Proximity formulas are initially evaluated as conjunctive queries, and the satisfaction of word order and distance is evaluated separately using an algorithm like [24]. Other useful query components, like equality, disjunction, and negation, are also straightforward to support in the current indexing scheme. Finally, notice that handling more complex semantics is possible through semi/fully-automated query generation as in recommender systems, or through query expansion/augmentation techniques with the aid of taxonomies or dictionaries.

4 EXPERIMENTAL EVALUATION

In this section, we present a series of experiments that compare the filtering performance of Algorithm STAR with the trie-based Algorithms RETRIE [4] and TREE [5].

4.1 Experimental setup

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

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

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

Two continuous query sets for this corpus were synthetically generated similarly to [4], [5] and will be referred to as the *general* and the *focused* query collection.

The *general query collection* contains queries formed by conjunctions of different terms; each term conjunct is selected equiprobably among the multi-set of words forming the *DBpedia* corpus vocabulary ($762K$) and the set of Wikipedia document titles. Due to the nature of the *DBpedia* corpus and the corresponding vocabulary size, the constructed queries are expected to cover a wide variety of topics and, thus, share few common words between them. This restricts clustering opportunities and makes this setting a stress test for the filtering performance of the algorithms, as they are forced to identify and exploit the few commonalities between the indexed queries. For this query set, we select $50K$ documents' extended abstracts from *DBpedia* and use them as the incoming documents.

The *focused query collection* is constructed by selecting $50K$ thematically related extended abstracts from *DBpedia* and using the $46K$ distinct words appearing in those documents. As these queries become more focused and the vocabulary of the query database is restricted, more clustering opportunities appear. In this setting, the performance of the different algorithms is expected to be similar, as all will exploit the many clustering opportunities offered. Notice that the $50K$ incoming documents utilised in this section are the same ones used in the general query collection, since we aim to study the behaviour of our algorithms when varying the query set.

The ClueWeb09 corpus. The second dataset used in our experiments is the *ClueWeb09* collection (http://lemurproject.org/clueweb09/) that contains $1B$ web pages from which we randomly selected $50K$ as incoming documents.

The TREC *Million Query Track* dataset (http://trec.nist.gov/data/million.query.html) is comprised of one-time queries submitted to a search engine along with their relevance assessments for the *ClueWeb09* documents. This query set was used, in our setup, as continuous queries for the *ClueWeb09* corpus. The original query collection included $60K$ entries that were cleaned from arithmetic and single-term queries (as they do not fit a textual IF scenario), resulting into a final collection size of $50K$ entries. Notice that this query set is very limited in size compared to the typical expected workload of an IF system and our baseline values for synthetic query sets. However, the results for this collection provide an overview of the performance of the algorithms in a real-life setup.

Algorithm configuration. There is a number of system parameters, which affect the performance of the presented algorithms, that have to be determined and set. For our

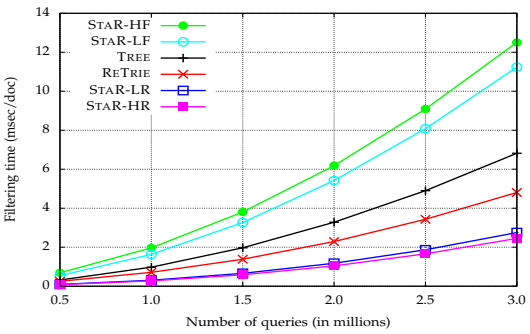


Fig. 3. Filtering time for $Q_L = 5$

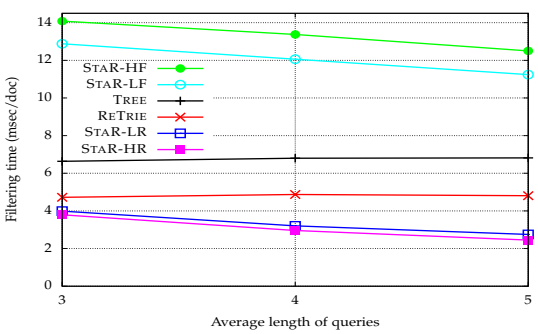


Fig. 4. Filtering time for $DB = 3M$

evaluation, we use a clustering ratio of 0.8 for RETRIE (selected after an exhaustive scan of all possible parameter values), while query reorganisation for under-clustered queries is invoked every $I_Q = 125K$ query insertions. Regarding STAR, the reorganisation of each variant is invoked when $I_Q = 500K$ new queries are indexed for STAR-LF and STAR-LR, and when $I_Q = 125K$ and $I_Q = 250K$ for STAR-HF and STAR-HR respectively. To measure the performance of the filtering process, we utilise the general query collection in our experimental setup to compare the performance of the two proposed approaches. Finally, for the parallelisation of the filtering process we utilised a set of 6 threads available in the processor. The baseline values for each tunable parameter in the experimental evaluation are: *(i)* average query length $Q_L = 5$, *(ii)* average document length $D_L = 53$, *(iii)* number of incoming queries $I_Q = 500K$, (iv) number of incoming documents $I_D = 50K$, *(v)* query database size $DB = 3M$, and *(vi)* threads used in the filtering process $Th = 1$. For more details about the parameter setting we refer the interested reader to [4], [5].

Metrics employed. In our evaluation, we use the *number of nodes* in the forest of tries to measure the quality of clustering for each algorithm. Additionally, we use *filtering time* to measure the filtering performance of each algorithm, i.e., the amount of time needed to locate all continuous queries satisfied by an incoming document. We also present the algorithms' *throughput* to study their performance as the query database size increases, i.e., the amount of filtered data per second. Finally, we measure *insertion* and *reorganisation time*, to identify the time needed to index and reorganise queries and give the *memory requirements* for each algorithm.

Technical configuration. All algorithms were implemented in C++ and an off-the-shelf PC (Core $i7$ $3.6GHz$, $8GB$ RAM, Ubuntu Linux 14.04) was used. For the parallelisation of the filtering process, the C++ library `<thread>` 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.

4.2 Results for the general query collection

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

Comparing filtering time. Fig. 3 shows the average time in milliseconds needed to filter a collection of $I_D = 50K$ documents with $D_L = 53$ words against a database of different size when indexing queries with $Q_L = 5$ terms. Observe that filtering time increases for all algorithms as the query database size increases. Algorithms STAR-LR and STAR-HR (that store rare words near the roots of tries) achieve the lowest filtering times, suggesting better performance than their counterparts STAR-HF and STAR-LF (that store frequent words near the roots) and competitors RETRIE and TREE. Additionally, Algorithms STAR-HF, STAR-LF, and TREE are more sensitive to query database size changes than the rest of their competitors since frequent words are stored near trie roots which requires traversing more tries at filtering time. Specifically, STAR-LR filters incoming documents 74.75% faster than RETRIE and 147% faster than TREE. Moreover, STAR-HR outperforms RETRIE by 96.14% and TREE by 178.15%. On the other hand, Algorithms STAR-HF and STAR-LF are slower than Algorithms RETRIE and TREE. More specifically, Algorithm STAR-HF needs 61.5% more time to filter an incoming document than RETRIE and 45.46% more time than TREE. Finally, STAR-LF shows similar performance, executing the filtering process 57.22% slower than RETRIE and 39.33% slower than TREE.

TABLE 2
Filtering scalability in a big data setup for variants of Algorithm STAR

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

Fig. 4 shows the filtering time for queries of different length. It is worth noting that all algorithms (except TREE that remains unaffected) improve their filtering performance when the query size is increased, as longer queries provide better indexing alternatives and more opportunities for pruning of tries at filtering time. In addition, STAR-HF and STAR-LF present a high decrease in filtering time (which is attributed to their poor filtering performance that has more margin for improvement) than STAR-HR, STAR-LR, and RETRIE when the query length is increased.

Table 2 reports the results for a stress test of the proposed algorithms under a big data setup. To do so, we conducted a filtering experiment in an Intel Xeon $2.7GHz$ server with $264GB$ RAM, using up to two orders of magnitude more queries and the whole *DBpedia* document collection as the stream of documents to be filtered. The resulting experiment ended up filtering a stream of $3.7M$ documents (totalling an uncompressed size of $5.5GB$) against a query database

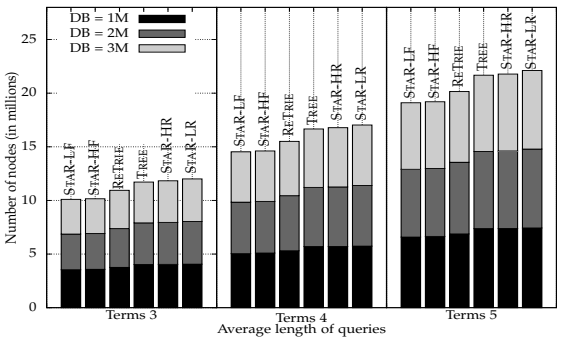


Fig. 5. Trie nodes created when varying Q_L and DB

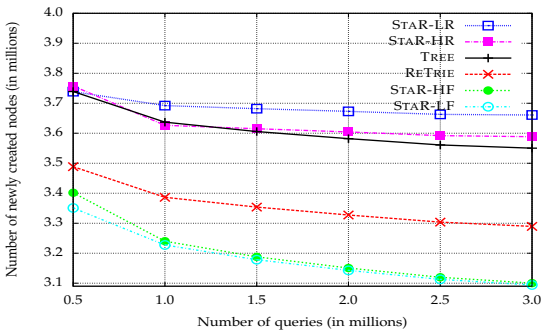


Fig. 6. Newly created trie nodes when inserting queries with $Q_L = 5$

of $100M$ queries (totalling a size of 7GB) for all variations of our algorithms. In the "Time" column of each algorithm we report the average filtering time per document for each query database, while in the "Increase" column we report the increase (in number of times) for the $10M$ and $100M$ queries against the base case of $1M$ queries. Our findings show that our solution is scalable: for a 10 (resp. 100) times increase in the query database size, the corresponding increase in filtering time of our best solution is no more than 0.4 (resp. 10) times.

Comparing trie compactness. Fig. 5 shows the number of created nodes by each algorithm for databases containing 1, 2 and 3M queries of different lengths. Observe that StaR-LR, StaR-HR and Tree create relatively larger forests, while StaR-HF and StaR-LF create relatively smaller ones. Also, increasing the number of query length, from 3 to 5 terms on average, results in an increase in the forest size. Interestingly, the difference in trie nodes between the algorithms remains unchanged. This can be explained as follows. StaR-LR and StaR-HR create relatively large forests as they index rare words towards the roots of tries and frequent words towards the leaves. In this way, the lower levels of tries tend to repeat words with high frequency of occurrence, thus, creating many tries with higher fanout and lower node reusability. Contrary, Algorithms StaR-HF and StaR-LF create relatively small forests as they push the most frequent words towards the roots of the tries. This, creates more compact tries as many repeated words are indexed in the same trie node, thus, resulting in high node utilisation. Algorithm Tree creates large forests as it utilises a naive query placement technique and implements no reorganisation. Finally, Algorithm ReTrie creates an average-sized forest, as it focuses on query clustering rather than word statistics to index queries.

The results in Fig. 5 suggest that StaR-HF creates 4.94% (948K) less nodes than ReTrie and 12.88% (2.474M) less nodes than Tree. Similarly, StaR-LF creates 5.46% less nodes than ReTrie and 13.44% less nodes than Tree. On the other hand, StaR-HR creates 7.5% more nodes than ReTrie and 0.5% more nodes than Tree. Additionally, StaR-LR creates a larger forest by 8.86% nodes compared to ReTrie and by 1.96% nodes compared to Tree.

Fig. 6 shows the increase in forest size after the insertion and reorganisation of $500K$ queries with $Q_L = 5$. As expected, results are similar to those of Fig. 5, as all algorithms demonstrate the behaviour discussed above and rank with regard to node increase. Additionally, Algorithms StaR-HF, StaR-LF and ReTrie are less sensitive to query insertion, demonstrating a continuously decreasing number of newly created nodes compared to StaR-LR and StaR-HR. This is due to the exploitation of frequent words between queries and the construction of more compact tries compared to Algorithms StaR-LR and StaR-HR, where query clustering is affected by infrequent words.

Efficiency vs. compactness. Comparing the results of node creation and filtering time, we can infer that the nature of the forest and the way it is constructed has a more significant effect on filtering performance than trie compactness, since it is shown (Figs. 3, 4, 5 and 6) that the algorithms creating the less compact forests result in the lowest filtering times. This can be explained as follows. Algorithms ReTrie, StaR-HF, and StaR-LF tend to locate and group queries under common words and create compact forests with frequent words near the trie roots. At filtering time, incoming documents match many trie roots, thus, needing to traverse more tries to examine all possible query matches. Additionally, for each trie, the filtering process cannot prune many subtries, as the infrequent words are stored near the trie leaves. This results in the traversal of the whole trie structure all the way to the leaves of every subtrie to determine whether a query is relevant to the incoming document or not. Contrary, Algorithms StaR-LR and StaR-HR locate and store rare words in nodes closer to trie roots, thus, organising queries in subtries under their common (rare) words. This organisation of queries, prevents the filtering process to visit many tries, and prunes subtries much earlier, due to the low probability of a rare word being present in an incoming document. Notice also that the utilisation of query score allows us to enforce at indexing time a query order based on query importance. In this way, queries consisting of many words ordered from rare to frequent (i.e., Algorithm StaR-HR) are inserted first during the reorganisation phase allowing the creation of more tries. Those tries are then used as the guides that push frequent words further down the trie structure. This approach causes StaR-HR to be 10.9% more efficient in terms of filtering time when compared to StaR-LR.

Comparing filtering throughput. Fig. 7 shows the throughput in KB/sec needed to filter $I_D = 50K$ incoming documents against a query database of different size indexing queries with 5 terms. Notice that throughput decreases for all algorithms rather rapidly as the query database size increases. Algorithms StaR-HR and StaR-LR achieve higher

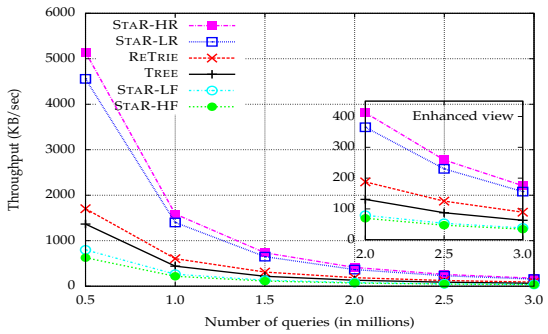


Fig. 7. Filtering throughput for $Q_L = 5$x

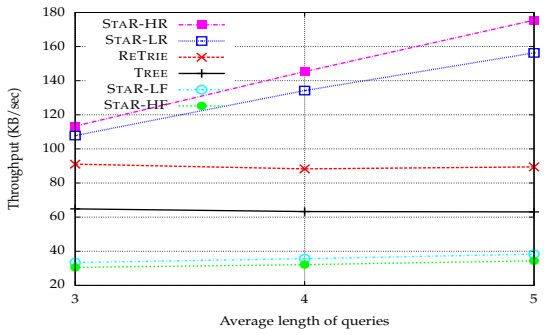


Fig. 8. Filtering throughput for $DB = 3M$

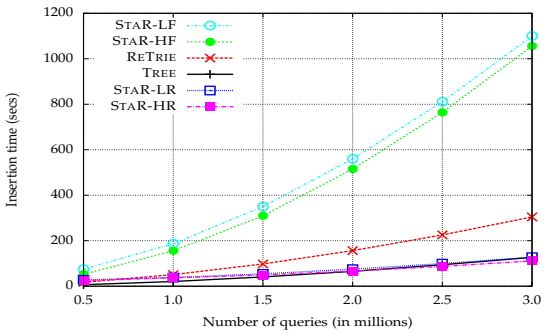


Fig. 9. Insertion time for queries with $Q_L = 5$

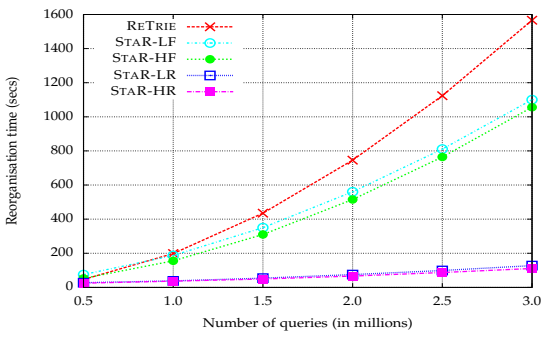


Fig. 10. Reorganisation time for queries with $Q_L = 5$

throughput than the other variants of STAR (i.e., STAR-HF and STAR-LF) and their competitors TREE and RETRIE. Observe the enhanced view area in Fig. 7 for a clearer presentation of throughput results for $[2M, 3M]$ queries. Fig. 8 shows the filtering throughput for queries of different lengths. As expected all algorithms (except TREE) exhibit an increase in filtering throughput as longer queries provide better indexing opportunities. Algorithms STAR-HR and STAR-LR present higher increase in terms of filtering throughput, which is attributed to the nature of tries they create, namely the pruning of tries at filtering time due to the existence of rare words near trie roots.

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

Comparing insertion time. In this section, we discuss the time needed to insert a query and reorganise databases of different sizes. Fig. 9 shows the time in seconds required to insert $I_Q = 500K$ queries with $Q_L = 5$ terms in databases of varying size. We observe that the insertion time of all algorithms increases with the query database size. Algorithms STAR-HF and STAR-LF need more time to insert new queries in the existing database since they need to examine more tries as possible indexing locations. This happens due to the nature of the tries, which index frequent words near the trie roots, thus, create more indexing opportunities. Algorithm RETRIE requires less time to insert new queries in the database than STAR-HF and STAR-LF, as it has less indexing opportunities. Next, Algorithm TREE requires less time to insert queries in the database as it places queries deterministically in the tries. Finally, Algorithms STAR-LR and STAR-HR tend to explore less candidate tries as rare words in trie roots exclude many clustering possibilities.

Comparing reorganisation time. In this section, we measure the time needed to reorganise $I_Q = 500K$ for varying database sizes. Fig. 10 presents the time needed to reorganise a query database for each of the presented algorithms (except TREE that does not consider any query index reorganisation). Algorithms RETRIE, STAR-LF and STAR-HF need more time to reorganise $I_Q = 500K$ queries compared to the rest of the examined algorithms, and the time needed increases as the query database increases. This can be explained as follows. As the database indexes new queries the clustering opportunities for queries increase. Algorithm RETRIE that aims at reorganising poorly clustered queries has to scan the whole query database to locate them and subsequently identify better indexing positions. Similarly, Algorithms STAR-LF and STAR-HF are affected by frequent words in the higher levels of the forest resulting in an extensive search on the query database at reorganisation time. Notice that, as the database grows in size, the re-indexing options increase, thus, resulting in increased reorganisation time. Contrary, Algorithms STAR-LR and STAR-HR are slightly affected by the increase in database size, due to the use of infrequent words near trie roots.

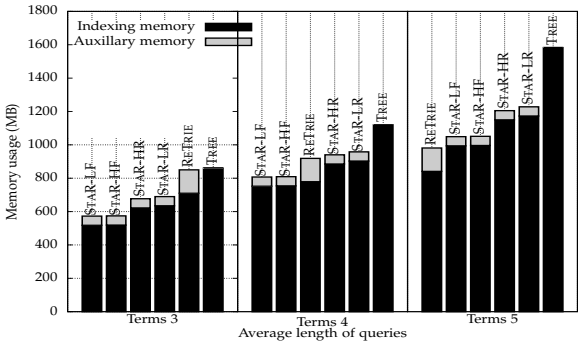


Fig. 11. Memory measurements for $DB = 3M$

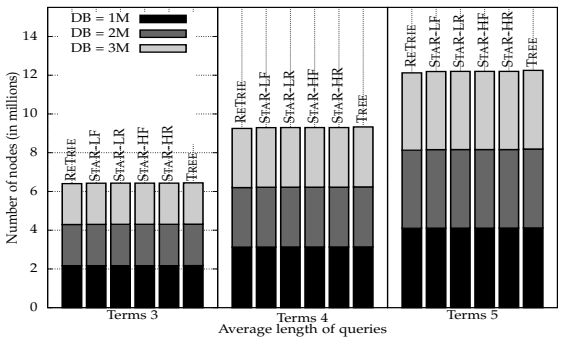


Fig. 12. Trie nodes created for the *focused query collection*

To test the efficacy of full query database reorganisation we have considered such a scenario for Algorithms ReTrie and StaR-LR (our fastest performing solution); in our setup both algorithms executed a complete reorganisation for a query database of $3M$ queries with $Q_L = 5$ terms. The total reorganisation times obtained were 68 minutes (a three orders of magnitude or 2626 times increase from the partial reorganisation solution) for Algorithm ReTrie and 24 minutes (a four orders of magnitude or 11673 times increase from the partial reorganisation solution) for Algorithm StaR-LR. The respective gain in filtering time was a 96% decrease for Algorithm ReTrie and a 95% decrease for Algorithm StaR-LR in filtering time. These results suggest that choosing a complete database reorganisation is an expensive choice with a small gain for a real world scenario where efficiency is of high importance, and will not be considered further.

Comparing memory usage. We have also executed experiments to specify memory requirements for each of the presented algorithms. Fig. 11 exhibits a good overview of the results for $DB = 3M$ queries and varying Q_L terms. For a database of $DB = 3M$ and $Q_L = 5$, Algorithm ReTrie has the lowest memory requirements needing 981 MB for storing the query database and all indexing components, while Algorithms StaR-LF and StaR-HF need approximately 1 GB of memory to store this information. Algorithms StaR-LR and StaR-HR need more than 1.1 GB of main memory due to the non-compact forests they create, and Algorithm Tree needs around 1.5 GB memory to store the query database. The excessive memory requirements of Algorithm Tree (compared to StaR and ReTrie) are explained as follows. Algorithm Tree creates a new node for every term that can not be indexed into an existing trie. In contrary, StaR makes use of the *uexp* structure (as described in Section 3.1), which allows the delay of the node creation, thus, sorting the word in a list of strings and resulting to less indexing memory requirements. Notice also, that StaR's variants and ReTrie have extra memory requirements in order to keep the appropriate data in auxiliary structures for their reorganisation phase; Algorithm ReTrie requires 150 MB and StaR's variants 50 MB extra (shown in Fig. 11).

4.3 Results for the *focused query collection*

In this set of experiments, we present the most interesting results concerning the *focused query collection* (described earlier in this section). Fig. 12 shows the number of nodes created by each algorithm for databases containing 1, 2 and

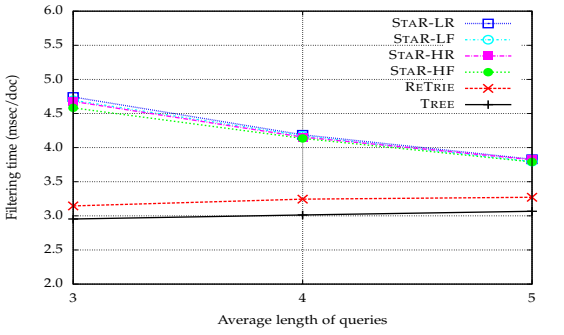


Fig. 13. Filtering time for the *focused query collection* for $DB = 3M$

3M queries of different lengths. Observe that all algorithms behave similarly in terms of nodes created; the differences in the sizes of the resulting forest among the examined algorithms did not exceed 1%. This was expected as the small vocabulary of the focused collection created many clustering opportunities for all algorithms. In such a setting it is not important where to index a given query, as most queries will eventually be well-clustered. Naturally, the differences in filtering time (shown in Fig. 13) are not significant (notice the small scale of the y-axis) with Algorithms Tree and ReTrie performing slightly better than the StaR variants.

The most interesting conclusions concerning the performance of the algorithms are extracted from the cross-comparison results of the examined query collections in Figs. 4 and 13 . By comparing the filtering performance of the examined algorithms, we observe that Tree, ReTrie, StaR-HF, and StaR-LF are very sensitive to vocabulary variations as the increase in filtering time is 122%, 46%, 229%, and 196% respectively when increasing the vocabulary size, for the same query database size and query length. On the other hand, Algorithms StaR-LR and StaR-HR present a decrease in filtering time, since word statistics for bigger vocabularies contain more information to be exploited. Finally, comparing the absolute filtering times for the two collections, we conclude that Algorithms StaR-LR and StaR-HR deliver a steady filtering efficiency independently of the vocabulary size used.

4.4 Result for the varying document length collections

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

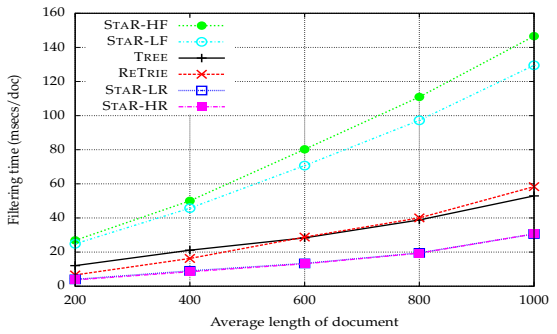


Fig. 14. Filtering time for $DB = 3M$ queries and $Q_L = 5$

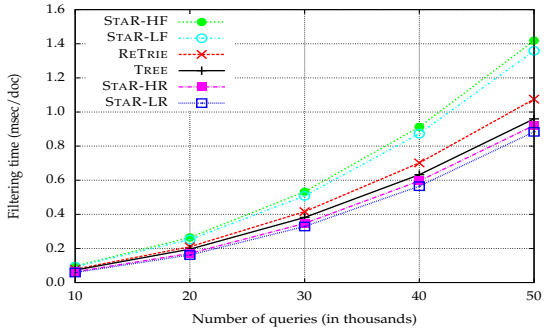


Fig. 15. Filtering time for the *Million Query Track* dataset

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

Fig. 14 shows the time in milliseconds needed to filter documents with varying average length, when storing $DB = 3M$ queries with average length of $Q_L = 5$ words. As expected, the filtering time increases with the increase in incoming documents length. All algorithms though, present the same behaviour discussed earlier in this section, except ReTrie that exhibits higher sensitivity to document size variation; as the average length of the incoming documents increases Algorithm ReTrie gradually needs more filtering time compared to Tree. This happens because Algorithms ReTrie, StaR-HF, and StaR-LF tend to group queries under common words and create forests with the majority of common words near the roots. Thus, due to larger document length, Algorithms ReTrie, StaR-HF, and StaR-LF are forced to visit the lower levels of the trie, i.e., near the leaves where the fan out is greater due to poor clustering. The presented results suggest that StaR-LR and StaR-HR perform 70% better in filtering time compared to ReTrie and Tree. These differences in filtering times hold for all sizes of document collections, allowing us to conclude that StaR-HR and StaR-LR present a steady filtering efficiency that remains relatively unaffected from the document size.

4.5 Results for the *Million Query Track* dataset

In this section, we present the most interesting results concerning the *Million Query Track* dataset (described earlier in this section). Fig. 15 presents the average time in milliseconds needed to filter a collection of $I_D = 50K$ incoming documents against a query database of increasing size. In this scenario, all algorithms exhibit a similar behaviour as in the general query collection presented earlier. Although, Algorithms StaR-HR and StaR-LR maintain their low filtering time in this query collection, Algorithm ReTrie preforms slightly worse, while Algorithm StaR-LR is faster compared to StaR-HR. More specifically, StaR-HR filters incoming documents 16.7% faster than ReTrie and 4.1% faster than Tree. Moreover, StaR-LR outperforms ReTrie by 21.6% and Tree by 8.5%. Finally, we observe that Algorithms StaR-HR and StaR-LR maintain their high filtering efficiency under real-life data and query sets (that are very different from the previous experimental setup).

The close performance of the StaR variants in the *Million Query Track* dataset is due to the small vocabulary of the queries and is in line with our previous findings (for the *focused query collection*) on how query vocabulary affects filtering time (see Fig. 13 and the explanation for this in Section 4.3). Moreover, the fact that ReTrie performs worse than Tree is attributed to the document length of the *ClueWeb09* corpus, which is much larger (1506 words) than the average document length in the *DBpedia* corpus. This ReTrie sensitivity to document size verifies our previous findings for the *DBpedia* corpus which are presented and discussed in Fig. 14 and Section 4.4 respectively.

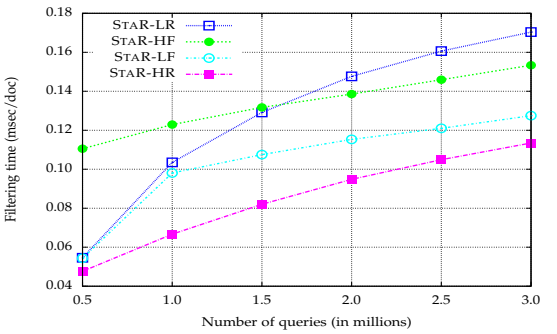


Fig. 16. Filtering time of DocPar, with $Q_L = 5$

4.6 Results for the parallelisation of filtering

In this section, we present the results concerning the two parallel filtering implementations of the four variations of Algorithm StaR, as described in Section 3.4.

Figs. 16 and 18 present the results for the DocPar approach. In this approach, each document that arrives at the system is assigned to an unoccupied thread. In our assessments we utilised 6 threads, thus allowing 6 documents to be filtered concurrently by the same processor. Fig. 16 shows the average time in milliseconds needed to filter the general query collection of $I_D = 50K$ documents with $D_L = 53$ words against database of different size when indexing queries with $Q_L = 5$ terms. Fig. 18 presents the filtering time for queries of different length; the variations of Algorithm StaR are significantly faster compared to their non-parallel counterparts (Figs. 3 and 4).

Figs. 17 and 18 present the results for the RootPar approach. In this approach, each thread is responsible for a set of roots present in the Forest. Thus, each root

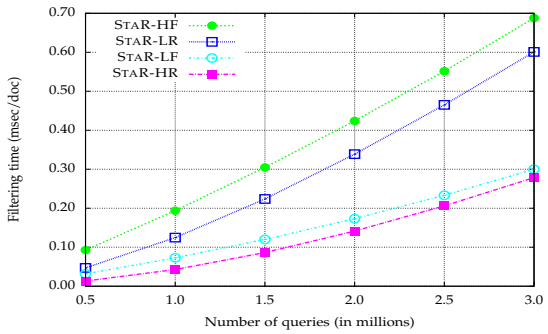


Fig. 17. Filtering time of RootPar, with $Q_L = 5$

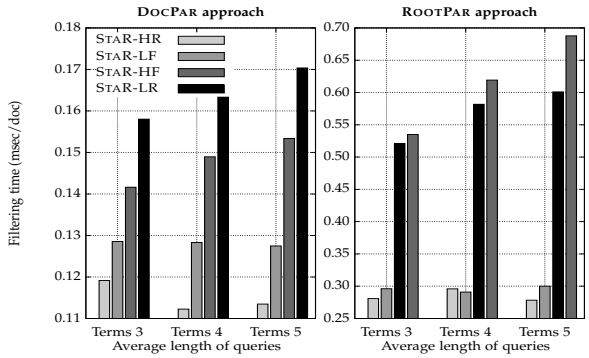


Fig. 18. Filtering time of DocPar and RootPar, with $DB = 3M$

traversal is executed from a different thread during the filtering of a document. In our assessments we utilised 6 threads, allowing us to split the total number of roots into 6 even parts and assign each part to a single thread. Fig. 17 shows the average time in milliseconds needed to filter the general query collection of $I_D = 50K$ documents with $D_L = 53$ words against database of different size when indexing queries with $Q_L = 5$ terms. In Fig. 18 we give the filtering time for queries of different length; the variations of Algorithm StaR are significantly faster compared to their non-parallel counterparts (Figs. 3 and 4).

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

4.7 Effectiveness comparison

Fig. 19 presents a proof-of-concept effectiveness evaluation and cross-comparison for the Boolean and VSM models. Our intention here is not to perform a full-scale study for the effectiveness of the two models, but rather to highlight that important publications are delivered to the users despite the crude nature of Boolean semantics.

To do so, we relied on the relevance judgements between queries and documents available at the TREC website, on official TREC tools (e.g., trec_eval), and publicly

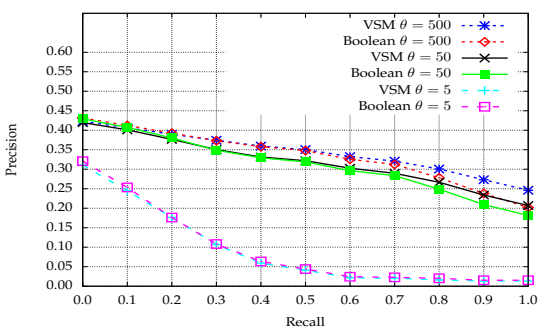


Fig. 19. Comparing the effectiveness of Boolean and VSM models

available Lemur/Lucene libraries for parsing and preprocessing *ClueWeb09* WARC files. To derive the plots of Fig. 19 we analysed the notifications produced under each model, and set the cut-off threshold $\theta = 5$, $\theta = 50$ and $\theta = 500$ to consider the top-5, top-50 and top-500 most relevant notifications respectively. We used the typical (log normalisation) tf.idf weighting scheme and vector normalisation for VSM for both documents and queries (usually referred to as ltc.ltc [1]). For the Boolean model we utilised term frequencies (i.e., word counts) as a score function to make the notification lists of the two models comparable. Fig. 19 presents the 11-point interpolated precision/recall graph, micro-averaged for the 686 TREC queries that had relevance judgements against 34013 different *ClueWeb09* documents.

Notice that the effectiveness between the two models is very similar when recall is low and comparable for high recall values, whereas an increase in θ results in better effectiveness. The reported Mean Average Precision values for $\theta = 5$ was 0.082, for $\theta = 50$ was 0.2763, and for $\theta = 500$ was 0.2972 for the Boolean model, and 0.0785, 0.2821, and 0.3055 for VSM respectively. Similarly, the reported NDCG values for $\theta = 5$, $\theta = 50$, and $\theta = 500$ were 0.1427, 0.4332, and 0.4695 (Boolean model) and 0.1380, 0.4508, and 0.4976 respectively (VSM). For more details on the effectiveness of the Boolean model and on how it compares against other alternatives the interested reader is referred to [36].

4.8 Summary of results

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

the delay in node creation; this strategy results in utilising each newly created node by as many queries as possible.

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

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

5 Outlook

Interesting directions for future research involve *(i)* the adaptation of automata/graph-based techniques as in [10], [30] to Boolean IF and their comparison against trie-based approaches, *(ii)* the extension of RDF-based data and SPARQL-based query models with text capabilities, and *(iii)* the construction of IF ontology systems that will be able to filter ontology data in a streaming fashion.

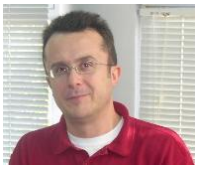
References

[1] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.
[2] P. Fischer and D. Kossmann, "Batched Processing for Information Filters," *ICDE*, 2005.
[3] C. Tryfonopoulos, M. Koubarakis, and Y. Drougas, "Filtering Algorithms for Information Retrieval Models with Named Attributes and Proximity Operators," *ACM SIGIR*, 2004.
[4] ——, "Information filtering and query indexing for an information retrieval model," *ACM TOIS*, 2009.
[5] T. Yan and H. Garcia-Molina, "Index structures for selective dissemination of information under the boolean model," *ACM TODS*, 1994.
[6] J. Yochum, "A High-Speed Text Scanning Algorithm Utilising Least Frequent Trigraphs," *IEEE SNDC*, 1985.
[7] T. Bell and A. Moffat, "The Design of a High Performance Information Filtering System," *ACM SIGIR*, 1996.
[8] T. Yan and H. Garcia-Molina, "Index Structures for Information Filtering under the Vector Space Model," *ICDE*, 1994.
[9] J. Callan, "Document Filtering With Inference Networks," *ACM SIGIR*, 1996.
[10] W. Rao, L. Chen, S. Chen, and S. Tarkoma, "Evaluating continuous top-k queries over document streams," *World Wide Web*, 2014.
[11] M. Franklin and S. Zdonik, ""Data in Your Face": Push Technology in Perspective," *SIGMOD Record*, 1998.
[12] M. Altinel, D. Aksoy, T. Baby, M. Franklin, W. Shapiro, and S. Zdonik, "DBIS-toolkit: Adaptable Middleware for Large-scale Data Delivery," in *ACM SIGMOD*, 1999.
[13] 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," *ACM SIGMOD*, 2001.
[14] B. Nguyen, S. Abiteboul, G.Cobena, and M. Preda, "Monitoring XML Data on the Web," *ACM SIGMOD*, 2001.
[15] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient Filtering in Publish Subscribe Systems Using Binary Decision Diagrams," in *ICSE*, 2001.
[16] M. Sadoghi and H. Jacobsen, "Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space," *ACM SIGMOD*, 2011.
[17] ——, "Analysis and optimization for boolean expression indexing," *ACM TODS*, 2013.
[18] M. Altinel and M. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," *VLDB*, 2000.
[19] Y. Diao, M. Altinel, M. Franklin, H. Zhang, and P. Fischer, "Path Sharing and Predicate Evaluation for High-performance XML Filtering," *ACM TODS*, 2003.
[20] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions," *ICDE*, 2002.
[21] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata," *ICDT*, 2003.
[22] D. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1973.
[23] T. Yan and H. Garcia-Molina, "The SIFT Information Dissemination System," *ACM TODS*, 1999.
[24] M. Koubarakis, C. Tryfonopoulos, P. Raftopoulou, and T. Koutris, "Data models and languages for agent-based textual information dissemination," in *CIA*, 2002.
[25] M. Morita and Y. Shinoda, "Information Filtering Based on User Behaviour Analysis and Best Match Text Retrieval," *ACM SIGIR*, 1994.
[26] Y.-I. Chang, J.-H. Shen, and T.-I. Chen, "A Data Mining-Based Method for the Incremental Update of Supporting Personalized Information Filtering," *J. of Inf. Sci. Eng.*, 2008.
[27] D. Hull, J. Pedersen, and H. Schütze, "Method Combination For Document Filtering," *ACM SIGIR*, 1996.
[28] Y. Li, X. Zhou, P. Bruza, Y. Xu, and R. Lau, "A two-stage text mining model for information filtering," *CIKM*, 2008.
[29] P. Foltz and S. Dumais, "Personalized Information Delivery: An Analysis of Information Filtering Methods," *CACM*, 1992.
[30] N. Nanas, M. Vavalis, and A. D. Roeck, "A network-based model for high-dimensional information filtering," *ACM SIGIR*, 2010.
[31] J. Callan, "Learning While Filtering Focuments," *ACM SIGIR*, 1998.
[32] Y. Zhang and J. Callan, "Maximum likelihood estimation for filtering thresholds," *ACM SIGIR*, 2001.
[33] T. Hofmann, "Collaborative filtering via gaussian probabilistic latent semantic analysis," *ACM SIGIR*, 2003.
[34] R. Y. K. Lau, P. D. Bruza, and D. Song, "Towards a belief-revision-based adaptive and context-sensitive information retrieval system," *ACM TOIS*, 2008.
[35] M. Koubarakis, S. Skiadopoulos, and C. Tryfonopoulos, "Logic and Computational Complexity for Boolean Information Retrieval," *IEEE TKDE*, 2006.
[36] G. Salton, E. A. Fox, and H. Wu, "Extended boolean information retrieval," *CACM*, 1983.

Lefteris Zervakis is currently pursuing a Ph.D. degree in Computer Science at the Dept. of Informatics and Telecommunications, University of the Peloponnese, working on the areas of data/information management, RDF data, and graph databases.

Christos Tryfonopoulos is an Assistant Professor at the Dept. of Informatics and Telecommunications, University of the Peloponnese. He has published more than 50 research papers in the areas of data/information management, large-scale distributed systems, and digital libraries.

Spiros Skiadopoulos is an Associate Professor at the Dept. of Informatics and Telecommunications, University of the Peloponnese. He has published more than 50 research papers in the areas of databases, data warehouses, knowledge representation, and artificial intelligence.



Manolis Koubarakis is a Professor at the Dept. of Informatics and Telecommunications, National and Kapodistrian University of Athens. He has published more than 130 research papers in the areas of semantic web and linked data, large-scale distributed systems, data and knowledge-based systems, and constraint programming.