Filtering Algorithms for Information Retrieval Models with Named Attributes and Proximity Operators* †

Christos Tryfonopoulos
Dept. of Electronic and Computer Engineering
Technical University of Crete
GR73100 Chania, Greece
trifon@intelligence.tuc.gr

Manolis Koubarakis
Dept. of Electronic and Computer Engineering
Technical University of Crete
GR73100 Chania, Greece
manolis@intelligence.tuc.gr

Yannis Drougas
Dept. of Computer Science and Engineering
Univ. of California Riverside
CA 92521, USA
drougas@cs.ucr.edu

ABSTRACT

In the selective dissemination of information (or publish/ subscribe) paradigm, clients subscribe to a server with continuous queries (or profiles) that express their information needs. Clients can also publish documents to servers. Whenever a document is published, the continuous queries satisfying this document are found and notifications are sent to appropriate clients. This paper deals with the filtering problem that needs to be solved efficiently by each server: Given a database of continuous queries *db* and a document *d*, find all queries $q \in db$ that match *d*. We present data structures and indexing algorithms that enable us to solve the filtering problem efficiently for large databases of queries expressed in the model $\mathcal{AWP}$ which is based on named attributes with values of type text, and word proximity operators.

1. INTRODUCTION

In the *selective dissemination of information (SDI or publish/subscribe) paradigm*, clients subscribe to a server with *continuous queries* or *profiles* that are expressed in some well-defined language and capture their information needs. Clients can also publish *documents* to servers. When a document is published, the continuous queries satisfying the document are found and notifications are sent to appropriate clients.

This paper deals with the *filtering problem* that needs to be solved efficiently by each server: Given a database of continuous queries *db* and a document *d*, find all queries $q \in db$ that match *d*. This functionality is crucial for a server because we expect deployed SDI systems to handle millions of client queries. We concentrate on selective dissemination of *textual information* using the data model $\mathcal{AWP}$ originally presented in [13, 12]. Data model $\mathcal{AWP}$ is based on *named attributes* with values of type *text*, and its query language includes attributes, comparison operators "equals" and "contains" and word proximity operators from the Boolean model of Information Retrieval (IR) [3]. By using linguistically motivated concepts such as *word* instead of arbitrary strings, $\mathcal{AWP}$ strives to be useful to certain applications e.g., alert systems for digital libraries or other commercial systems where similar models are supported already for retrieval. In work presented in [11] and [12], we discuss the distributed alert system DIAS and its ancestor, the peer-to-peer system P2P-DIET[1] [10] that uses $\mathcal{AWP}$ as its *meta-data* model for describing and querying digital resources and a new filtering algorithm named BestFitTrie for matching incoming documents against stored continuous queries.

In this paper we develop and evaluate efficient main-memory algorithms that are able to filter millions of continuous $\mathcal{AWP}$ queries in just a few hundred milliseconds. The filtering algorithms we present are the first in the literature that deal with IR-based models like $\mathcal{AWP}$. The algorithms closest to ours are the ones employed in the Boolean version of SIFT [16] where documents are free text and queries are conjunctions of keywords. SIFT has been the inspiration for this work and the results presented in Sections 3 and 4 extend and improve the results of [16]. In particular, we evaluate experimentally algorithms BF, SWIN and PrefixTrie that are extensions of the algorithms BF, Key and Tree of [16] for the model $\mathcal{AWP}$. We also discuss in detail the new algorithms BestFitTrie and LCWTrie as alternatives to PrefixTrie, and compare them under various experimental settings.

Work on recent filtering algorithms for XML-based query languages [5, 2, 9] is complementary to ours. The query languages of [5, 2, 9] *cannot* express word proximity as in $\mathcal{AWP}$ and the same is true for other W3C XML query languages. The recent W3C working draft [14] and papers like [8, 1, 4] are expected to pave the way for the introduction of IR concepts in XQuery/XPath. Our work on $\mathcal{AWP}$ serves a similar goal but we have chosen to work with the simple concept of a flat document interpreted under the Boolean model of IR, instead of XML and the vector space model.

*This work was supported in part by the European Commission projects DIET (5th Framework Programme IST/FET) and Evergrow (6th Framework Programme IST/FET). Christos Tryfonopoulos is partially supported by a Ph.D. fellowship from the program Heraclitus of the Greek Ministry of Education.

†This work is a modified version of the paper to be presented at the 27th International ACM SIGIR 2004 Conference.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
.

[1] http://www.intelligence.tuc.gr/p2pdiet

The main ideas of this paper can be transferred to languages such as [8, 1, 4] and this is where we concentrate our current efforts.

The rest of the paper is organized as follows. Section 2 presents the model $\mathcal{AWP}$ and Section 3 presents the filtering algorithms we developed. Section 4 gives a flavor of our query creation methodology and evaluates the algorithms experimentally under various different parameters. Finally, Section 5 hints at our current work.

2. THE DATA MODEL $\mathcal{AWP}$

In [11] we present the data model $\mathcal{AWP}$ for specifying queries and *textual* resource meta-data in SDI systems. $\mathcal{AWP}$ is based on the concept of *named attributes* with values of type *text*. The query language of $\mathcal{AWP}$ offers *Boolean* and *proximity operators* on attribute values as in the work of [3] which is based on the Boolean model of Information Retrieval (IR).

Syntax. Let $\mathbf{\Sigma}$ be a finite *alphabet*. A *word* is a finite non-empty sequence of letters from $\mathbf{\Sigma}$. Let $\mathcal{V}$ be a (finite or infinite) set of words called the *vocabulary*. A *text value* s of length n over vocabulary $\mathcal{V}$ is a total function $s : \{1, 2, \ldots, n\} \rightarrow \mathcal{V}$.

Let $\mathcal{I}$ be a set of *(distance) intervals* $\mathcal{I} = \{[l, u] : l, u \in \mathbb{N}, l \geq 0 \text{ and } l \leq u\} \cup \{[l, \infty) : l \in \mathbb{N} \text{ and } l \geq 0\}$. A *proximity formula* is an expression of the form $w_1 \prec_{i_1} \cdots \prec_{i_{n-1}} w_n$ where $w_1, \ldots, w_n$ are words of $\mathcal{V}$ and $i_1, \ldots, i_n$ are intervals of $\mathcal{I}$. Operators $\prec_i$ are called *proximity operators* and are generalizations of the traditional IR operators kW and kN [3]. Proximity operators are used to capture the concepts of *order* and *distance* between words in a text document. The proximity word pattern $w_1 \prec_{[l,u]} w_2$ stands for "word w_1 is *before* w_2 and is separated by w_2 by *at least* l and *at most* u *words*". The interpretation of proximity word patterns with more than one operator $\prec_i$ is similar. A *word pattern* over vocabulary $\mathcal{V}$ is a conjunction of words and proximity formulas. An example of a word pattern is $applications \wedge selective \prec_{[0,0]} dissemination \prec_{[0,3]} information$.

Let $\mathcal{A}$ be a countably infinite set of attributes called the *attribute universe*. In practice attributes will come from *namespaces* appropriate for the application at hand e.g., from the set of Dublin Core Metadata Elements[2].

A *document* d is a set of attribute-value pairs (A, s) where $A \in \mathcal{A}$, s is a text value over $\mathcal{V}$, and all attributes are *distinct*. The following set of pairs is a document:

$$\{\ (AUTHOR, \text{“}John\ Smith\text{”}),\\ (TITLE, \text{“}Selective\ dissemination\ of\ information\ in\ \ldots\text{”}),\\ (ABSTRACT, \text{“}In\ this\ paper\ we\ show\ that\ \ldots\text{”})\ \}$$

A *query* is a conjunction of the form

$$A_1 = s_1 \wedge \ldots \wedge A_n = s_n \wedge B_1 \sqsupseteq wp_1 \wedge \ldots \wedge B_m \sqsupseteq wp_m$$

where each $A_i, B_i \in \mathcal{A}$, each s_i is a text value and each wp_i is a word pattern. The following formula is a query:

$$AUTHOR = \text{“}John\ Smith\text{”} \wedge\\ TITLE \sqsupseteq (selective \prec_{[0,0]} dissemination \prec_{[0,3]}\\ information) \wedge peer\text{-}to\text{-}peer$$

Semantics. The semantics of $\mathcal{AWP}$ have been defined in [11] and will not be presented here in detail. It is straightforward to define when a document d *satisfies* an atomic formula of the form $A = s$ or $B \sqsupseteq wp$, and then use this notion to define when d satisfies a query [11]. The example document given above satisfies the example query.

[2]http://purl.org/dc/elements/1.1/

3. FILTERING ALGORITHMS

In this section we present and evaluate four main memory algorithms that solve the filtering problem for *conjunctive queries* in $\mathcal{AWP}$. Because our work extends and improves previous algorithms for SIFT [16], we adopt terminology from SIFT in many cases.

3.1 The Algorithm BestFitTrie

BestFitTrie uses two data structures to represent each published document d: the *occurrence table* $OT(d)$ and the *distinct attribute list* $DAL(d)$. $OT(d)$ is a hash table that uses words as keys, and is used for storing all the attributes of the document in which a specific word appears, along with the positions that each word occupies in the attribute text. $DAL(d)$ is a linked list with one element for each distinct attribute of d. The element of $DAL(d)$ for attribute A points to another linked list, the *distinct word list* for A (denoted by $DWL(A)$) which contains all the distinct words that appear in $A(d)$.

To index queries BestFitTrie utilises an array, called the *attribute directory* (AD), that stores pointers to word directories. AD has one element for each distinct attribute in the query database. A *word directory* $WD(B_i)$ is a hash table that provides fast access to roots of *tries* in a *forest* that is used to organize *sets of words* – the set of words in wp_i (denoted by $words(wp_i)$) for each atomic formula $B_i \sqsupseteq wp_i$ in a query. The proximity formulas contained in each wp_i are stored in an array called the *proximity array* (PA). PA stores pointers to trie nodes (words) that are operands in proximity formulas along with the respective proximity intervals for each formula. There is also a hash table, called *equality table* (ET), that indexes all text values s_i that appear in atomic formulas of the form $A_i = s_i$.

When a new query q of the form given above arrives, the index structures are populated as follows. For each attribute $A_i, 1 \leq i \leq n$, we hash text value s_i to obtain a slot in ET where we store the value A_i. For each attribute $B_j, 1 \leq j \leq m$, we compute $words(wp_j)$ and insert them in one of the tries with roots indexed by $WD(B_j)$. Finally, we visit PA and store pointers to trie nodes and proximity intervals for the proximity formulas contained in wp_j.

Let us now explain how each word directory $WD(B_j)$ and its forest of tries are organised. The main idea behind this data structure is to store sets of words compactly by exploiting their *common elements*. In this way, memory space is preserved and filtering becomes more efficient as we will see below.

Definition 1. *Let S be a set of sets of words and $s_1, s_2 \in S$ with $s_2 \subseteq s_1$. We say that s_2 is an* identifying subset *of s_1 with respect to S iff $s_2 = s_1$ or $\nexists\, r \in S$ such that $s_2 \subseteq r$.*

The sets of identifying subsets of two sets of words s_1 and s_2 with respect to a set S is the same if and only if s_1 is identical to s_2. Table 1 shows some examples that clarify these concepts.

The sets of words $words(wp_i)$ are organised in the word directory $WD(B_i)$ as follows. Let S be the set of sets of words currently in $WD(B_i)$. When a new set of words s arrives, BestFitTrie selects the best trie in the forest of tries

Id	Query $B_i \sqsupseteq wp_i$	Identifying Subsets
0	$B_i \sqsupseteq$ databases	{databases}
1	$B_i \sqsupseteq$ relational $\prec_{[0,2]}$ databases	{databases, relational}
2	$B_i \sqsupseteq$ databases $\wedge$ relational	{databases, relational}
3	$B_i \sqsupseteq$ (software $\prec_{[0,2]}$ neural $\prec_{[0,0]}$ networks) $\wedge$ (software $\prec_{[0,3]}$ relational $\prec_{[0,0]}$ databases)	{databases, relational, neural}, ...
4	$B_i \sqsupseteq$ optimal $\wedge$ (artificial $\prec_{[0,0]}$ intelligence) $\wedge$ relational $\wedge$ databases	{databases, relational, artificial, intelligence, optimal}, ...
5	$B_i \sqsupseteq$ artificial $\wedge$ relational $\wedge$ intelligence $\wedge$ databases $\wedge$ knowledge	{databases, relational, artificial, intelligence, knowledge }, ...

Table 1: Identifying subsets of $words(wp_i)$ with respect to $S = \{words(wp_i),\ i = 0, \dots, 5\}$.

of $WD(B_i)$, and the best location the that trie to insert s. The algorithm for choosing t depends on the current organization of the word directory and will be given below.

Throughout its existence, each trie T of $WD(B_i)$ has the following properties. The nodes of T store sets of words and other data items related to these sets. Let $sets\text{-}of\text{-}words(T)$ denote the set of all sets of words stored by the nodes of T. A node of T stores more than one set of words if and only if these sets are identical. The root of T (at depth 0) stores sets of words with an identifying subset of cardinality one. In general, a node n of T at depth i stores sets of words with an identifying subset of cardinality $i + 1$. A node n of T at depth i storing sets of words equal to s is implemented as a structure consisting of the following fields:

- $Word(n)$: the $(i + 1)$-th word w_i of identifying subset $\{w_0, \dots, w_{i-1}, w_i\}$ of s where $w_0, \dots, w_{i-1}$ are the words of nodes appearing earlier on the path from the root to node n.
- $Query(n)$: a linked list containing the identifier of query q that contained word pattern wp for which $\{w_0, \dots, w_i\}$ is the identifying subset of $sets\text{-}of\text{-}words(T)$.
- $Remainder(n)$: if node n is a leaf, this field is a linked list containing the words of s that are not included in $\{w_0, \dots, w_i\}$. If n is not a leaf, this field is empty.
- $Children(n)$: a linked list of pairs (w_{i+1}, ptr), where w_{i+1} is a word such that $\{w_0, \dots, w_i, w_{i+1}\}$ is an identifying subset for the sets of words stored at a child of w_i and ptr is a pointer to the node containing the word w_{i+1}.

The sets of words stored at node n of T are equal to $\{w_0, \dots, w_n\} \cup Remainder(n)$, where $w_0, \dots, w_n$ are the words on the path from the root of T to n. An identifying subset of these sets of words is $\{w_0, \dots, w_n\}$. Figure 1 shows the general form of our index structure (we have omitted ET and PA). The part of $WD(B_i)$ corresponding to the queries of Table 1 is shown in full including lists $Query(n)$ and $Remainder(n)$. The purpose of $Remainder(n)$ is to allow for the delayed creation of nodes in trie. This delayed creation lets us choose which word from $Remainder(n)$ will become the child of current node n depending on the sets of words that will arrive later on.

The algorithm for inserting a new set of words s in a word directory is as follows. The first set of words to arrive will create a trie with a randomly chosen word as the root and the rest stored as the remainder. The second set of words will consider being stored at the existing trie or create a trie of its own. In general, to insert a new set of words s, BestFitTrie iterates through the words in s and utilises the hash table implementation of the word directory to find all *candidate tries* for storing s: the tries with root a word of s. To store sets as compactly as possible, BestFitTrie then looks for a trie node n such that the set of words $(\{w_0, \dots, w_n\} \cup Remainder(n)) \cap s$, where $\{w_0, \dots, w_n\}$ is the set of words on the path from the root to n, has maximum cardinality. There may be more than one node that satisfies this requirements and such nodes might belong to different tries. Thus BestFitTrie performs a depth-first search down to depth $|s| - 1$ in *all* candidate tries in order to decide the optimal node n. The path from the root to n is then extended with new nodes containing the words in $\tau = (s \setminus \{w_0, \dots, w_n\}) \cap Remainder(n)$. If $s \subseteq \{w_0, \dots, w_n\} \cup Remainder(n)$, then the last of these nodes l becomes a new leaf in the trie with $Query(l) = Query(n) \cup \{q\}$ (q is the new query from which s was extracted) and $Remainder(l) = Remainder(n) \setminus \tau$. Otherwise, the last of these nodes l points to two child nodes l_1 and l_2. Node l_1 will have $Word(l_1) = u$, where $u \in Remainder(n) \setminus \tau$, $Query(l_1) = Query(n)$ and $Remainder(l_1) = Remainder(n) \setminus (\tau \cup \{u\})$. Similarly node l_2 will have $Word(l_2) = v$, where $v \in s \setminus (\{w_0, \dots, w_n\} \cup \tau)$, $Query(l_2) = q$ and $Remainder(l_2) = s \setminus (\{w_0, \dots, w_n\} \cup \tau \cup \{u\})$. The complexity of inserting a set of words in a word directory is *linear* in the size of the word directory.

The filtering procedure utilises two arrays named $Total$ and $Count$. $Total$ has one element for each query in the database and stores the number of atomic formulas contained in that query. Array $Count$ is used for counting how many of the atomic formulas of a query match the corresponding attributes of a document. Each element of array $Count$ is set to zero at the beginning of the filtering algorithm. If at algorithm termination, a query's entry in array $Total$ equals its entry in $Count$, then the query matches the published document, since all of its atomic formulas match the corresponding document attributes.

When a document d is published at the server, filtering proceeds as follows. BestFitTrie hashes the text value $C(d)$ contained in each document attribute C and probes the ET to find matching atomic formulas with equality. Then for each attribute C in $DAL(d)$ and for each word w in $DWL(C)$, the trie of $WD(C)$ with root w is traversed in a breadth-first manner. Only subtrees having as root a word contained in $C(d)$ are examined, and hash table $OT(d)$ is used to identify them quickly. At each node n of the trie, the list $Query(n)$ gives implicitly all atomic formulas $C \sqsupseteq wp_i$ that can potentially match $C(d)$ if the proximity formulas

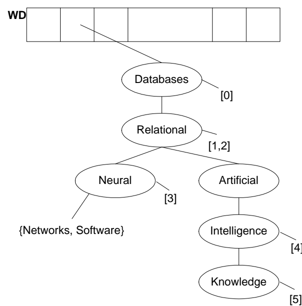

Figure 1: BestFitTrie organisation of the atomic queries of Table 1

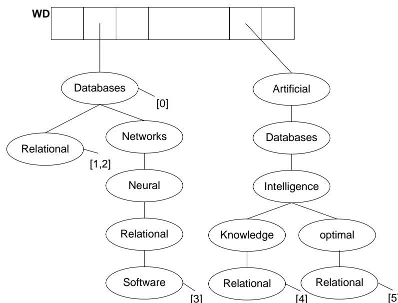

Figure 2: PrefixTrie organisation of the atomic queries of Table 1

in wp_i are also satisfied. This is repeated for all the words in $DWL(C)$, to identify all the qualifying atomic formulas for attribute C. Then the proximity formulas for each qualifying query are examined using the polynomial time algorithm *prox* from [13]. For each atomic formula satisfied by $C(d)$, the corresponding query element in array $Count$ is increased by one. At the end of the filtering algorithm arrays $Total$ and $Count$ are traversed and the values stored for each query are compared. The equal entries in the two arrays give us the queries satisfied by d.

3.2 Other Filtering Algorithms

To evaluate the performance of BestFitTrie we have also implemented algorithms BF, SWIN and PrefixTrie. BF (Brute Force) has no indexing strategy and scans the query database sequentially to determine matching queries. SWIN (Single Word INdex) utilises a two-level index for accessing queries in an efficient way. A query of the form presented at the beginning of this section is indexed by SWIN under all its attributes $A_1, \ldots, A_n, B_1, \ldots, B_m$ and also under n text values $s_1, \ldots, s_n$ and m words selected randomly from $wp_1, \ldots, wp_m$. More specifically SWIN utilises an ET to index equalities and an AD pointing to several WDs to index the atomic containment queries. Atomic queries within a WD slot are stored in a list. PrefixTrie is an extension of the algorithm Tree of [16] appropriately modified to cope with attributes and proximity information. Tree was originally proposed for storing *conjunctions of keywords* in secondary storage in the context of the SDI system SIFT. Following Tree, PrefixTrie uses *sequences* of words sorted in lexicographic order for capturing the words appearing in the word patterns of atomic formulas (instead of sets used by BestFitTrie). A trie is then used to store sequences compactly by exploiting *common prefixes* [16].

Algorithm BestFitTrie constitutes an improvement over PrefixTrie. Because PrefixTrie examines only the prefixes of sequences of words in lexicographic order to identify common parts, it misses many opportunities for clustering (see Figure 2). BestFitTrie keeps the main idea behind PrefixTrie but (a) handles the words contained in a query as a set rather than as a sorted sequence and (b) searches exhaustively the forest of trie to discover the best place to introduce a new set of words. This allows BestFitTrie to achieve better clustering as shown in Figures 1 and 2, where we can see that it needs only one trie to store the set of words for the formulas of Table 1, whereas PrefixTrie introduces redundant nodes that are the result of using a lexicographic order to identify common parts. This node redundancy can be the cause of deceleration of the filtering process as we will show in the next section. To improve beyond BestFitTrie it would be interesting to consider *re-organizing* the word directory every time a new set of words arrives, or periodically, but this might turn out to be prohibitively expensive. In this work we have not explored this approach in any depth.

4. EXPERIMENTAL EVALUATION

To carry out the experimental evaluation of the algorithms described in the previous section, we needed data to be used as incoming documents, as well as user queries. It may not be difficult to collect data to use in the evaluation of filtering algorithms for SDI scenarios. For the model $\mathcal{AWP}$ considered in this paper there are various document sources that one could consider: meta-data for papers on various publisher Web sites (e.g., ACM or IEEE), electronic newspaper articles, articles from news alerts on the Web (e.g., `http://www.cnn.com/EMAIL`) etc. However, it is rather difficult to find user queries except by obtaining proprietary data (e.g., from CNN's news alert system).

For our experiments we chose to use a set of documents downloaded from ResearchIndex[3] and originally compiled in [6]. The documents are research papers in the area of Neural Networks and we will refer to them as the NN corpus[4]. Because no database of queries was available to us, we developed a methodology for creating user queries using *words* and *technical terms* extracted automatically from the Research Index documents using the C-value/NC-value approach of [7]. The extracted multi-word technical terms are used to create proximity formulas and also as conjunctions of keywords in user queries. For the formulation of user queries author names and words extracted from paper abstracts are also used. The attribute universe for the experiments presented in this section consists of paper title, authors, abstract and body.

More specifically the basic concept for the query creation in our methodology is that of a *unit*. Atomic queries are created as conjunctions of units selected uniformly from unit sets, whereas queries are created as conjunctions of atomic queries selected from the attribute universe with a probability p_{C_i}. In our scenario four different types of units sets exist:

[3] `http://www.researchindex.com`

[4] We would like to thank Evangellos Milios and his group at Dalhousie University for providing us the original Neural Network Corpus.

- *Author unit set.* This set contains the last names of authors appearing in the full citation graph of ResearchIndex. Each author appears in the author unit set as many times as the in-degree of the papers he has published. Thus the probability $P(\alpha)$ of author α to appear in a query is $P(\alpha) = N_\alpha / \sum_{k \in V_\alpha} N_k$, where N_α is the number of papers in the citation graph that cite the work of author α, and V_α is the author vocabulary obtained also by the full citation graph.

- *Proximity formulas unit set.* This set contains proximity formulas created using the extracted multi word terms. The technical terms with more than five words were excluded since they were noise, and the set was produced after applying upper and lower NC-value cut-off thresholds for the remaining terms. The proximity operators in this set contain distances according to the number of words contained in each multi-word term.

- *Keywords from technical terms.* This unit set contains keywords extracted from technical terms. These keywords are used as conjuncts in the creation of atomic queries.

- *Nouns from abstracts.* This set contains the nouns used in the corpus abstracts after the cut-off of the most and least frequent words. The rationale behind this is that abstracts are intended to be a comprehensive summary of the publication content, thus nouns from abstracts are appropriate candidates for use in queries.

An example of a user query created artificially from the methodology briefly sketched above is:

$$(\text{AUTHOR} \sqsupseteq \textit{Riedel}) \quad \wedge$$
$$(\text{TITLE} \sqsupseteq \textit{implementation} \wedge (\textit{RBF} \prec_{[0,3]} \textit{networks}))$$

For space reasons the full description of the methodology is omitted but the interested reader can refer to [15].

All the algorithms were implemented in C/C++, and the experiments were run on a PC, with a Pentium III 1.7GHz processor, with 1GB RAM, running Linux. The results of each experiment are averaged over 10 runs to eliminate any fluctuations in the time measurements.

4.1 Varying the Database Size

The first experiment that we conducted to evaluate our algorithms targeted the performance of the four algorithms under different sizes of the query database. In this experiment we randomly selected one hundred documents from the NN corpus and used them as incoming documents in the query databases of different sizes. The size and the matching percentage for each document used was different but the average document size was 6869 words, whereas on average 1% of the queries stored matched the incoming documents.

As we can see in Figure 3, the time taken by each algorithm grows linearly with the size of the query database. However SWIN, PrefixTrie and BestFitTrie are less sensitive than Brute Force to changes in the query database size. The trie-based algorithms outperform SWIN mainly due to the clustering technique that allows the exclusion of more non-matching atomic queries filtering. We can also observe that the better exploitation of the commonalities between

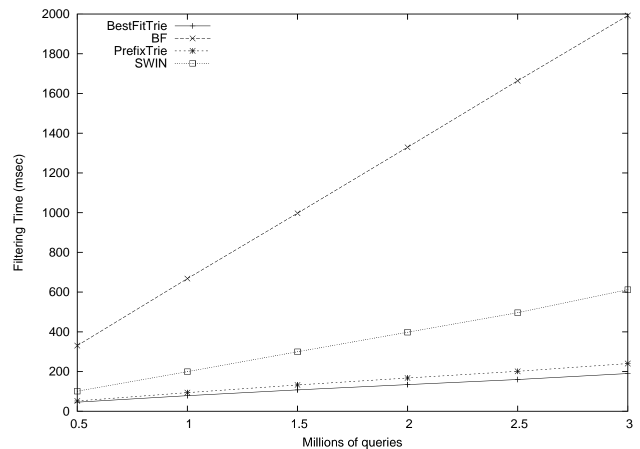

Figure 3: Effect of the query database size in filtering time

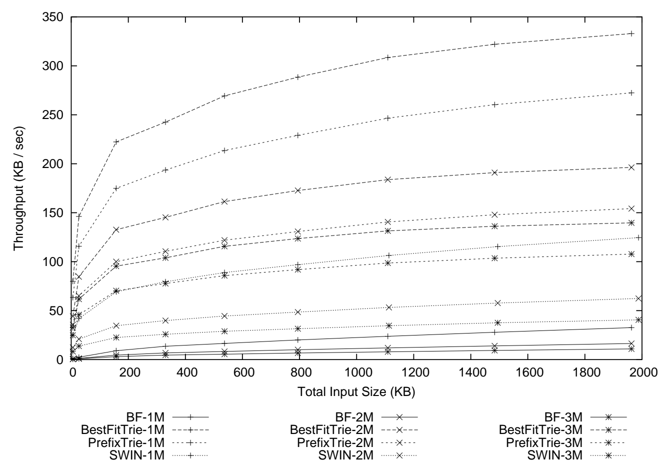

Figure 4: Performance in terms of throughput for the algorithms of Section 3

queries improves the performance of BestFitTrie over PrefixTrie, resulting in a significant speedup in filtering time for *large query databases.* Additionally, Figure 4 contrasts the algorithms in terms of throughput were we can see that BestFitTrie gives the best filtering performance managing to process a load of about 150KB per second for a query database of 3 million queries.

In terms of space requirements BF needs about 15% less space than the trie-based algorithms, due to the simple data structure that poses small space requirements. Additionally the rate of increase for the two trie-based algorithms is similar to that of BF, requiring a fixed amount of extra space each time. From the experiments above it is clear that BestFitTrie speeds up the filtering process with a small extra storage cost, and proves faster than the rest of the algorithms, managing to filter as much as 3 million queries in less the 200 milliseconds, which is about 1000% times faster than the sequential scan method and 20% faster than PrefixTrie.

4.2 Varying the Document Size / Matching Percentage

In this set of experiments we wanted to observe the behaviour of the four algorithms in two different aspects. Initially we varied the matching percentage of the queries for a specific document to observe the matching time variations, and secondly we varied the length of the document to observe the sensitivity of each of the algorithms to such a change.

For the first experiment we used two documents A and B that contained the same number of distinct words and the same attributes, but number of queries that matched each document was different. Notice that the way the algorithms are designed the important parameter of a document is the number of distinct words contained, rather than its size. This happens because the probing of the query index uses the distinct words contained in the attribute text. Practically the increase in the number of distinct words, increases the probability of a specific word contained in a query, to be also contained in the incoming document. This in turn increases the number of queries with proximity formulas that need to be evaluated[5], which is a time consuming process. The size of the document is of smaller importance, since it only increases the number of positions of a specific word in the document, and thus the number of checks at proximity evaluation time. However due to the algorithm presented in [13] the majority of the positions of a specific word in a document can be excluded from the proximity evaluation.

Figure 5 shows the % increase in matching time for two documents A and B with the same number of distinct words, but different number of queries matching them. Document B contained 47155 words, and matched 20% more queries than document A, which contained 42419 words. Both documents contained four $(attribute, value)$ pairs, and the query database contained 3 million queries. Apart from BF which showed a 97% increase in the matching time, BestFitTrie appears to be the most sensitive to the increase in the matching percentage (showing a 19% increase in filtering time), in contrast to PrefixTrie and SWIN, which appear to be less affected (with 13% and 9% increase respectively). This can be explained as follows. The trie structure of PrefixTrie and BestFitTrie forces them to explore a big number of child nodes when a word node appears in a document, in contrast to SWIN that searches in either case all the nodes that are hashed under a specific word. This means that in higher matching percentages, the trie-based algorithms loose some of the advantages offered by their sophisticated data structures and show greater sensitivity to the matching degree. However the trie-based algorithms are still significantly faster, with BestFitTrie being the faster algorithm of all four despite the high increase.

In the second experiment we wanted to observe the behaviour of the four algorithms when the size of the incoming document increases. This time two documents with about the same number of queries matching them were chosen, and the variations in the performance of the four algorithms were examined. Document A was 75344 words long and contained 1864 distinct words, whereas document B was 148609 words long and contained 2304 distinct words, that is about 24% more words than document A. The differences in the performance were below 5% in matching time for SWIN, PrefixTrie and BestFitTrie, whereas BF showed an increase of about 50%. The insensitivity of SWIN, PrefixTrie and BestFitTrie in the document size is mainly due to the hash representation of the document and the way the matching process is carried out. During the matching process we actually consider only the distinct words of the document (that are obviously significantly less than the document itself for large documents), and check the existence of a word in the document using a hash function, which provides fast answer times. Moreover the proximity evaluation is not greatly affected from the large number of word positions inside a document due to the well-designed proximity evaluation algorithm of [13] that allows the omission of a large number of word positions in a document.

[5]Remember that the evaluation of an atomic query is done in two phases; the existence of keywords is checked first and the evaluation of the proximity formulas follows.

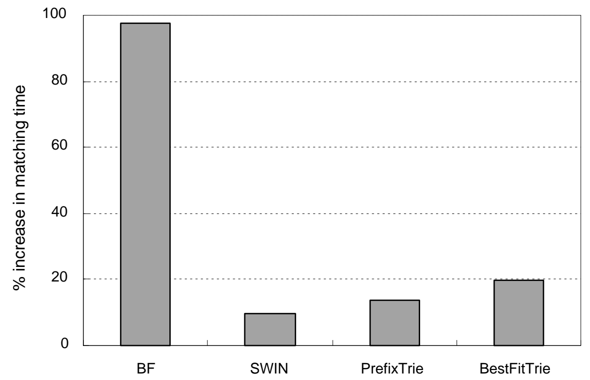

Figure 5: % increase in filtering time for a 20% increase in the number of matching queries

Since in an SDI scenario one may not always have to deal with large documents (for example, if $\mathcal{AWP}$ is used for describing metadata about research papers) we carried out experiments with documents with smaller size. More specifically experiments with documents of mean size of 551 words, show that BestFitTrie performs even better in terms of filtering time, being 1.75 times faster that PrefixTrie and about 86 times faster that BF (as opposed to 1.2 and about 10 times faster respectively for documents of mean size 6869 words).

4.3 Updating the Query Database

In this experiment we investigated the update time for the four different algorithms. To measure the average time needed to insert a single query in the database we worked as follows. Starting with the empty database, we measured the total time needed to populate it with 500K queries, and proceeded in a similar way by adding "bulks" of 500K queries in our database and measuring the total insertion time per bulk. Subsequently the average insertion time per query for a given bulk of queries can be found simply by dividing the total time measured with the population of the bulk to produce a single point in the graph of Figure 6.

It should be clear that for BF and SWIN the query insertion time will be constant on average, since BF does a simple insertion at the end of a list, while SWIN utilises a hash table and inserts each atomic query in the beginning of the list pointed to by the hash table slot. For the

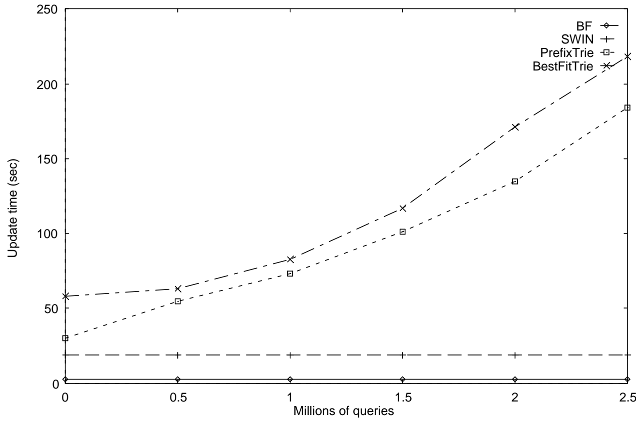

Figure 6: Query insertion time for different query database sizes

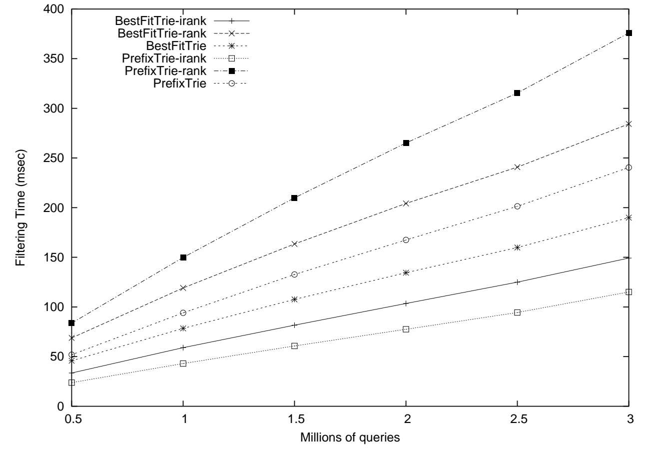

Figure 7: Incorporating word frequency information into the trie-based algorithms, and its effect in filtering time

trie-based algorithms the query insertion is a more complex process that involves the examination of lists at every level of the trie. While PrefixTrie examines only a single path in a single trie of the forest, BestFitTrie needs to examine several paths in the trie and also several tries (in the usual case as many tries as the number of words in a profile). Our is remarks are verified by the graph in Figure 6 that shows the average insertion time in milliseconds for a query q for a given database size. In this figure we can see that BestFitTrie needs about 20% more time than PrefixTrie to insert a query in a database with 2.5 million profiles. This is a standard tradeoff where the algorithm spends some extra time at indexing to save it at query execution.

4.4 Incorporating Ranking Information

To examine the performance of the two trie-based algorithms (namely PrefixTrie and BestFitTrie), we modified them in order to take into account information about the frequency of occurrence of words in documents. More specifically we use the *document frequency* of a word w_i (denoted by df_i), which represents the number of documents in a collection that contain w_i, to identify the frequent and infrequent words among the documents. In an SDI scenario where no document collection is available, we can compute df_i on the collection of recently processed documents [17] (say k most recent documents arrived, where k is large enough). Using these information we created variations of the trie-based algorithms that use different heuristics for storing user queries in tries.

The *rank* heuristic stores the most frequent words among the documents (that is the words with the highest df) near the roots of the tries, while the less frequent words (that is the words with the lowest df) are pushed deeper in them resulting in relatively few big and "wide" tries (since their roots will exist in more queries). The algorithms using the rank heuristic are PrefixTrie-rank and BestFitTrie-rank.

Contrary to *rank*, the *inverse rank* heuristic (*irank*) [17] stores the least frequent words of the queries near the roots of the tries, while the frequent ones are pushed deeper in the tries, resulting in many narrow tries. Thus more queries are put in subtrees of words occurring less frequently, resulting in less lookups during filtering time. The algorithms using the irank heuristic are PrefixTrie-irank and BestFitTrie-irank.

The probability that any word w_i appears in an incoming document d is defined to follow probability distribution $D(w_i)$, where $0 \leq D(w_i) \leq 1$. The number of nodes N that will be examined within each trie depends on the clustering heuristic and is equal to

$$N = D(w_1)N_1 + D(w_2)N_2 + \ldots + D(w_{|V|})N_{|V|} \quad (1)$$

where N_i is the number of nodes in the trie that have word w_i as root node, and $|V|$ is the size of our vocabulary. The sum

$$N_1 + N_2 + \ldots + N_{|V|} \quad (2)$$

is positive and it is always less than or equal to the number of words in the query database.

From Equations 1 and 2 we can see that the number N of nodes examined is minimised if we assign more words to WD slots pointing to words (trie roots) with smaller probability to appear in a document. Based on the above observation we created a modification of BestFitTrie, called LCWTrie (Least Common Word) by limiting BestFitTrie to consider only one candidate trie during insertion: the one that has the least frequent word of the atomic query as root. This way, the atomic query can only be inserted in that trie (or that trie will be created if it does not exist), while the remainder of the words of the atomic query will be organised following the insertion algorithm of BestFitTrie (this will give us the best organisation considering only this trie instead of the whole forest).

In Figure 7 we present the performance of PrefixTrie and BestFitTrie and their ranking variations. We can see that using the rank heuristic the performance of both algorithms deteriorates, due to the creation of large tries that need bigger exploration time. We can also observe that the irank heuristic improves the performance of both trie-based algorithms, with the greater effect shown on PrefixTrie that becomes faster than BestFitTrie-irank. This improvement in performance for both algorithms was expected as shown earlier in this section.

Figure 8 presents the performance of the three faster algorithms, namely PrefixTrie-irank, BestFitTrie-irank and

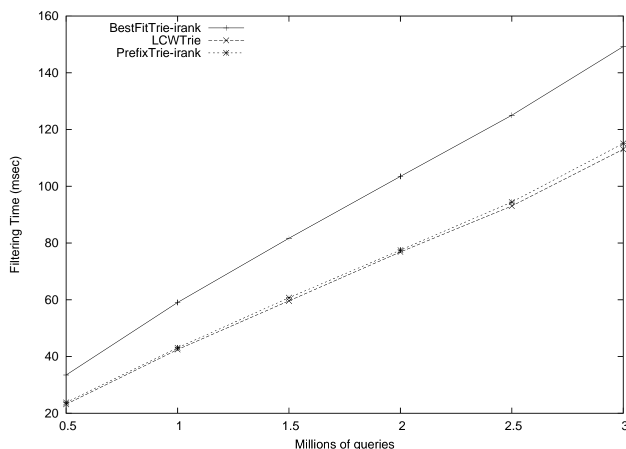

Figure 8: Performance of LCWTrie in comparison to the two faster filtering algorithms

LCWTrie. BestFitTrie-irank prioritises clustering over frequency information by examining all candidate tries and choosing the one that has the most common words. Word frequency information plays a secondary role, allowing the algorithm to choose between tries with the same common words the trie that has the highest ranked word as a root. On the other hand, PrefixTrie-irank and LCWTrie are designed to show a preference in frequency information against clustering. More specifically both algorithms examine exactly one candidate trie, that with the least frequent word as root. Additionally, LCWTrie organises the query within that trie in the best possible way, taking into account common words between the queries already stored. In contrast, PrefixTrie-irank does not care about clustering and stores the query according to frequency information only, that is the word with the lowest rank goes deeper in the trie.

Our observations about the significance of frequency information presented in the beginning of the section are verified. From the experiments of Figure 8 we see that LCWTrie performs similarly with PrefixTrie-irank, although it presents a slight advantage for large query databases, due to the clustering within the trie. Additionally both algorithms outperform BestFitTrie that owes its performance mainly to clustering, giving little consideration to frequency information.

5. OUTLOOK

We are currently working on SDI with data models based on XML and queries based on XQuery/XPath with IR features (phrases, word proximity, similarity etc.).

6. REFERENCES

[1] R.A. Baeza-Yates and G. Navarro. XQL and proximal nodes. *JASIST*, 53(6):504–514, 2002.

[2] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of ICDE*, pages 235–244, February 2002.

[3] C.-C. K. Chang, H. Garcia-Molina, and A. Paepcke. Predicate Rewriting for Translating Boolean Queries in a Heterogeneous Information System. *ACM TOIS*, 17(1):1–39, 1999.

[4] T. T. Chinenyanga and N. Kushmerick. Expressive retrieval from XML documents. In *Proceedings of SIGIR'01*, September 2001.

[5] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM TODS*, 2003.

[6] L. Dong. Automatic term extraction and similarity assessment in a domain specific document corpus. Master's thesis, Department of Computer Science, Dalhousie University, Halifax, Canada, 2002.

[7] K. Frantzi, S. Ananiadou, and H. Mima. Automatic recognition of multi-word terms:the c-value/nc-value method. *JODL*, 5(2), 2000.

[8] N. Fuhr. XML Information Retrieval and Information Extraction. In F. Franke, G. Nakhaeizadeh, and I. Renz, editors, *Text Mining. Theoretical Aspects and Applications*. Physica Verlag, Heidelberg, 2003.

[9] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata. In *Proceedings of ICDT*, pages 173–189, Siena, Italy, January 2003.

[10] S. Idreos, M. Koubarakis, and C. Tryfonopoulos. P2P-DIET: An Extensible P2P Service that Unifies Ad-hoc and Continuous Querying in Super-Peer Networks. In *Proceedings of SIGMOD*, 2004.

[11] M. Koubarakis, T. Koutris, C. Tryfonopoulos, and P. Raftopoulou. Information Alert in Distributed Digital Libraries: The Models, Languages and Architecture of DIAS. In *Proceedings of ECDL*, 2002.

[12] M. Koubarakis, C. Tryfonopoulos, S. Idreos, and Y. Drougas. Selective Information Dissemination in P2P Networks: Problems and Solutions. *ACM SIGMOD Record, Special issue on Peer-to-Peer Data Management, K. Aberer (editor)*, 32(3), 2003.

[13] M. Koubarakis, C. Tryfonopoulos, P. Raftopoulou, and T. Koutris. Data models and languages for agent-based textual information dissemination. In *Proceedings of CIA*, 2002.

[14] XQuery and XPath Full-Text Use Cases. W3C Working Draft 14 February 2003. Available at `http://www.w3.org/TR/xmlquery-full-text-use-cases`.

[15] C. Tryfonopoulos and M. Koubarakis. Selective Information Dissemination: Data Models Based on Information Retrieval and Scalable Filtering Algorithms. Technical Report TR-TUC-ISL2003-01, Technical University of Crete, 2003.

[16] T.W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM TODS*, 19(2):332–364, 1994.

[17] T.W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM TODS*, 1999.