Information Filtering and Query Indexing for an Information Retrieval Model

CHRISTOS TRYFONOPOULOS
Max-Planck Institute for Informatics
MANOLIS KOUBARAKIS
National and Kapodistrian University of Athens
and
YANNIS DROUGAS
University of California Riverside

10

In the information filtering 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 article 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 $\mathcal{AWP}$. $\mathcal{AWP}$ is based on named attributes with values of type text, and its query language includes Boolean and word proximity operators.

Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Information filtering, selective dissemination of information, query indexing algorithms, performance evaluation

This work was supported in part by the European Commission projects DIET (5th Framework Programme IST/FET). C. Tryfonopoulos was partially supported by a Ph.D. fellowship from the program Heraclitus of the Greek Ministry of Education.
This work was performed while the authors were with the Technical University of Crete. This is a revised and extended version of the paper Tryfonopoulos et al. [2004].
Authors' address: C. Tryfonopoulos, Databases and Information Systems Department, Max-Planck Institute for Informatics, Saarbrücken 66123 Germany; email: trifon@mpi-inf.mpg.de; M. Koubarakis, Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Panepistimiopolis, Ilisia, Athens 15784 Greece; email: koubarak@di.uoa.gr; Y. Drougas, Department of Computer Science and Engineering, University of California, Riverside, Riverside, CA 92521; email: drougas@cs.ucr.edu.

ACM Reference Format:
Tryfonopoulos, C., Koubarakis, M., and Drougas, Y. 2009. Information filtering and query indexing for an information retrieval model. ACM Trans. Inform. Syst. 27, 2, Article 10 (February 2009), 47 pages. DOI = 10.1145/ 1462198.1462202 http://doi.acm.org/10.1145/1462198.1462202

1. INTRODUCTION

In the *information filtering 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 article 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 information filtering systems to handle millions of client queries. We concentrate on filtering of *textual information* using a data model which is well-understood in Information Retrieval and has been called $\mathcal{AWP}$ (Attributes with Word Patterns) by us in [Koubarakis et al. 2002, 2003]. 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) [Chang et al. 1999].

As main memory has become cheaper and a memory chip of a few gigabytes is now affordable for a commodity PC, main memory applications have gained higher popularity, as they are able to provide orders-of-magnitude improvement in response times compared to secondary memory approaches. An example of this type of memory-resident applications are MMDBs (Main Memory Database Systems) [Garcia-Molina and Salem 1992; Severance and Pramanik 1990; DeWitt et al. 1984], in which the whole database is kept in main memory, and backups may be stored on disk to avoid data loss. Naturally, not all applications are perfect candidates for MMDBs, and this decision depends on the specifics of the application. In some real-time applications (such as telephony) the database must be entirely memory resident, while in others (such as banking applications) the data may not fit in main memory. Even when the database is too big for main memory, cluster computing and networked workstations are able to provide useful solutions that allow splitting the database into several smaller ones that may fit in a single memory chip, so that parallel processing techniques may then be utilized. These solutions render main-memory algorithms an appealing approach for both mid- and large-scale information providers that emphasize the timeliness of information delivery. Thus, in this article, we develop and evaluate efficient main-memory data structures that are used to index continuous $\mathcal{AWP}$ queries, and algorithms that are able to filter incoming documents against millions of indexed queries in just a few hundred milliseconds.

Our algorithms are the first in the literature that deal with IR models like $\mathcal{AWP}$ supporting Boolean queries, named attributes with values of type text,

and word proximity operators. The main idea behind our algorithms is to use tries to capture common elements of queries. In this way clustering of queries is improved and smaller filtering times are achieved. The algorithms closest to ours are the ones employed in the Boolean version of SIFT [Yan and Garcia-Molina 1994b] 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 4 and 6 extend and improve the results of Yan and Garcia-Molina [1994b]. In particular, we experimentally evaluate algorithms BF, SWIN and PrefixTrie that are extensions of the algorithms BF, Key and Tree of Yan and Garcia-Molina [1994b] 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. Finally, we study the problem of reorganizing the query database in order to achieve greater efficiency at filtering time. We introduce the algorithm ReTrie, an extension of BestFitTrie, which reorganizes the query database based on a clustering quality criterion, and evaluate it against algorithm Periodic, a baseline algorithm that reorganizes the queries periodically.

Preliminary results of this research have appeared in Tryfonopoulos et al. [2004]. The current article revises Tryfonopoulos et al. [2004] and presents the following extensions and additional contributions. We study the problem of reorganizing queries in the database to achieve better clustering and thus shorter filtering times, and present two efficient algorithms for this problem (Section 5). We also include more detailed descriptions of the data model and algorithms of Tryfonopoulos et al. [2004] and their respective data structures, extend the experimental section (Section 6) with new experiments, and give more details for the experiments presented in Tryfonopoulos et al. [2004]. Whereas Tryfonopoulos et al. [2004] used a single corpus of documents to evaluate the proposed algorithms, this article uses two corpora: a small focused corpus of papers about neural networks (called NN corpus), and a bigger and more diverse corpus of documents crawled from the Web (the .GOV corpus). The NN corpus was chosen because it provides a thematically focused setting that matches the requirements and assumptions of our algorithms, while the .GOV corpus was used to assess the limitations of our approach, in a thematically diverse setting. These corpora are representative of digital library scenarios and are also used by other researchers in the area of information retrieval and filtering. Finally, we provide an extensive survey of related work ranging from background on tries, to information filtering methods and other data models and query languages related to $\mathcal{AWP}$.

In work presented in Koubarakis et al. [2002] and Koubarakis et al. [2003], we discuss the distributed alert system DIAS and its ancestor, the peer-to-peer system P2P-DIET [Koubarakis et al. 2003; Idreos et al. 2004a, 2004b]. DIAS and P2P-DIET use $\mathcal{AWP}$ as their *metadata* model for describing and querying digital resources and the filtering algorithm BestFitTrie for matching incoming documents against stored continuous queries. The algorithms presented here are also utilized in systems DHTrie [Tryfonopoulos et al. 2005b] and LibraRing [Tryfonopoulos et al. 2005a], in the context of information filtering and digital library applications built on top of Distributed Hash Tables (DHTs). In all of

these distributed systems, the algorithms of this article are used *locally* in each server, while appropriate protocols (not discussed in this article) guarantee successful distributed operation.

The rest of the article is organized as follows. Section 2 presents related work. Section 3 presents the model $\mathcal{AWP}$, and Section 4 presents the filtering algorithms we developed. Two algorithms for the reorganization of stored queries are presented in Section 5. Section 6 presents our document corpora and the query creation methodology and evaluates the algorithms experimentally under various parameters. Finally, Section 7 concludes the article and hints at our current work.

2. RELATED WORK

In this section we survey related work. We first provide background on tries and their variations. Then, we give an extensive survey of information filtering in various research areas, including IR, databases, distributed systems, and P2P systems. Finally, we compare the expressiveness of the model $\mathcal{AWP}$ with related data models and query languages.

2.1 Tries

The concept of tries was initially conceived by de la Briandais [1959], but the actual term was coined by Fredkin [1960], who derived the name from the term re*trie*val used in information retrieval systems. Tries are widely used in a number of different application domains ranging from dictionary management [Aho et al. 1983; Aoe et al. 1992; Knuth 1973a] and text compression [Bell et al. 1990] to natural language processing [Baeza-Yates and Gonnet 1996; Peterson 1980], pattern matching [Flajolet and Puech 1986; Rivest 1976], IP routing [Nilsson and Karlsson 1999] or searching for reserved words in a compiler [Aho et al. 1986]. The broad applicability of tries has resulted in considering them a general-purpose data structure with properties that are now well understood due to a series of studies [Devroye 1992; Flajolet 1983; Jacquet and Szpankowski 1991; Knuth 1973a; Regnier and Jacquet 1989; Rivest 1976].

There are several ways to implement a trie node depending on the application in mind, but the two most common ways are using arrays in the size of the alphabet (array tries) [Fredkin 1960] and using lists with nonempty elements as roots of subtrees (list tries) [de la Briandais 1959; Knuth 1973b]. Array tries are better suited when the alphabet is relatively small, whereas the list implementation is best suited for large alphabets or trie nodes with few children, as opposed to a fixed size array consisting mainly of null pointers.

There are generally two ways to reduce the size of a trie, reducing the size of each one of the nodes and reducing the number of nodes needed to represent a set of strings. Compact tries [Sussenguth 1963] are variants that are used to reduce the number of nodes needed to represent a certain string, by compacting chains of nodes that lead to a leaf without branching to a single node. Another idea for size reduction in a trie is to view the indexed strings as a set, rather than as a sequence. In this way the size of the resulting trie can be influenced, leading to the smallest trie. However, Comer showed that

the problem of determining the smallest trie is NP-complete [Comer and Sethi 1977]; thus, several works have proposed heuristics to minimize a static trie (e.g., the O-Trie [Comer 1981]).

In our approach we extend the concept of a list trie to implement the data structures used to store user queries in main memory. The algorithm BestFitTrie (and its variations LCWTrie and ReTrie), which we present in this work, utilize variations of list tries and techniques from compact tries to allow for the late creation of trie nodes in order to explore commonalities between user queries.

2.2 Information Filtering

Information retrieval and information filtering are often referred as two sides of the same coin [Belkin and Croft 1992]. Although many of the underlying issues are similar in retrieval and filtering, since they share the common goal of information delivery to information seekers, the design issues (e.g., timeliness of data, identification and representation of user needs) and also the techniques and algorithms devised to satisfy these information needs differ significantly.

Historically, work on selective dissemination of information started by a 1958 article of Luhn [Luhn 1958], where a "Business Intelligence System" is described. In his concept, individual users would have their interests described in profiles, and a text selection system would produce lists of new documents that would allow users to choose between ordering a new document or not. At that day, the selection module was described using the terms *selective dissemination of new information*. The term *information filtering* was coined later by Denning [1982], where he described the need to filter incoming mail messages to sort them in order of urgency. Here we will discuss only the papers that are more relevant to our work and mainly those referring to content filtering (now commonly referred as *content-based filtering*).

Early approaches to information filtering by IR researchers focused more on appropriate representations of user interests [Morita and Shinoda 1994] and on improving filtering effectiveness Hull et al. [1996]. In Morita and Shinoda [1994] behavior monitoring and a substring indexing method is proposed in order to decide which documents are of interest to the user. In [Hull et al. 1996] filtering is addressed using ensemble methods from machine learning, where combinations of strategies are explored as a means to increase filtering effectiveness. Other approaches include statistical filtering systems such as LSI-SDI [Foltz and Dumais 1992], that uses the LSI method to filter incoming documents.

One of the first papers in this area to address performance is Bell and Moffat [1996], where an information filtering system capable of scaling up to large filtering tasks is described. The authors assume a server that receives documents at a high rate, and propose algorithms that support vector space queries by improving the algorithm SQI of Yan and Garcia-Molina [1994a]. InRoute [Callan 1996] was another influential system based on inference networks with emphasis on filtering efficiency. InRoute creates documents and query networks and uses belief propagation techniques to filter incoming documents. Other works in the area mainly focus on adaptive filtering [Callan 1998; Zhang and Callan

2001] and how vector space queries and their dissemination thresholds are adapted based on documents processed in the past.

Apart from the statistical filtering approaches we have described, filtering systems based on the Boolean model have also been developed. A representative example is LMDS [Yochum 1985], that uses least frequent trigrams to allow for fast processing of incoming documents. 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.

Most of the work on information filtering in the database literature has its origins in Franklin and Zdonik [1998] which also used the term *selective dissemination of information (SDI)*. Their preliminary work on the system DBIS appears in Altinel et al. [1999]. The term *publish/subscribe* (pub/sub) system, which comes from distributed systems, has also been used in this context by database researchers. Another influential system is SIFT [Yan and Garcia-Molina 1994b, 1999b], where publications are documents in free text form and queries are conjunctions of keywords. SIFT was the first system to emphasize query indexing as a means to achieve scalability in pub/sub systems [Yan and Garcia-Molina 1994b]. Later on, similar work concentrated on pub/sub systems with data models based on attribute-value pairs and query languages based on attributes with arithmetic and string comparison operators (e.g., Le Subscribe [Fabret et al. 2001], the monitoring subsystem of Xyleme [Nguyen et al. 2001] and others). [Campailla et al. 2001] is also notable because it considers a data model based on attribute-value pairs but goes beyond conjunctive queries—the standard class of queries considered by other systems [Fabret et al. 2001]. More recent work has concentrated on publications that are XML documents and queries that are subsets of XPath or XQuery (e.g., XFilter [Altinel and Franklin 2000], YFilter [Diao et al. 2003], Xtrie [Chan et al. 2002] and *xmltk* [Green et al. 2003]). All these papers discuss sophisticated filtering algorithms based on indexing queries.

In the area of distributed systems and networks various pub/sub systems have been developed over the years over. Researchers have utilized here various data models based on channels, topics and attribute-value pairs (exactly like the models of the database papers discussed above) [Carzaniga et al. 2001]. The latter systems are called content-based like in the IR literature, as attribute-value data models are flexible enough to express the content of messages in various applications. The query languages of these systems are based on Boolean combinations of arithmetic and string operations. Work in this area has concentrated not only on filtering algorithms as in the database papers previously surveyed, but also on distributed pub/sub architectures [Aguilera et al. 1999; Carzaniga et al. 2001] based on *unstructured networks*. SIENA [Carzaniga et al. 2001] is probably the most elegant example of a system to be developed in this area. SIENA uses a data model and language based on attribute-value pairs and demonstrates how to express notifications, subscriptions and advertisements in this language. The core ideas of SIENA have recently been used in the pub/sub systems DIAS [Koubarakis et al. 2002] and P2P-DIET [Koubarakis

et al. 2003; Idreos et al. 2004b], but now the data models $\mathcal{AWP}$ and $\mathcal{AWPS}$ utilized were inspired from Information Retrieval. DIAS and P2P-DIET have also emphasized the use of sophisticated subscription indexing at each server to facilitate efficient forwarding of notifications [Tryfonopoulos et al. 2004]. In some sense, the approach of DIAS and P2P-DIET puts together prominent ideas from the database and distributed systems tradition in a single unifying framework. Another important contribution of P2P-DIET is that it demonstrates how to support by very similar protocols the traditional *ad-hoc* or *one-time* query scenarios of typical super-peer systems [Yang and Garcia-Molina 2003] and the pub/sub features of SIENA [Carzaniga et al. 2001]. Lately, the iClusterDL system [Raftopoulou et al. 2008] has showed how to exploit a special form of unstructured networks, called Semantic Overlay Networks [Crespo and Garcia-Molina 2002], to support both IR and IF functionality in a Digital Libary domain.

With the advent of DHTs such as CAN [Ratnasamy et al. 2001], Chord [Stoica et al. 2001] and Pastry [Rowstron and Druschel 2001], a new wave of pub/sub systems based on *structured networks* has appeared. Scribe [Rowstron et al. 2001] is a topic-based publish/subscribe system based on Pastry [Rowstron and Druschel 2001]. Hermes [Pietzuch and Bacon 2002] is similar to Scribe because it uses the same underlying DHT (Pastry) but it allows more expressive subscriptions by supporting the notion of an event type with attributes. Each event type in Hermes is managed by an event broker, which is a rendezvous node for subscriptions and publications related to this event. Related ideas appear in Tam et al. [2003] and Terpstra et al. [2003]. PeerCQ [Gedik and Liu 2003] is another notable pub/sub system implemented on top of a DHT infrastructure. The most important contribution of PeerCQ is that it takes into account peer heterogeneity and extends consistent hashing [Karger et al. 1997] with simple load balancing techniques based on appropriate assignment of peer identifiers to network nodes.

Meghdoot [Gupta et al. 2004] is a very recent pub/sub system implemented on top of a CAN-like DHT [Ratnasamy et al. 2001]. Meghdoot supports an attribute-value data model and offers new ideas for the processing of subscriptions with range predicates (e.g., the price is between 20 and 40 Euros) and load balancing. A P2P system with a similar attribute-value data model that has been utilized in the implementation of a publish-subscribe system for network games is Mercury [Bharambe et al. 2002, 2004]. A recent paper [Aekaterinidis and Triantafillou 2005] implements a DHT-agnostic solution to support prefix and suffix operations over string attributes in a pub/sub environment.

Recently, several systems that employed an IR-based query language to support information filtering on top of structured overlay networks have been deployed. pFilter [Tang and Xu 2003] uses a hierarchical extension of the CAN DHT to store user queries and relies on multicast trees to notify subscribers, while DHTrie [Tryfonopoulos et al. 2005b] extends the Chord protocol to achieve exact information filtering functionality and applied document-granularity dissemination to achieve the recall of a centralized system. In the same spirit, LibraRing [Tryfonopoulos et al. 2005a] presents a framework to provide information retrieval and filtering services in two-tier digital library

environments. Contrary to DHTrie and LibraRing, the MinervaDL [Zimmer et al. 2007] and MAPS [Tryfonopoulos et al. 2007; Zimmer et al. 2008] systems suggest using the Chord DHT to disseminate and store statistics about the document providers rather than the documents themselves. In this way, only a few carefully selected, specialized, and promising peers store the user queries and are monitored for new publications. This *approximate information filtering* relaxes the assumption—which holds in most pub/sub systems—of potentially delivering notifications from every producer and amplifies scalability.

2.3 Data Models and Query Languages Related to $\mathcal{AWP}$

In the database literature, word patterns have been studied by Chang and colleagues in the context of integrating heterogeneous digital libraries [Chang et al. 1996; Chang et al. 1999; Chang 2001]. The model $\mathcal{AWP}$ is essentially the model of Chang [2001] but with a slightly different class of word patterns.

Limited forms of proximity operators as we use them in $\mathcal{AWP}$ have been offered in the past by various search engines (e.g., Altavista had an operator *NEAR* that meant word-distance 10, Lycos had an operator *NEAR* that meant word-distance 25, and Infoseek used to have a more sophisticated facility). Proximity operators have also been implemented in other systems such as freeWAIS [Pfeifer et al. 1995] and InQuery [Callan et al. 1992]. There are also advanced IR models such as the model of proximal nodes [Navarro and Baeza-Yates 1997] with proximity operators between arbitrary structural components of a document (e.g., paragraphs or sections). Data models and query languages for full-text extensions to XML, for example, TeXQuery [Amer-Yahia et al. 2004] is the most recent area of research where proximity operators have been used.

The data model $\mathcal{AWP}$ is also related to recent proposals for representing and querying textual information in publish/subscribe systems [Carzaniga et al. 2000; Campailla et al. 2001]. However, $\mathcal{AWP}$ uses linguistically motivated concepts such as *word* and traditional IR operators instead of strings and string operators used in the publish/subscribe models [Carzaniga et al. 2000; Campailla et al. 2001].

In Koubarakis et al. [2002] we have extended the model $\mathcal{AWP}$ by introducing a "similarity" operator based on the IR vector space model [Baeza-Yates and Ribeiro-Neto 1999]. The similarity concept of this model, called $\mathcal{AWPS}$ (where $\mathcal{S}$ stands for similarity), has in the past been used in database systems with IR influences (e.g., WHIRL [Cohen 2000]) and more recently in XML-based query languages, for example, ELIXIR [Chinenyanga and Kushmerick 2001], XIRQL [Fuhr and Großjohann 2004], XXL [Theobald and Weikum 2000] and TopX [Theobald et al. 2005].

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

In this section we present the data model $\mathcal{AWP}$ for specifying documents and queries [Koubarakis et al. 2002]. Documents in $\mathcal{AWP}$ are defined using *named attributes* with values of type *text*. The query language of $\mathcal{AWP}$ offers *Boolean* and *proximity operators* on attribute values as in Chang et al. [1999].

3.1 Defining Documents

Let Σ be a finite *alphabet*. A *word* is a finite nonempty sequence of letters from Σ. Let $\mathcal{V}$ be a finite set of words called the *vocabulary*. In the examples of this article, $\mathcal{V}$ will be the vocabulary of the English language.

Definition 3.1. A *text value* s of length n is a total function $s : \{1, 2, \ldots, n\} \to \mathcal{V}$.

In other words, a text value s is a finite sequence of words from the assumed vocabulary and $s(i)$ gives the i-th element of s. Text values can be used to represent finite-length strings consisting of words separated by blanks. The *length* of a text value s (i.e., its number of words) will be denoted by $|s|$.

Example 3.2. The string "this paper is about data structures" can be represented by a text value s of length 6 with $s(1) = this$, $s(2) = paper$ etc.

In the rest of this section, we use the intuitive string notation to give examples of text values while our definitions use the mathematical definition given above.

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[1].

Definition 3.3. A document d is a set of attribute-value pairs (A, s) where $A \in \mathcal{A}$, s is a text value, and all attributes are distinct.

In the rest of this article, we will often use the intuitive notation $A(d)$ to refer to the unique text value s such that $(A, s) \in d$.

Example 3.4. The following set of pairs is a document:

$$d = \{\ (AUTHOR, \text{"}John\ Smith\text{"}),$$
$$(TITLE, \text{"}Selective\ dissemination\ of\ information\ in\ peer\text{-}to\text{-}peer\ systems\text{"}),$$
$$(ABSTRACT, \text{"}In\ this\ paper\ we\ show\ that\ \ldots\text{"})\ \}$$

Obviously, $AUTHOR(d) =$ "*John Smith*".

3.2 Defining Queries

To define queries in $\mathcal{AWP}$, we start with the concept of word defined earlier and use it, together with the concept of interval to be defined immediately, to define proximity formulas and word patterns. Finally, using the concept of attributes as well, we define queries.

Let $\mathcal{I}$ be a set of *(distance) intervals*

$$\mathcal{I} = \{[l, u] :\ l, u \in \mathbb{N}, l \geq 0 \text{ and } l \leq u\} \cup \{[l, \infty) :\ l \in \mathbb{N} \text{ and } l \geq 0\}.$$

Definition 3.5. A proximity formula is an expression of the form

$$w_1 \prec_{i_1} \cdots \prec_{i_n} w_n$$

where $w_1, \ldots, w_n$ are words of $\mathcal{V}$ and $i_1, \ldots, i_n$ are intervals of $\mathcal{I}$.

[1] `http://purl.org/dc/elements/1.1/`

The operators $\prec_i$ used in a proximity formula are called *proximity operators*. Proximity operators are used to capture the concepts of *order* and *distance* between words in a text document, using intervals that impose lower and upper bounds on distances between words. 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 formulas with more than one operator $\prec_i$ is similar.

Example 3.6. The following are proximity formulas:

$$\textit{information} \prec_{[0,0]} \textit{retrieval}, \quad \textit{Web} \prec_{[0,0]} \textit{information} \prec_{[0,0]} \textit{retrieval},$$
$$\textit{topics} \prec_{[0,5]} \textit{information} \prec_{[0,0]} \textit{retrieval}$$

In this example, the proximity formula $\textit{information} \prec_{[0,0]} \textit{retrieval}$ denotes that the word "information" appears exactly before word "retrieval" so this is a way to encode the string "information retrieval" in $\mathcal{AWP}$. We can also have arbitrarily longer sequences of proximity operators with similar meaning. In the proximity formula

$$\textit{topics} \prec_{[0,5]} \textit{information} \prec_{[0,0]} \textit{retrieval}$$

the word "topics" is constrained to precede the word "information" by at least 0 words and at most 5 words. Similarly, the word "information" should appear exactly before the word "retrieval".

Definition 3.7. A *word pattern* is a conjunction of words of $\mathcal{V}$ and proximity formulas.

Example 3.8. The following expression is a word pattern:

$$\textit{applications} \wedge (\textit{selective} \prec_{[0,0]} \textit{dissemination} \prec_{[0,3]} \textit{information}).$$

In the following definition, the symbol $\sqsupseteq$ should be read as "contains." The notation $B \sqsupseteq wp$ is a formal way of saying that the value of attribute B contains the pattern of words specified by word pattern wp.

Definition 3.9. 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.

Example 3.10. The following formula is a query:

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

3.3 Semantics of Query Answering

Let us now define the semantics of query answering in $\mathcal{AWP}$. In the information filtering scenario studied in this article, query answering is equivalent to knowing when a document *satisfies* or *matches* a user query. This notion is now defined formally in two steps. First, Definition 3.11 defines when a text value satisfies a word pattern. Then, this concept is used by Definition 3.13 to capture when a document satisfies a query.

Definition 3.11. Let s be a text value and wp a word pattern. The concept of s satisfying wp (denoted by $s \models wp$) is defined as follows:

(1) If wp is a word then $s \models wp$ iff there exists $p \in \{1, \ldots, |s|\}$ and $s(p) = wp$.
(2) If wp is a proximity formula of the form $w_1 \prec_{\boldsymbol{i}_1} \cdots \prec_{\boldsymbol{i}_{n-1}} w_n$ then $s \models wp$ iff there exist $p_1, \ldots, p_n \in \{1, \ldots, |s|\}$ such that, for all $j = 2, \ldots, n$ we have $s(p_j) = w_j$ and $p_j - p_{j-1} - 1 \in \boldsymbol{i}_{j-1}$.
(3) If wp is of the form $wp_1 \wedge wp_2$ then $s \models wp$ iff $s \models wp_1$ and $s \models wp_2$.

Example 3.12. The text value "applications of selective dissemination of information" satisfies the word pattern of Example 3.8.

Definition 3.13. Let d be a document and ϕ be a query. The concept of document d *satisfying* query ϕ (denoted by $d \models \phi$) is defined as follows:

(1) If ϕ is of the form $A \sqsupseteq wp$ then $d \models \phi$ iff there exists a pair $(A, s) \in d$ and $s \models wp$.
(2) If ϕ is of the form $A = s$ then $d \models \phi$ iff there exists a pair $(A, s) \in d$.
(3) If ϕ is of the form $\phi_1 \wedge \phi_2$ then $d \models \phi$ iff $d \models \phi_1$ and $d \models \phi_2$.

Example 3.14. The value of attribute *AUTHOR* of document d in Example 3.4 is "John Smith" and the value of attribute *TITLE* satisfies word pattern

$$(\textit{selective} \prec_{[0,0]} \textit{dissemination} \prec_{[0,3]} \textit{information}) \wedge \textit{peer-to-peer}$$

according to Definition 3.11. Thus, d satisfies the query of Example 3.10 according to Definition 3.13.

3.4 The Expressive Power of Proximity Operators in $\mathcal{AWP}$

We have already surveyed models related to $\mathcal{AWP}$ in Section 2.3. Here we discuss in more detail the expressive power of proximity operators which are a powerful feature of $\mathcal{AWP}$. In this respect, we compare $\mathcal{AWP}$ with the model of Chang et al. [1999] where the most detailed formal analysis of proximity operators that exists in the literature has been given, and XQuery Full-Text,[2] the most recent query language with sophisticated proximity operators.

The proximity operators we use in this article are more expressive than the traditional IR proximity operator kW with meaning "the first operand must precede the second by no more than k words" used in Chang et al. [1999] and other papers. More precisely, expression $w_1\, kW\, w_2$ in the model of Chang et al. [1999] is equivalent to expression $w_1 \prec_{[0,k]} w_2$ in our model. In addition, the operators kW of Chang et al. [1999] are not powerful enough to express proximity formulas such as $w_1 \prec_{[l,u]} w_2$ of our model when $l > 0$ or $u = \infty$.

The operator kN of [Chang et al. 1999] with meaning "the operands must have a distance of k words but the order does not matter" cannot be expressed in our model. However, notice that if we introduce disjunction $w_1\, kN\, w_2$ can be *approximated* by $w_1 \prec_{[0,k]} w_2 \vee w_2 \prec_{[0,k]} w_1$. Chang et al. [1999] give an example (page 23) that demonstrates why these two expressions are not equivalent

[2] `http://www.w3.org/TR/xpath-full-text-10/`

given the meaning of operator kN. The example involves a text value and word patterns with overlapping positions in that text value hence the difference.

Similar comments hold with respect to the predicates

`with distance` *range* `words` or `within window` *range* `words`

of XQuery Full Text, where *range* is `exactly` x or `at most` x or `at least` x or `from` x `to` x and $x, y \geq 0$. The proximity operators of $\mathcal{AWP}$ have exactly the same expressive power with the `with distance` predicate but are less expressive than the `within window` predicate.

Because operator kN of Chang et al. [1999] and predicate `within window` of XQuery Full Text can be useful in implemented information filtering systems, Section 4.1.5 explains that it is straightforward to extend the indexing and filtering algorithms of this article to deal with them.

We have now completed the formal presentation of model $\mathcal{AWP}$ used in this article. The reader is invited to read Koubarakis et al. [2006], where more results on the "logic" of $\mathcal{AWP}$ are presented.

4. FILTERING ALGORITHMS

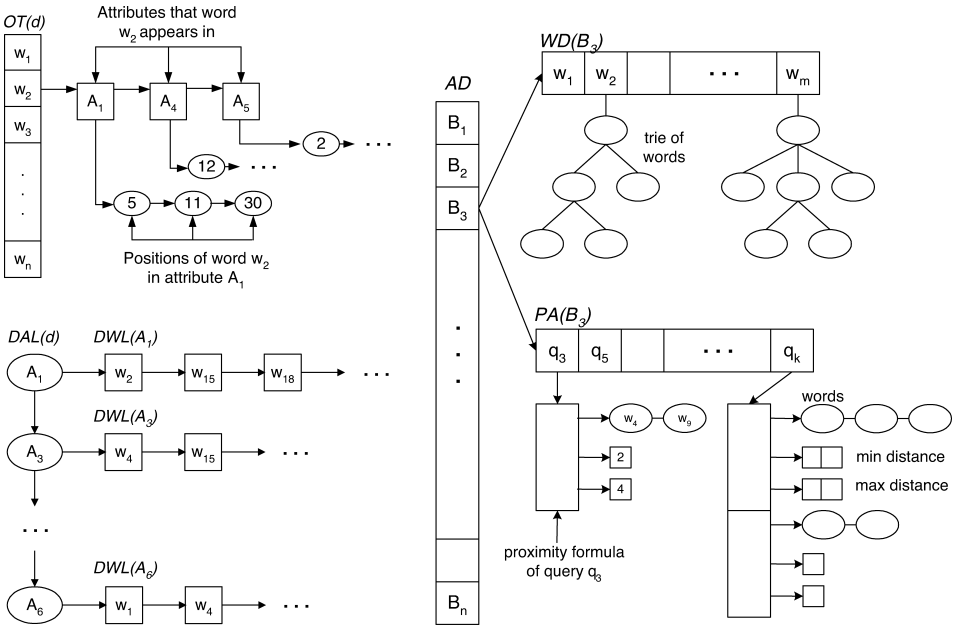
In this section, we present four main memory algorithms that solve the filtering problem for *conjunctive queries* in $\mathcal{AWP}$. Because our work extends and improves previous algorithms from SIFT [Yan and Garcia-Molina 1994b], we adopt terminology from SIFT in many cases.

4.1 The Algorithm BestFitTrie

In a filtering setting, contrary to the retrieval scenario, queries are indexed and matched against incoming documents. To speed up the matching procedure, documents and queries need to be represented in an appropriate form, by utilizing appropriate data structures. In the following, we describe the data structures used for representing incoming documents and indexing queries, and present the query insertion and filtering mechanisms.

4.1.1 *Data Structures for Documents.* When a document is published in the system, it has to be represented by appropriate data structures that will facilitate the matching process. BestFitTrie uses two data structures to represent each published document d: the *occurrence table* $OT(d)$ and the *distinct attribute list* $DAL(d)$ (shown graphically in Figure 1(a)). $OT(d)$ 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 each 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 $A(d)$.

4.1.2 *Data Structures for Queries.* To index queries BestFitTrie utilizes an array, called the *attribute directory* (AD), that stores pointers to word directories and proximity arrays. AD has one element for each distinct attribute B in the query database. Each element of AD is a *word directory* for attribute B that



(a) Occurrence table $OT(d)$ and distinct attribute list $DAL(d)$

(b) Query index with word directory $WD(B_i)$ and proximity array $PA(B_i)$

Fig. 1. Document and query data structures.

indexes the set of words in wp (denoted by $words(wp)$) for each atomic formula $B \sqsupseteq wp$ in a query. Technically, a word directory $WD(B)$ for attribute B is a hash table that uses words as keys and provides fast access to roots of *tries* in a *forest* that is used to organize the *sets of words* that result from $words(wp)$. The proximity formulas contained in wp are stored in an array called the *proximity array* (denoted by $PA(B)$). $PA(B)$ stores pointers to trie nodes (words) that are operands in proximity formulas along with the respective proximity intervals for each formula. Thus for a proximity formula of the form $w_1 \prec_{[l_1,u_1]} \cdots \prec_{[l_n,u_n]} w_n$, array $PA(B)$ will contain pointers to the words $w_1, \cdots, w_n$ stored in $WD(B)$, and also all the integers $l_1, u_1, \cdots, l_n, u_n$ representing the distance intervals in the proximity formula. Finally a hash table, called *equality table* (ET), that indexes, under key s, all text values s that appear in formulas of the form $A = s$ is also used. Figure 1(b) illustrates the AD, the $WD(B)$ and the $PA(B)$ used to index submitted user queries.

4.1.3 *Query Insertion.* When a new query q of the form

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

arrives, the index structures are populated as follows. For each attribute A_i, $1 \leq i \leq n$, we hash the text value s_i to obtain a slot in ET that will index this text value. 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

Table I. An Example with Identifying Subsets

k	Query $B \sqsupseteq wp_j$	Identifying Subsets
1	$B \sqsupseteq$ databases	{databases}
2	$B \sqsupseteq$ relational $\prec_{[0,2]}$ databases	{databases, relational}
3	$B \sqsupseteq$ databases $\wedge$ relational	{databases, relational}
4	$B \sqsupseteq$ (software $\prec_{[0,2]}$ neural $\prec_{[0,0]}$ networks) $\wedge$ (software $\prec_{[0,3]}$ relational $\prec_{[0,0]}$ databases)	{databases, relational, neural}, {databases, relational, software}, {databases, relational, networks}, ...
5	$B \sqsupseteq$ optimal $\wedge$ (artificial $\prec_{[0,0]}$ intelligence) $\wedge$ relational $\wedge$ databases	{optimal}, {optimal, databases}, {databases, relational, artificial, intelligence, optimal}, ...
6	$B \sqsupseteq$ artificial $\wedge$ relational $\wedge$ intelligence $\wedge$ databases $\wedge$ knowledge	{knowledge}, {knowledge, databases}, {databases, relational, artificial, knowledge}, ...

$PA(B_j)$ 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 organized. 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 optimized and filtering can be efficiently carried out as we will see below. To understand how $WD(B_j)$ is constructed, we need the following definition.

Definition 4.1. Let S be a set of nonempty 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 there is no $r \in S$, such that $r \neq s_1$ and $s_2 \subseteq r$.

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.

Example 4.2. Table I shows some examples that clarify these concepts. In each row of the table, we give query $B \sqsupseteq wp_j$ and some identifying subsets of words(wp_j) with respect to $S = \{words(wp_k) : \ 1 \leq k \leq j-1\}$.

Notice that in some entries of the table we do not give the complete set of identifying subsets due space considerations.

For each formula $B \sqsupseteq wp_j$ in a user query, the sets of words $words(wp_j)$ are incrementally organized in the word directory $WD(B)$ as follows. Let S be the set of sets of words currently in $WD(B)$. When a new set of words s arrives, BestFitTrie selects the best trie T in the forest of tries of $WD(B)$, and the best location in that trie to insert s. The algorithm for choosing T takes into account the current organization of the word directory and will be presented below.

Throughout its existence, each trie T of $WD(B)$ has the following properties. The nodes of T store sets of words and other data items related to these sets. Let *sets-of-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 $\{w_0, \ldots, w_{i-1}, w_i\}$ of s where $w_0, \ldots, w_{i-1}$ 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 $\{w_0, \ldots, w_i\}$ is the identifying subset of *sets-of-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 $\{w_0, \ldots, w_i\}$. 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 $\{w_0, \ldots, w_i, w_{i+1}\}$ is an identifying subset for the sets of words stored at a child of w_i and ptr is a pointer to the node n', where $Word(n') = w_{i+1}$.

The sets of words stored at node n of T are equal to $\{w_0, \ldots, w_n\} \cup Remainder(n)$, where $w_0, \ldots, w_n$ are the words on the path from the root of T to n. An identifying subset of these sets of words is $\{w_0, \ldots, w_n\}$. Figure 4(a) shows an example of a trie created by BestFitTrie for the queries of Table I, together with the *Query*(n) and *Remainder*(n) lists. The purpose of *Remainder*(n) is to allow for the delayed creation of nodes in a 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.

To continue with the algorithm for inserting a new set of words s in a word directory, we will first need to define the concept of clustering ratio.

Definition 4.3. Let s be a set of words indexed at node n of trie T. For this set of words, we have that $s = \{w_0, \ldots, w_n\} \cup Remainder(n)$, where $w_0, \ldots, w_n$ are the words in the path from the root of T to node n. The clustering ratio of s in T, denoted as *ClusteRat*(s, T), is $ClusteRat(s, T) = \frac{|\{w_0, \ldots, w_n\}|}{|s|}$, where $|\{w_0, \ldots, w_n\}|$ and $|s|$ denote the number of words in the corresponding sets of words.

ClusteRat(s, T) is used to quantify how well the set of words s is clustered in trie T. From the definition, it follows that $0 < ClusteRat(s, T) \leq 1$. Generally when *ClusteRat*(s, T) is near 0, the set of words s is considered badly clustered, whereas when *ClusteRat*(s, T) is near 1, the set of words is considered highly clustered.

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 utilizes the hash table implementation of the word directory to find all *candidate tries* T' for storing s: the tries with a word of s as root. To store sets as compactly as possible, BestFitTrie then looks for a trie node n in trie T' such that, if s was indexed there, *ClusteRat*(s, T') would be maximized.

To identify the optimal node n, BestFitTrie performs a depth-first search down to depth $|s|-1$ in *all* candidate tries. If more than one nodes that maximize *ClusteRat*(s, T) are found, BestFitTrie randomly chooses one. The path from

Algorithm *BestFitTrie*

```
01: begin
02:   for each attribute C ∈ q do                          ▷ for all query attributes
03:     curClusterRat ← 0
04:     curPosition ← Null
05:     if formula is of the form C = s then               ▷ handle equalities
06:      store s at ET
07:     else                                               ▷ handle containment operators
08:       for each trie T such that root(T) ∈ s do         ▷ for all candidate tries
09:         for each node n ∈ T such that word(n) ∈ s do   ▷ for all possible storage
                                                             positions in candidate tries
                                                             perform a DFS
10:           calculate clusterRat(s)
11:           if clusterRat(s) > curClusterRat then        ▷ if a better position is
                                                             found make a note of it
12:             curClusterRat ← clusterRat(s)
13:             curPosition ← n
14:           end if
15:         end for
16:       end for
17:       if curPosition = Null then                       ▷ if s cannot be indexed in
                                                             any existing trie
18:         create trie T with root(T) ← w_i ∈ s           ▷ create new trie with a random
                                                             word from s as a root
19:         curPosition ← root(T)
20:         Remainder(curPosition) ← s \ {w_i}             ▷ put the rest of the words in s
                                                             in the Remainder(curPosition)
21:         updatePA(C)                                    ▷ store proximity bounds
                                                             for indexed queries
22:       else
23:         store s at node curPosition
24:         Remainder(curPosition) ← s \ {w_0, ..., w_n}   ▷ put the rest of the words in s
                                                             in the Remainder(curPosition)
25:       end if
26:     end if
27:   end for
28: end
```

Fig. 2. Pseudocode for query insertion under algorithm BestFitTrie.

the root to n is then extended with new nodes containing the words in $\tau = (s \setminus \{w_0, \ldots, w_n\}) \cap \mathit{Remainder}(n)$. If $s \subseteq \{w_0, \ldots, w_n\} \cup \mathit{Remainder}(n)$, then the last of these nodes l becomes a new leaf in the trie with $\mathit{Query}(l) = \mathit{Query}(n) \cup \{q\}$ (q is the new query from which s was extracted) and $\mathit{Remainder}(l) = \mathit{Remainder}(n) \setminus \tau$. Otherwise, the last of these nodes l points to two child nodes l_1 and l_2. Node l_1 will have $\mathit{Word}(l_1) = u$, where $u \in \mathit{Remainder}(n) \setminus \tau$, $\mathit{Query}(l_1) = \mathit{Query}(n)$ and $\mathit{Remainder}(l_1) = \mathit{Remainder}(n) \setminus (\tau \cup \{u\})$. Similarly node l_2 will have $\mathit{Word}(l_2) = v$, where $v \in s \setminus (\{w_0, \ldots, w_n\} \cup \tau)$, $\mathit{Query}(l_2) = q$ and $\mathit{Remainder}(l_2) = s \setminus (\{w_0, \ldots, w_n\} \cup \tau \cup \{u\})$.

Figure 2 presents the pseudocode for query indexing under algorithm BestFitTrie. The complexity of inserting a set of words in a word directory is *linear* in the size of the word directory.

4.1.4 *Filtering Incoming Documents.* The filtering procedure utilizes 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 and increases the corresponding query element in array *Count* by one. 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 depth-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 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 a straight forward polynomial-time algorithm that takes a proximity formula and a text value as an input, and decides whether the proximity formula is satisfied by the text value (see Koubarakis et al. [2002] for details). 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. Figure 3 presents the pseudocode for filtering incoming documents in BestFitTrie.

4.1.5 *Dealing with Other Proximity Operators.* It is straightforward to extend the indexing and filtering algorithms of this article to deal with the kN operator of Chang et al. [1999] and predicate `within window` of XQuery Full Text. Similarly to the current approach, the words that are operands to the proximity formulas will be indexed in the appropriate trie of the respective $WD(B)$, and the values of the proximity intervals will be stored at the $PA(B)$. At filtering time, the containment of the words of the proximity formula in the incoming document will be checked using $WD(B)$, and the proximity constraints for each qualifying formula will be examined using an appropriate algorithm similar to the one in Koubarakis et al. [2002].

4.2 Other Filtering Algorithms

To evaluate the performance of BestFitTrie, we have also implemented algorithms BF, SWIN and PrefixTrie, which are modified versions of the algorithms BF, Key and Tree developed for SIFT in Yan and Garcia-Molina [1994b]. These modifications and extensions to the SIFT algorithms were necessary for supporting our attribute-based data model and to efficiently

Algorithm *Filter*

```
01: begin
02:   potentMatch ← Null
03:   eqMatch ← Null
04:   for each attribute C ∈ DAL(d) do                ▷ for all document attributes
05:     for each trie T with root(T) = w ∈ DWL(C) do  ▷ for all tries with roots the
                                                        distinct words in attribute
06:       for each node n ∈ T
07:         if word(n) ∈ OT(C) then                   ▷ if the word is contained in d
08:           if remainder(n) ⊆ OT(C) then            ▷ and the words of remainder(n)
                                                        (if any) are also contained in d
09:             potentMatch ← potentMatch ∪ query(n)  ▷ the query ids of this node are added
                                                        in the potentially matching queries
10:             n ← children(n)                       ▷ traverse the trie in DFS
11:           end if
12:         else prune n                              ▷ if the word is not contained in d
                                                        no need to search the sub-trie
13:         end if
14:       end for
15:     end for
16:     for each q ∈ potentMatch                      ▷ for all potentially matching
                                                        queries
17:       examine PA(C) for q                         ▷ evaluate proximity formulas
18:       if proximity is satisfied then
19:         Count[q] ← Count[q] + 1                   ▷ increase the counter for
                                                        matching attributes
20:       end if
21:     end for
22:     potentMatch ← Null                            ▷ reset for coping with
                                                        next attribute
23:     retrieve eqMatch from ET
24:     for each p ∈ eqMatch                          ▷ for all matching equalities
25:       Count[p] ← Count[p] + 1                     ▷ increase the counter for
                                                        matching attributes
26:     end for
27:   end for
28:   for i = 0 to N do
29:     if Total[i]=Count[i] then                     ▷ if matching attributes equal the
                                                        total number of attributes
30:       query i matches d
31:     else
32:       query i does not match d
33:     end if
34:   end for
35: end
```

Fig. 3. Pseudocode for filtering incoming documents.

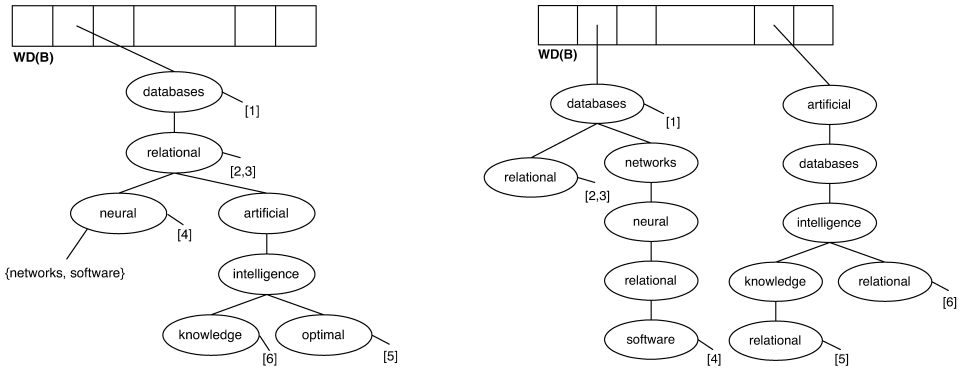
cope with the proximity operator that exists in our query language. BF (Brute Force) has no indexing strategy and scans the query database sequentially to determine matching queries. The BF algorithm stores the full query in a linked list using an appropriate representation that also facilitates the evaluation of the proximity operations. At publication time the incoming document is

matched against every single query in the list. The BF algorithm was mainly implemented to serve as the baseline in the comparison.

SWIN (Single Word INdex) is an extension of algorithm Key of Yan and Garcia-Molina [1994b]. It utilizes a two-level index for accessing queries in an efficient way. A query of the form of Definition 3.9 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 $wp_1, \ldots, wp_m$. SWIN utilizes an ET to index equalities and an AD pointing to several WDs to index the atomic containment queries. Queries of the form $B \sqsupseteq wp$ within a WD slot are stored in a linked list.

PrefixTrie is an extension of the algorithm Tree of Yan and Garcia-Molina [1994b] appropriately modified to cope with attributes and proximity information. This trie-based data structure was initially proposed based on the observation that users subscribing in the same area or topic, would probably use similar words to describe their information need. This would likely result in a large number of profiles with similar words. Using this observation, the authors showed that tries are able to store queries more efficiently than lists (as for example in SWIN). 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). This sorting of query words was a simple heuristic utilized by the proposed algorithm of SIFT to increase the common prefixes between submitted queries and enable the algorithm to store the queries more compactly and retrieve them faster at filtering time. After sorting the queries, a trie-based data structure was used to store the sorted sequences compactly by exploiting their *common prefixes* [Yan and Garcia-Molina 1994b].

Algorithm BestFitTrie constitutes an improvement over PrefixTrie. Because PrefixTrie examines only the prefixes of word sequences in lexicographic order to identify common parts, it misses many opportunities for clustering (see Figure 4). 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 tries to discover the best place to introduce a new set of words. This allows BestFitTrie to achieve better clustering as shown by the example in Figures 4(a) and 4(b), where we can see that BestFitTrie needs only one trie to store the set of words for the formulas of Table I, 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 Section 6. Additionally, variations of BestFitTrie (presented in Section 6.6), also utilize the sorting heuristic to further improve query indexing by storing the least frequent words towards the trie roots. This allows for pruning large portions of the trie at filtering time, since these words are, most likely, not contained in the incoming documents. Finally, all the trie-based algorithms use heuristics to identify and cluster similar queries. These heuristics provide an organization of queries that is dependent on the order of *insertion* of the queries in the system. As we will show in Section 5, this causes the loss of clustering opportunities, and algorithm ReTrie is proposed as a solution to this problem.



(a) BestFitTrie (b) PrefixTrie

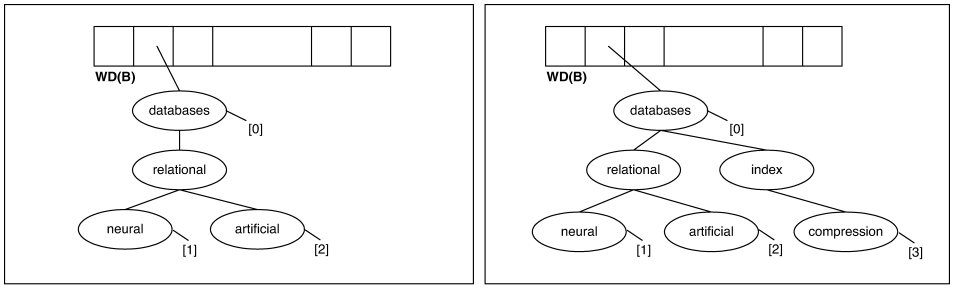
Fig. 4. BestFitTrie vs. PrefixTrie for the atomic queries of Table I.

Notice that the common assumption among the trie-based algorithms (PrefixTrie and BestFitTrie and all their variants) is the existence of many queries that have similar words. This scenario is very realistic given that users may share similar interests and have submitted their continuous queries to a central server. Additionally, important scheduled events (e.g., football finals, elections, etc.) or even unpredicted incidents (e.g., an earthquake or a terrorist act) can also cause the submission of lots of similar queries, by users interested to subscribe to the flow of information available for these events. These scenarios make the case for the usage of the trie-based algorithms even stronger, since they could relieve the system of a significant processing load by exploiting the similarities between queries to provide fast filtering times.

5. REORGANIZATION OF QUERIES

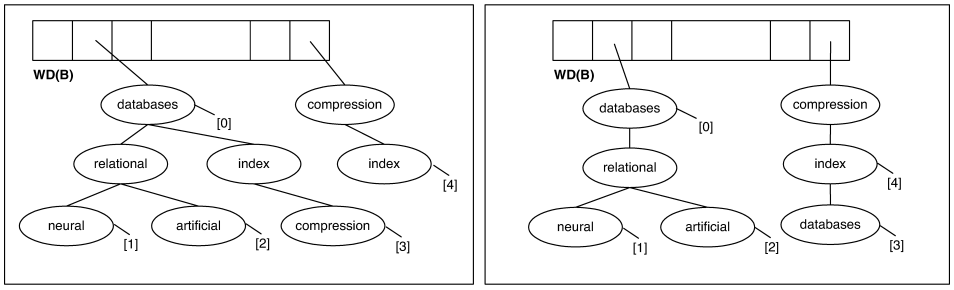
Most of the algorithms presented earlier use heuristics to identify and cluster similar queries, in order to achieve better performance during matching. These heuristics provide an organization of queries that is dependent on the order of *insertion* of the queries in the system. Taking BestFitTrie as an example, we can notice that for a given set of user queries Q, and two different orderings of these queries, the resulting clustering that is achieved is different. In other words, if we consider the clustering problem as a search problem over the search space of all possible query organizations, then BestFitTrie actually provides us with a greedy heuristic that results in a possibly nonoptimal solution (but yet a fairly good one, as we will see in Section 6). An alternative to organizing the user queries in a heuristic fashion is to search over the space of all possible organizations for the optimal one. This is obviously prohibitively expensive and will not be considered further in this article.

There are two main questions when trying to design an algorithm for reorganizing the database. The first question is what to reorganize and the second is when this reorganization should take place. We will discuss our approaches to address these two questions.



(a) A $WD(B)$ with only one trie and three sets of words (s_0, s_1 and s_2) stored

(b) The $WD(B)$ after the insertion of $s_3 = \{databases, index, compression\}$



(c) The $WD(B)$ after the insertion of $s_4 = \{compression, index\}$

(d) The $WD(B)$ after ReTrie has reorganized the index

Fig. 5. Query insertions and reorganization achieved by ReTrie.

The straightforward approach is to reorganize the database after a fixed number of query insertions or at fixed time intervals. This approach has the advantage of bringing the database in a "good shape" as often as we want, but has the drawback of ignoring the clustering criteria, and invokes reorganization even at cases where there might not be much need for it. In addition, periodic reorganization imposes a significant load in the system since queries have to be repositioned at given intervals. Another option is defining a criterion for the quality of clustering and reorganize when the clustering quality drops. One can design various different approaches for defining this criterion. For example, we can consider the quality of clustering in correlation with the number of redundant nodes in the forest. This approach exploits the fact that badly clustered queries introduce more redundant nodes (to see this consider the number of nodes in Figures 4(a) and (b), or Figures 5(c) and (d)). Another approach is to quantify the clustering quality in terms of the percentage of underclustered queries in the system. A third approach would be to use the fan-out of trie nodes to measure the clustering achieved by each algorithm. Trying to minimize node fan-out would then have an impact at document filtering time, as less branching would be required and a high number of sub-tries would be pruned. In all approaches discussed above, an appropriate threshold can be used to trigger the reorganization process. Finally, a simple, effective and widely used (in other related cases) approach is to consider reorganization when the system is idle.

This approach, apart from the obvious benefits of not overloading the system, offers the advantage that the reconstruction of the database can be done incrementally, which is useful if we want to exploit even small time intervals to do the heavy work of reorganization.

In this article we chose to implement and evaluate two of the approaches previously presented: a baseline approach that reorganizes the query database at given intervals (i.e., after a fixed number of query insertions), and an approach that uses a clustering criterion to decide when to reorganize the query database. Thus, we modify algorithm BestFitTrie to consider the reorganization of the query database and the modified algorithms we propose will be called Periodic and ReTrie respectively. Periodic is straightforward to implement and will serve as the baseline algorithm. The second approach, called ReTrie, uses the percentage of underclustered queries as the clustering quality metric that triggers the reorganization process. To describe ReTrie, we will need the following definition.

Definition 5.1. Let s be a set of words indexed at trie node n of trie T. The set of words s is considered *underclustered* iff $ClusteRat(s, T) < c$, where $0 \leq c \leq 1$ is a *clustering threshold*.

To keep track of the clustering ratio of each set of words s, ReTrie utilizes a *clustering array* (CA) that contains an entry for every set of words inserted in $WD(B)$. Each CA entry contains a pointer to the position s is currently stored at and a number representing $ClusteRat(s, T)$. When a new set of words s is indexed at node n of trie T, the clustering ratio of s in CA is initialized using the formula from Definition 4.3. Additionally, if $Remainder(n)$ is expanded to create new nodes, then the clustering ratios of the other sets of words stored at n should be updated, since they now have more of their words clustered. If $Remainder(n)$ is not expanded, no other update is necessary to array CA.

Algorithm ReTrie, presented in Figure 6, repositions badly clustered sets of words based on clustering quality metrics to achieve better query clustering. All underclustered sets of words (identified by scanning CA) are candidates for moving when the algorithm is executed. For each underclustered set of words s with clustering ratio $ClusterRat(s, T)$, ReTrie searches the forest of tries of $WD(B)$ looking for all nodes that can store s. For each one of these positions, ReTrie calculates the new clustering ratio for s and chooses the position that results in the maximum clustering ratio, $maxClu(s, T')$. s is moved to the new position found only if $ClusterRat(s, T) < maxClu(s, T')$. Moving s to a new position results in updates to array CA. The update procedure is the same as the one described in the previous paragraph for query insertions.

ReTrie is able to improve the clustering of queries because not all queries have the same clustering opportunities when entering the query index with BestFitTrie. This can be explained as follows. When a new set of words s corresponding to formula $B \sqsupseteq wp$ needs to be indexed, the clustering algorithm looks for a trie in the forest of tries of $WD(B)$ and a node in that trie, such that $ClusterRat(s, T)$ is maximized. It is easy to see that the number of tries and the number of nodes in those tries affect the clustering opportunities of s: the higher the number of candidate positions to insert s,

Algorithm *ReTrie*

```
01: begin
02:   for each attribute C do                          ▷ for all document attributes
03:     for i = 0 to N do                               ▷ for all stored sets of words
                                                          (N: # of stored sets of words)
04:       if CA[i].clusterRat < c then                  ▷ identify under-clustered ones
05:         let s be the i-th set of words
06:         for each trie T such that root(T) ∈ s do    ▷ for all candidate tries
07:           for each node n ∈ T such that word(n) ∈ s do  ▷ for all possible storage
                                                              positions in candidate tries
08:             calculate ClusterRat(s, T)
09:             if ClusterRat(s, T) > curClusterRat then  ▷ if a better position is
                                                              found make a note of it
10:               curClusterRat ← clusterRat(s, T)
11:               curPosition ← n
12:             end if
13:           end for
14:         end for
15:       end if
16:       if curPosition ≠ CA[i].position then          ▷ if the best position found is
                                                          better than the initial
17:         move s to curPosition                       ▷ move s there
18:         CA[i].position ← curPosition                ▷ and update CA
19:         CA[i].clusterRat ← curClusterRat
20:       end if
21:     end for
22:   end for
23: end
```

Fig. 6. Pseudocode for algorithm ReTrie.

the higher the possibility for the algorithm to cluster s effectively (i.e., in a position with a higher $ClusterRat(s, T)$). This is shown by the example of Figure 5. Consider a forest consisting of a single trie T currently indexing three sets of words: $s_0 = \{databases\}$, $s_1 = \{databases, relational, neural\}$ and $s_2 = \{databases, relational, artificial\}$ (Figure 5(a)). When a set of words $s_3 = \{databases, index, compression\}$ arrives, it is inserted in the only position available and is clustered only under one word (Figure 5(b)). Now $ClusterRat(s_3, T) = 0.33$. Upon arrival of $s_4 = \{index, compression\}$, a new trie is created, since s_4 cannot be indexed in the existing trie, so as to create a forest of tries as the one shown in Figure 5(c). It is however obvious that there is a better position to index s_3 and this position is together with s_4, as it is shown in Figure 5(d) where $ClusterRat(s_3, T) = 0.66$. Notice that in the forest of tries of $WD(B)$ shown in Figure 5(c), words *index* and *compression* appear in two nodes each (this *redundancy* in nodes is one of the factors that slow down filtering), whereas after the reorganization of the forest (Figure 5(d)) there are no redundant nodes for these words (i.e., they appear only once in the forest). Generally, it is not possible to remove all redundant nodes in a forest (notice for example the word *database* in this example), so our effort concentrates on minimizing these nodes by reorganization.

Table II. Some Characteristics of the NN Corpus

Description	Value
Number of documents	10,426
Document vocabulary size	641,242
Maximum document size (words)	248,609
Minimum word size (letters)	1
Maximum word size (letters)	35

Table III. Some Attribute Characteristics of the Corpus Documents

Attribute	Percentage of Documents
TITLE	63%
AUTHORS	58%
ABSTRACT	88%
BODY	86%
YEAR	63%

(a)

# of Attributes	Percentage of Documents
1	7.9%
2	28.7%
3	2.4%
4	16.0%
5	45.0%

(b)

6. EXPERIMENTAL EVALUATION

To carry out the experimental evaluation of the algorithms described in the previous sections, 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 article, there are various document sources that one could consider: metadata for papers on various publisher Web sites (e.g., ACM or IEEE), electronic newspaper articles, articles from news alerts on the Web[3] etc. However, it is rather difficult to find user queries except by obtaining proprietary data (e.g., from CNN's news alert system).

6.1 Dataset and Experimental Setup

For our experiments we chose to use two sets of documents. The first set is composed of 10426 documents downloaded from ResearchIndex[4] and originally compiled in Dong [2002]. The documents are research papers in the area of Neural Networks, and we will refer to them as the NN corpus.[5] Table II summarizes some key characteristics of the document corpus, where Tables III(a) and III(b) give details on the fraction of documents that contain each attribute, and on the fraction of documents that contain a specific number of attributes respectively. This document set was chosen because it provides a focused setting that allows us to exploit the common words between queries to achieve better clustering. As also pointed out in Section 4.2, this scenario corresponds to users that are interested in the same area and submit their continuous queries in a central server that also publishes documents from this area. To assess the limitations of our approach we have also conducted experiments with a wider and

[3]http://www.cnn.com/EMAIL

[4]http://www.researchindex.com

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

more varied corpus consisting of documents that are web sites crawled from the .gov domain (we will refer to this corpus as the .GOV corpus). All the experiments shown in this section were carried out with both document corpora. Since the NN corpus is closer to our assumptions about having a thematically focused setting, Sections 6.2 to 6.7 describe the experiments carried out with this corpus. The description of the .GOV corpus along with the most interesting experiments is given in Section 6.8.

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 [Frantzi et al. 2000]. The extracted multiword 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 attributes *paper title, authors, abstract*, and *body*.

The basic concept for the query creation in our methodology is that of a *unit*. Word patterns for atomic queries (i.e., queries of type $C \sqsupseteq wp$) are created as conjunctions of units selected uniformly from unit sets, whereas queries are created as conjunctions of atomic queries with attributes selected from the attribute universe with a probability p_C. In our scenario four different types of unit sets exist:

—*Author unit set*. This set contains the last names of authors appearing in the full citation graph of ResearchIndex.[6] 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 multiword 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 multiword 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.

[6]The citation graph contains information about the citations between research papers and was compiled in Milios et al. [2003].

Table IV. Parameters Varied in Experiments and Their Descriptions

Parameter	Description
W	average # of words per document
W_d	average # of distinct words per document
N	# of queries in database
D	# of incoming documents
m	percentage of stored queries matching the incoming documents
c	clustering threshold

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

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

For more details of the methodology the interested reader can refer to Tryfonopoulos and Koubarakis [2002]. As a byproduct of our work on this topic, we have a new corpus of documents and continuous queries which is representative of digital library scenarios, and can also be used by other researchers in the area of information filtering.

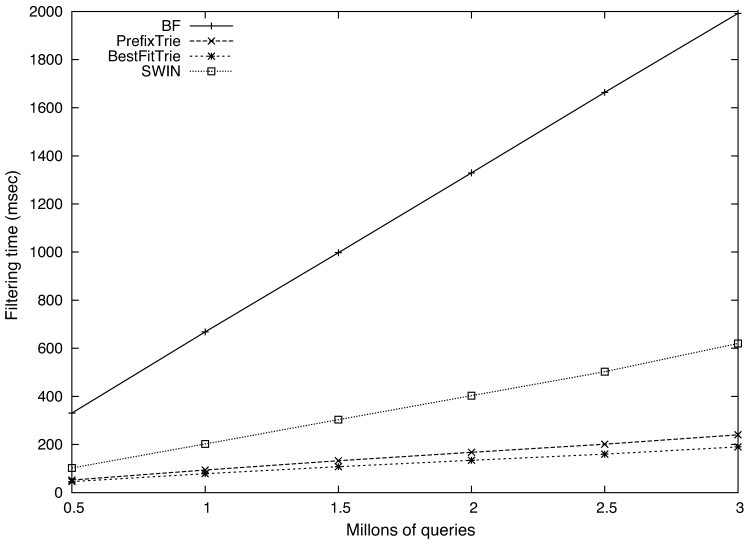
All the algorithms were implemented in C/C++, and the experiments were run on a PC, with a Pentium III 1.7GHz processor, with 1GB RAM, running Linux. Time shown in the graphs is wall-clock time and the results of each experiment are averaged over 10 runs to eliminate any fluctuations in the time measurements. Table IV summarizes the parameters examined in our experimental evaluation.

6.2 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 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.

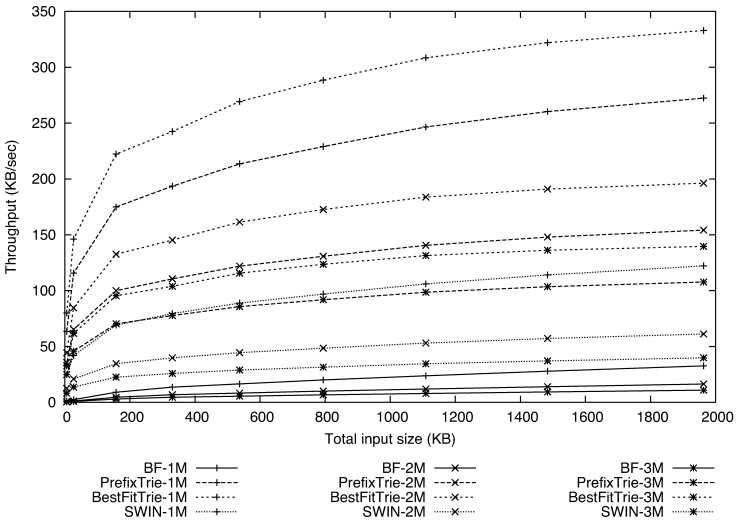
As we can see in Figure 7, the time taken by each algorithm grows linearly with the size of the query database. However SWIN, PrefixTrie and BestFitTrie are less sensitive than Brute Force to changes in the query database size. The trie-based algorithms outperform SWIN mainly due to the clustering technique that allows the exclusion of more non-matching atomic queries at filtering time. We can also observe that the better exploitation of the commonalities between queries improves the performance of BestFitTrie over PrefixTrie, resulting in a speedup in filtering time for *large query databases*. Additionally, Figure 8 contrasts the algorithms in terms of throughput where 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 memory requirements, BF needs about 50% less space than the trie-based algorithms, due to the simple data structure that poses small



Parameter	Value
W	6869
W_d	1115
N	0.5M-3M
D	100
m	1%
c	-

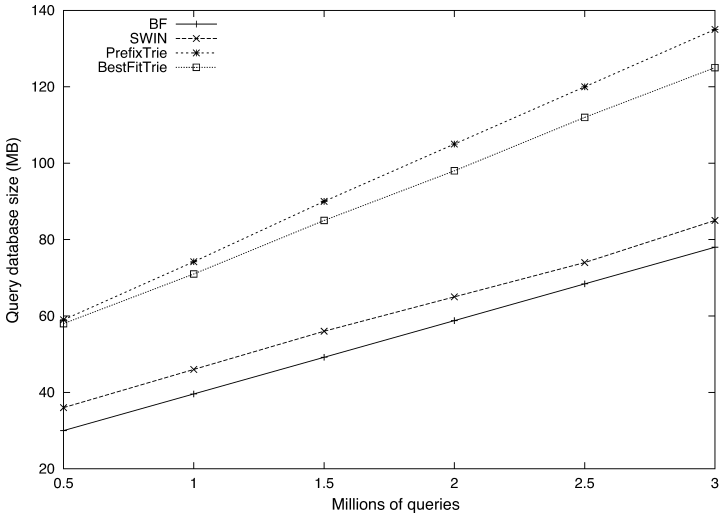
Fig. 7. Effect of the query database size in filtering time.



Parameter	Value
W	6869
W_d	1115
N	1M-3M
D	100
m	1%
c	-

Fig. 8. Performance in terms of throughput for the algorithms of Section 4.

memory 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. Figure 9 shows the memory requirements of the trie-based algorithms in comparison to the BF and SWIN approaches. 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.



Parameter	Value
W	-
W_d	-
N	0.5M-3M
D	-
m	-
c	-

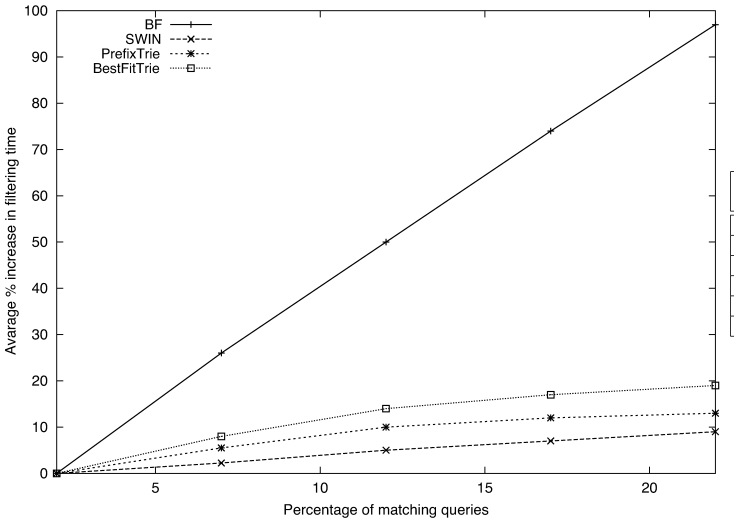
Fig. 9. Memory requirements for the trie-based algorithms.

6.3 Varying the Matching Percentage

In this experiment we wanted to observe the sensitivity of each algorithm when the number of queries that matched incoming documents varies. To do this, we used five document sets, S_1 to S_5, that contained about the same number of distinct words and the same attributes, but the number of queries that matched each document set was different. Document set S_1 was used as the baseline and the % increase in filtering time for the rest of the document sets was calculated in relation to set S_1. Notice that given 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,[7] 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 Koubarakis et al. [2002], the majority of the positions of a specific word in a document can be excluded from the proximity evaluation.

Figure 10 shows the percent increase in matching time with respect to the baseline document set S_1, against the percentage of matching queries for all algorithms, for document sets that match from 2% to 22% of the indexed queries. All document sets contained three or four (*attribute*, *value*) pairs, and the query database contained 3 million queries. Apart from BF, which showed an overall

[7]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.



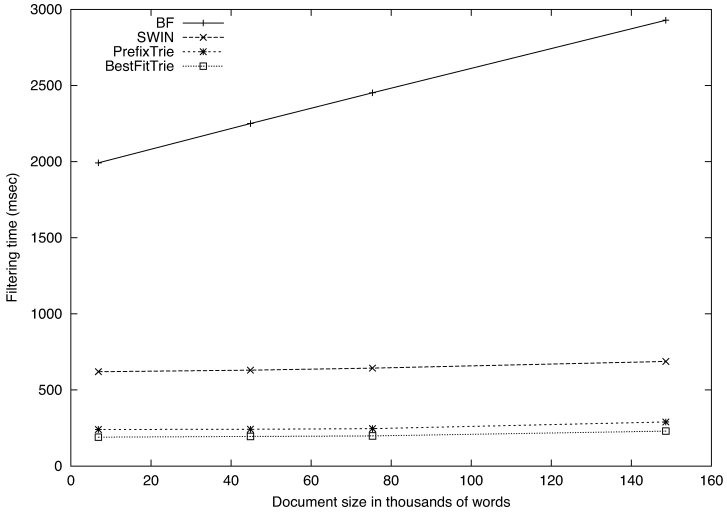
Parameter	Values for sets S_i, $1 \leq i \leq 5$
W	varying
W_d	2920-3098
N	3M
D	10
m	2-22%
c	-

Fig. 10. % increase in filtering time against the % increase in the number of matching queries.

97% increase in the matching time, BestFitTrie appears to be the most sensitive to the increase in the matching percentage (showing an overall of 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 faster, with BestFitTrie being the faster algorithm of all four despite the high increase (the filtering time for the algorithms was 771 msecs for SWIN, 322 msecs for PrefixTrie and 293 msecs for BestFitTrie for set S_5).

6.4 Varying the Document Size

The purpose of this experiment is to observe the behavior of the various algorithms with respect to the incoming document's size. This time four sets of documents (S_1 to S_4), with sizes varying from about 7K words (1115 distinct words) to 148K words (2304 distinct words), were chosen. These documents had about the same number of queries matching them to allow for comparisons with respect to document size in the performance of the four algorithms. Performance differences are shown in Figure 11 and are 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 less than the



Parameter	Value for sets S_i, $1 \leq i \leq 5$
W	6869 - 148609
W_d	1115 - 2304
N	3M
D	10
m	2%
c	-

Fig. 11. Increase in filtering time when document size grows.

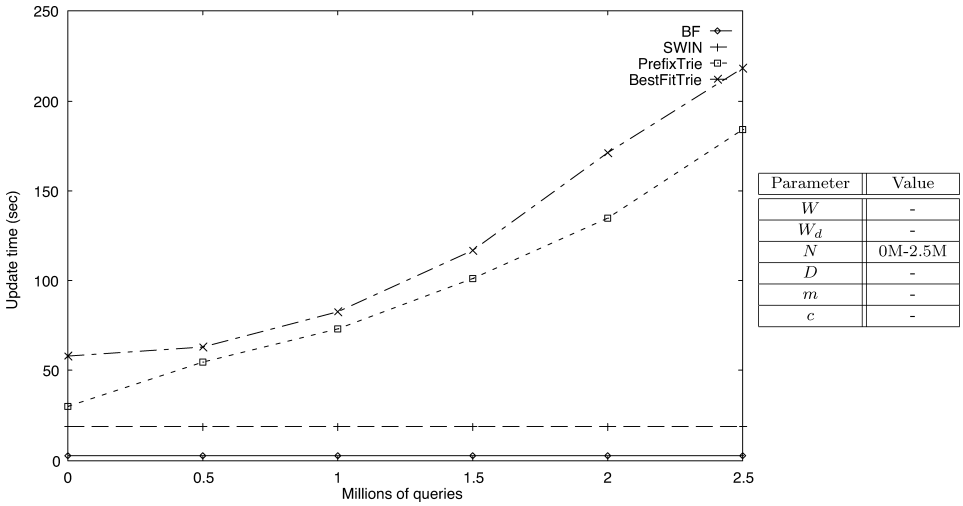
document itself for large documents), and check the existence of a word in the document using a hash function, which provides fast answer times. Moreover the proximity evaluation is not greatly affected from the large number of word positions inside a document due to the well-designed proximity evaluation algorithm of Koubarakis et al. [2002] that allows the omission of a large number of word positions in a document.

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 of smaller size (see also our evaluation with a different corpus of smaller documents in Section 6.8). 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).

6.5 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 batches of 500K queries in our database and measuring the total insertion time per batch. Subsequently the average insertion time per query for a given batch of queries can be found simply by dividing the total time measured with the population of the batch to produce a single point in the graph of Figure 12.

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 utilizes a hash table and inserts each atomic query in the beginning of



Parameter	Value
W	-
W_d	-
N	0M-2.5M
D	-
m	-
c	-

Fig. 12. Query insertion time for different query database sizes.

the list pointed to by the hash table slot. For the 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 query). Our remarks are verified by the graph in Figure 12, which 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 queries. This is a standard tradeoff where the algorithm spends some extra time at indexing to save it at query execution.

6.6 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. 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 [Yan and Garcia-Molina 1999] (say k most recent documents arrived, where k is large enough). Using this 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*) [Yan and Garcia-Molina 1999] 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

$$N = D(w_1)N_1 + D(w_2)N_2 + \cdots + D(w_{|V|})N_{|V|}, \tag{1}$$

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

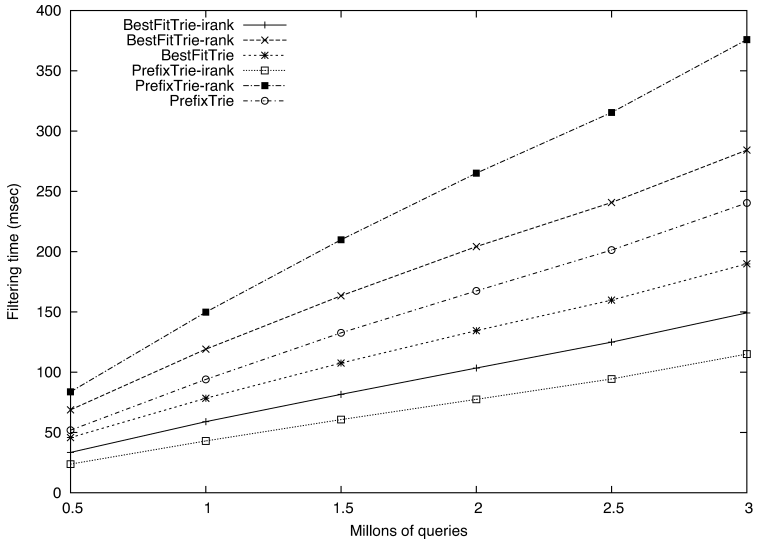
$$N_1 + N_2 + \cdots + N_{|V|} \tag{2}$$

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 minimized 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 Trie) 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. In 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 organized following the insertion algorithm of BestFitTrie (this will give us the best organization considering only this trie instead of the whole forest).

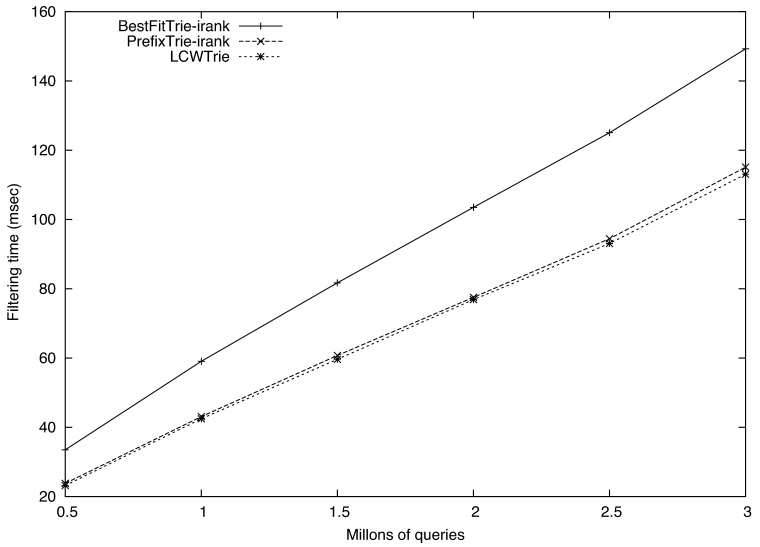
In Figure 13 we present the performance of PrefixTrie and BestFitTrie and their ranking variations. We can see that using the rank heuristic the performance of both algorithms deteriorates, due to the creation of large tries that need bigger exploration time. We can also observe that the irank heuristic 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 14 presents the performance of the three faster algorithms, namely PrefixTrie-irank, BestFitTrie-irank and LCWTrie. BestFitTrie-irank prioritizes 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. Both algorithms



Parameter	Value
W	6869
W_d	1115
N	0.5M-3M
D	100
m	1%
c	-

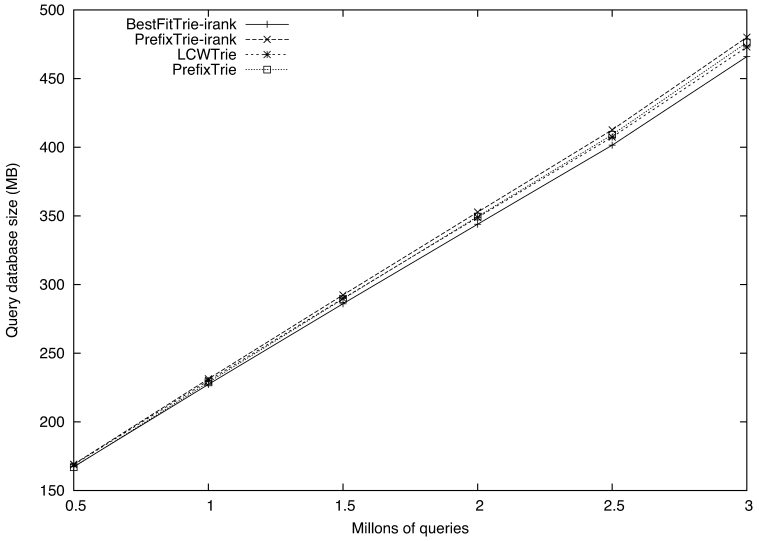
Fig. 13. Incorporating word frequency information into the trie-based algorithms, and its effect in filtering time.



Parameter	Value
W	6869
W_d	1115
N	0.5M-3M
D	100
m	1%
c	-

Fig. 14. Performance of LCWTrie in comparison to the two faster filtering algorithms.

examine exactly one candidate trie, that with the least frequent word as root. Additionally, LCWTrie organizes 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 take clustering into account. Consequently, it stores the query according to frequency information only, that is the word with the lowest rank goes deeper in the trie.



Parameter	Value
W	-
W_d	-
N	0.5M-3M
D	-
m	-
c	-

Fig. 15. Memory requirements of ranking variations of BestFitTrie and PrefixTrie.

Our observations about the significance of frequency information presented in the beginning of the section are verified. From the experiments of Figure 14, 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. Figure 15 shows the memory requirements of the different algorithms for varying database sizes. The improvement in clustering using the heuristics discussed here is obvious. All algorithms that exploit frequency information need less space to store the same number of queries compared to PrefixTrie. This difference in storage requirements comes from the existence of less redundant nodes.

6.7 Reorganization of Queries

In the experiments described in this section, we ran the algorithm BestFitTrie against the reorganization algorithms Periodic and ReTrie.[8] Notice that the ranking counterparts of BestFitTrie presented earlier (namely BestFitTrie-rank and -irank) are not suitable for comparing with our reorganization algorithms, because of the non flexible way of clustering they use. The rank and irank heuristic do not allow for many alternatives to cluster a set of words, resulting in the inapplicability of the ReTrie algorithm. For the same reasons, ReTrie cannot be used in conjunction with PrefixTrie. Thus, in this section we study the effect of our reorganization strategies to the performance of the algorithms at filtering time by comparing algorithms ReTrie, Periodic and BestFitTrie.

[8]We remind the reader that Periodic and ReTrie, use BestFitTrie as their filtering component.

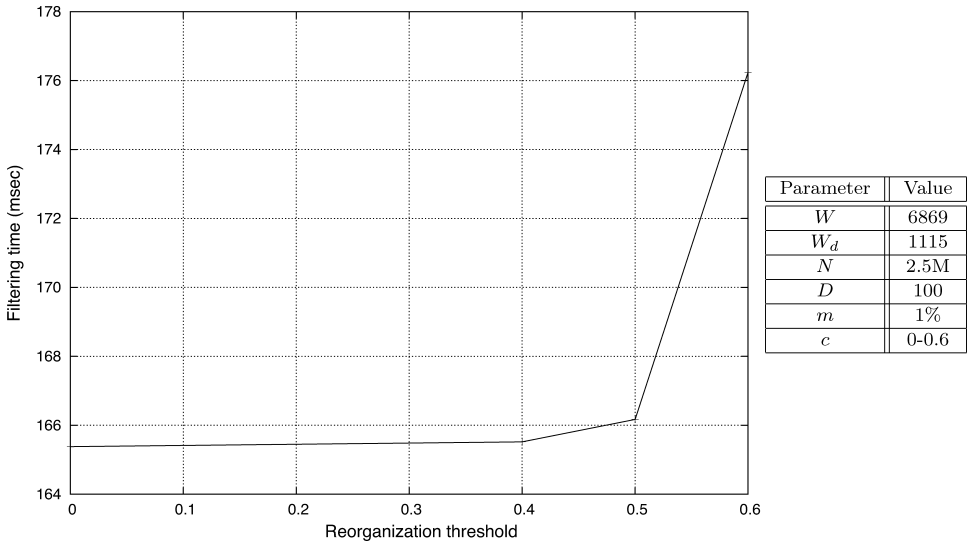
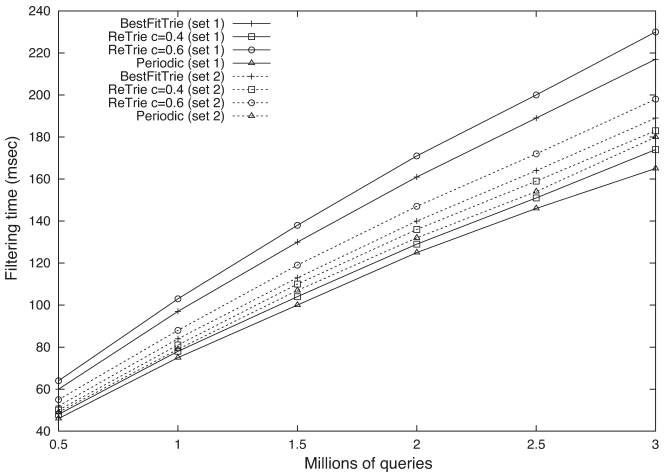


Fig. 16. Filtering time for different clustering thresholds.

6.7.1 *Varying the Clustering Threshold.* In this experiment we wanted to determine a baseline value for the clustering threshold c to be used in the evaluation of ReTrie. To do so, we populated the query database db with 2.5 million queries and invoked ReTrie with different reorganization thresholds ranging from 0 to 0.6 with an increase step of 0.1. For each reorganization threshold, a different reorganized database db' was created, since different sets of words were chosen to be moved. Then, we randomly selected one hundred documents from the NN corpus as incoming documents to be matched against each one of the different databases. The average filtering time for each one of the different clustering thresholds is shown in Figure 16. We observe that after a threshold value of around 0.4, the filtering time shows a slight increase. As we move to even higher values of c, the increase in filtering time is even sharper. This can be explained as follows. At high values of c, sets of words with a high clustering ratio are also moved. These sets usually contain words that are often in user queries, and thus create subtries that store a large number of such sets. Moving highly clustered queries results in the perturbation of large tries that need to recluster, probably in a position with a lower clustering ratio. In this way, by moving some already highly clustered queries to an even better position, we disturb the clustering of many other queries that subsequently cluster at a position with lower clustering ratio. Based on the results shown in Figure 16, we chose 0.4 as the clustering threshold to be used in the subsequent experiments.

6.7.2 *The Effect of Reorganization.* The second experiment targets the effect of the two reorganization strategies we have implemented and compares their performance against BestFitTrie. In this experiment we populated the query database db with different numbers of queries (ranging from 500K to 3M). We then used algorithms Periodic (with a reorganization period of 10K query insertions) and ReTrie (with clustering thresholds 0.4 and 0.6) to



Parameter	Value for set S_1	Value for set S_2
W	6702	6869
W_d	1004	1115
N	0.5M-3M	0.5M-3M
D	100	100
m	2.5%	2%
c	0.4, 0.6	0.4, 0.6

Algorithm	# of reorganizations	timefor reorganizations
Periodic	300	284 min
ReTrie ($c = 0.4$)	26	42 min
ReTrie ($c = 0.6$)	26	68 min

Fig. 17. Performance of algorithm ReTrie for different clustering thresholds and sets of documents.

reorganize the queries stored in the database *db*. Subsequently, we chose two document sets and used them as incoming documents to test the efficiency of the BestFitTrie against the reorganization algorithms Periodic an ReTrie for databases of different size.

The two different sets of documents we used, namely S_1 and S_2, were chosen from the NN corpus and contained one hundred documents each. Set S_1 was chosen so as to contain many words that are also contained in the under-clustered queries that algorithm ReTrie has chosen to move. This set is used to demonstrate the performance gain of algorithm ReTrie at times where similar documents arrive at high rates, for example, imagine a scenario where the above algorithms are used in a news alert system and a sudden crisis occurs (e.g., an earthquake or terrorist act). Set S_2 is a randomly chosen set of documents from the NN corpus. This set is used to demonstrate the performance of the algorithm in the standard setting used to conduct the rest of the experiments.

Figure 17 shows the performance of ReTrie for two clustering thresholds ($c = 0.4$ and $c = 0.6$) in comparison with BestFitTrie and Periodic described above. Initially one can observe that the results of the experiments in Section 6.7.1 and also our choice of 0.4 as a value for c are verified. The filtering performance of algorithm ReTrie for $c = 0.6$ is worse not only from its counterpart with $c = 0.4$, but also from algorithm BestFitTrie. The reasons for this are explained in the previous section. The performance of the algorithms is also affected by the document set used as publications. We can see that ReTrie is better than BestFitTrie in cases where many similar documents get published during a short time interval, whereas the gain in filtering performance is small when a random document set is used. This can be explained as follows. In the case of random documents the improvement in clustering that is achieved by ReTrie is not exploited since not many of the reclustered subtries are used, due to the statistical properties of the document set (many diverse subjects, larger vocabulary, etc.). On the other hand, in the case of similar documents, the

clustering improvement leads to a performance benefit since the optimized subtries are used more often and lookups in $OT(d)$ are reduced. In general, from the set of experiments we conducted with document sets of different statistical properties, we observed improvements in filtering performance ranging from 2% to 18% for clustering thresholds ranging from 0.1 to 0.4 respectively. The time needed to reposition an underclustered set of words was similar to the update time shown in the experiments of Section 6.5.

Algorithm Periodic is marginally faster than ReTrie for both document sets. This is attributed to the periodic reorganization of the query database that better exploits the clustering opportunities. The more frequent these database reorganizations are, the higher the clustering achieved and the shorter the filtering time. In the runs shown in Figure 17, the Periodic algorithm was executed every 10K query insertions. This marginal benefit (compared to the performance of ReTrie with $c = 0.4$) comes at the cost of extra computational effort imposed to the system. The table of Figure 17 compares the algorithms in terms of computational effort to reorganize the query database. There, for each algorithm, we measure the total time over all reorganizations needed to move the queries to a better position. Contrary to ReTrie that chooses to reorganize the database based on a clustering quality metric, Periodic is more active and thus more computationally demanding, showing the tradeoff between the two approaches. Notice that Periodic is less than 4% faster at filtering time (as shown in the graph of Figure 17), while needing as much as 7 times more time to reorganize the query database.

6.7.3 *Memory Requirements.* ReTrie needs about 22MB more memory than the rest of the trie-based algorithms for a database of 3M queries (see also Figure 9), due to the size of *CA*. This amount, although small compared with today's main memory capacities, is about one third of the total memory requirements for the rest of the trie-based algorithms. This increase in memory usage is the cost for the faster database reorganization and filtering times achieved by ReTrie. Therefore, the developer of an information filtering system can choose between ReTrie or Periodic depending on resource availability and which resource he wishes to optimize.

6.8 Performance with a Different Document Corpus

To assess the performance of the algorithms in a dataset that is wider and not thematically focused, we repeated all the experiments presented in the previous sections with the .GOV corpus. The documents in this corpus are Web sites crawled from the .gov domain in early 2002. The corpus contains around 1.25M documents with an average document length of 974 words. Table V summarizes some key characteristics of the corpus. Since no continuous queries were available for this document corpus, we used the methodology described in Section 6.1 to construct the query database. The experiments with the .GOV corpus were conducted using the same machinery as before, to enable the comparison across the different settings. In this section we summarize our observations from the experimentation with this corpus and present the most interesting graphs that show the performance of the algorithms under this new setting.

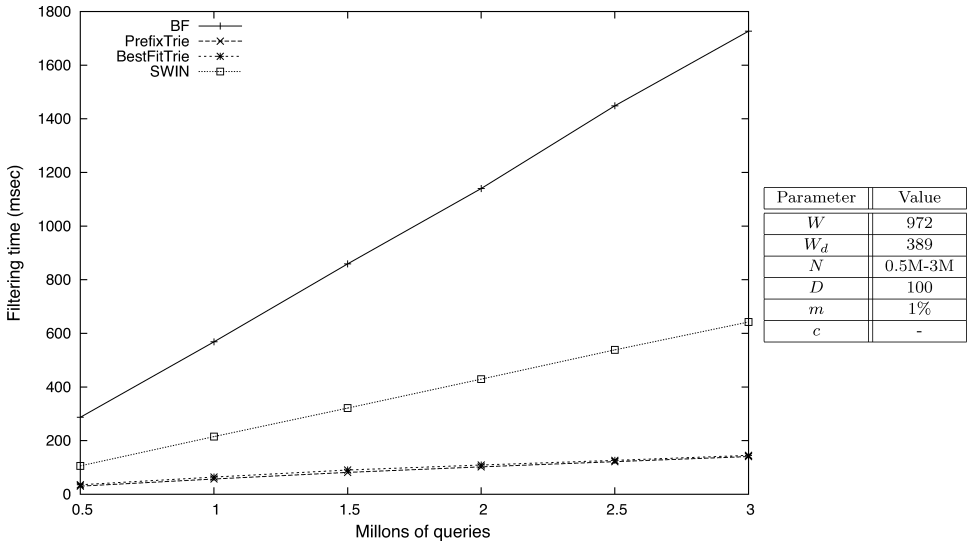
Table V. Some Characteristics of the .GOV Corpus

Description	Value
Number of documents	1,247,753
Document vocabulary size	9,946,052
Maximum document size (words)	26,789
Minimum word size (letters)	1
Maximum word size (letters)	42

The main assumption behind the clustering algorithms, is that the submitted queries will present commonalities among the words they contain. These commonalities are then exploited by the trie-based algorithms to allow clustering of queries under their identifying subsets. The NN corpus used in the previous experiments, was a focused dataset that contained documents about the same topic (i.e., neural networks). This focus was naturally depicted in the constructed queries, since they contained a lot of common words among them. This offered a lot of clustering opportunities to the trie-based algorithms, and showed their advantages and disadvantages in such a setting. Contrary to the NN corpus of focused scientific papers, the .GOV corpus is comprised of documents crawled from the web with a wide variety of topics and thus a wider and less focused vocabulary. This resulted in the construction of queries that did not have a lot of common words among them, thus restricting the clustering opportunities of the trie-based algorithms. This diversity in topics (ranging from security announcements and court decisions, to governmental laws and pages with how-to instructions for governmental employees), vocabulary and corpus size were the reasons for the selection of this corpus, which was meant to be a stress test for the filtering performance of the proposed algorithms. To compare the two corpora in terms of some quantitative characteristics one should refer to Tables II and V.

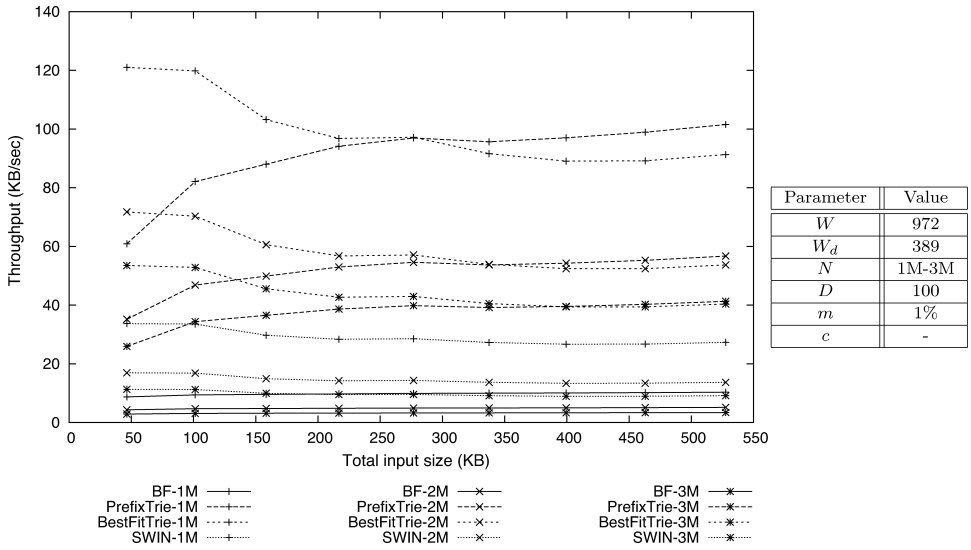
Figure 18 shows the performance of the algorithms BF, SWIN, PrefixTrie and BestFitTrie under different sizes of the query database. In this experiment we randomly selected one hundred documents from the .GOV corpus and used them as incoming documents in query databases of different sizes. The details of the setting are summarized in the table of Figure 18. Similarly to the NN corpus (Figure 7), the filtering time of all algorithms is linear to the database size, with the trie-based algorithms being faster than BF and SWIN. Notice that, in this setting, BestFitTrie is marginally better that PrefixTrie, even for large query databases. This can be explained as follows. Due to the lack of focus in the document corpus, and thus its large vocabulary, the constructed queries do not have many words in common. This results in less clustering opportunities for the queries, and thus in similar clustering performance for both PrefixTrie and BestFitTrie. Finally notice that since the average document size in the .GOV corpus is smaller compared to the NN corpus, the algorithms need less filtering time per incoming document (compare the filtering times shown in Figure 18 with those of Figure 7).

Figure 19 presents the throughput of the different algorithms with respect to the total input size. As we can observe, for small query databases the trie-based algorithms achieve high throughput, which is naturally reduced as the



Parameter	Value
W	972
W_d	389
N	0.5M-3M
D	100
m	1%
c	-

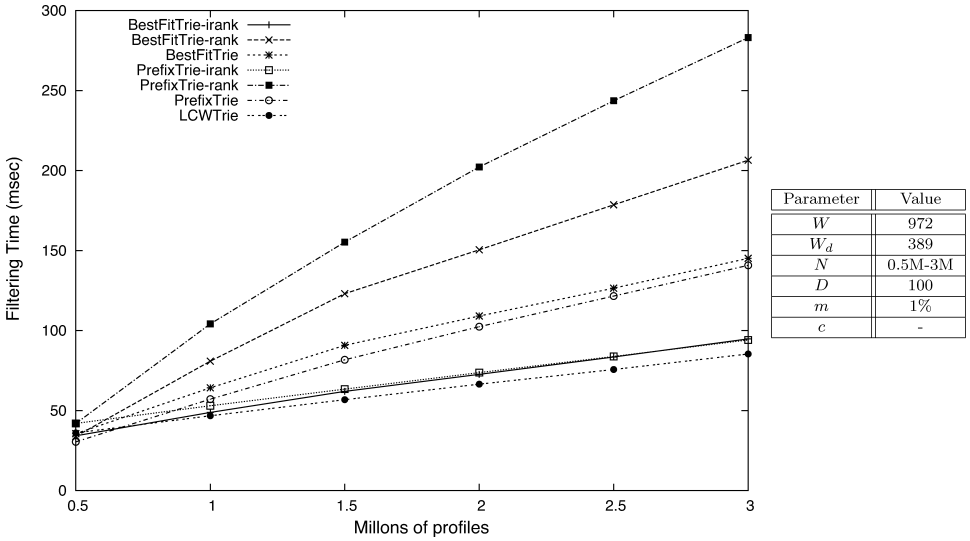
Fig. 18. Effect of the query database size in filtering time.



Parameter	Value
W	972
W_d	389
N	1M-3M
D	100
m	1%
c	-

Fig. 19. Performance in terms of throughput for the algorithms of Section 4.

query database size increases. Additionally, for small query databases algorithm PrefixTrie achieves higher throughput than BestFitTrie, while as the database grows, BestFitTrie manages to exploit the opportunities for better query clustering and improves over PrefixTrie, managing to achieve higher throughput. Finally, in terms of memory consumption, our experiments with the .GOV corpus showed that the behavior of all algorithms is similar to that observed with the NN corpus, with the trie-based algorithms needing a constant amount of extra space for their index structures compared to BF and SWIN.

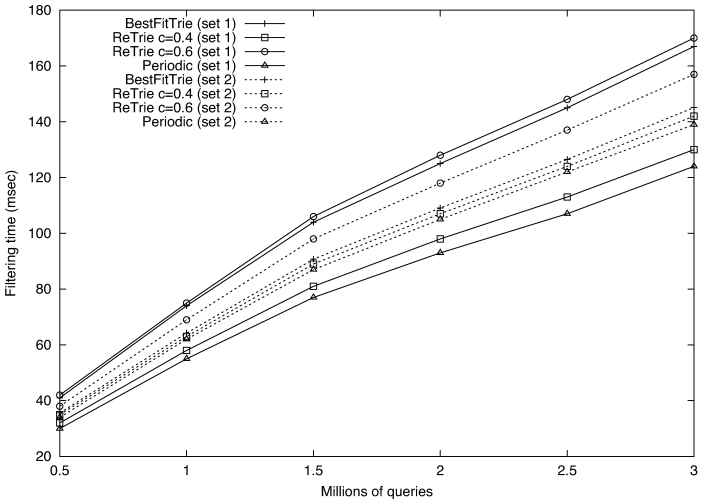


Parameter	Value
W	972
W_d	389
N	0.5M-3M
D	100
m	1%
c	-

Fig. 20. Incorporating word frequency information into the trie-based algorithms, and its effect in filtering time.

Figure 20 presents the filtering performance of the trie-based algorithms together with their ranking variations and LCWTrie for the .GOV corpus runs. Again, for the reasons explained in detail in Section 6.6, using the rank heuristic causes the filtering performance of both PrefixTrie and BestFitTrie to deteriorate. Similarly, using the irank heuristic improves the performance of both trie-based algorithms as expected, since larger parts of the tries can be pruned at filtering time. Notice that both PrefixTrie-irank and BestFitTrie-irank perform similarly in this set of experiments. This can be explained as follows. Since the vocabulary used to construct the queries is large and varied, queries do not present many clustering opportunities, since they do not contain many words in common. This lack of clustering opportunities causes the two irank variants of the trie-based algorithms to perform similarly, since the margin for BestFitTrie-irank to further improve the clustering achieved by PrefixTrie-irank is very small. Finally, notice that LCWTrie is the fastest of all ranked alternatives, contrary to the NN corpus runs where it was only marginally better (see Figure 14). This difference in the performance of LCWTrie is attributed again to the lack of clustering opportunities for the indexed queries. Since not many queries share the same words, clustering cannot achieve a big improvement at filtering time. This gap is filled by LCWTrie, which exploits the frequency information available and indexes the queries in the trie that has as root the least frequent word in the query. This indexing scheme helps in pruning a big number of tries, thus speeding up the filtering process.

To study the effect of our reorganization strategy we compared algorithm ReTrie with algorithms Periodic and BestFitTrie. This set of experiments was set up similarly to the one described in Section 6.7.2. We chose two sets of documents from the .GOV corpus, containing one hundred documents each. Set S_1 was chosen so as to contain many words that are also contained in the



Parameter	Value for set S_1	Value for set S_2
W	961	972
W_d	370	389
N	0.5M-3M	0.5M-3M
D	100	100
m	2%	1%
c	0.4, 0.6	0.4, 0.6

Algorithm	# of reorganizations	time for reorganizations
Periodic	300	192 min
ReTrie ($c = 0.4$)	26	31 min
ReTrie ($c = 0.6$)	26	52 min

Fig. 21. Performance of algorithm ReTrie for different clustering thresholds and sets of documents.

under-clustered queries, where set S_2 consisted of randomly chosen documents from the .GOV corpus. The first set intended to demonstrate the performance gain of ReTrie in settings where similar documents arrive at high rates, whereas the second set was representative of the standard setting used in the rest of the experiments.

Figure 21 shows the performance of the filtering algorithms for different database sizes, and for the two documents sets described above. Notice that for both document sets the Periodic algorithm is the fastest. This was expected, since this periodic query reorganization exploits well all clustering opportunities and manages to organize similar queries together. However, this comes at a high computational cost, since the complete query database is reorganized every 10K query insertions (for the experiments conducted here algorithm Periodic reorganized the query database 300 times). On the other hand, algorithm ReTrie is slightly slower than Periodic, but manages to achieve fast filtering times by resorting on the clustering quality metrics defined in Section 5 and avoiding expensive periodic database reorganizations (see the table of Figure 21). The computational cost for Periodic comes from the frequent reorganization of the database, contrary to ReTrie that chooses to reorganize only those queries that are not sufficiently clustered. Finally, notice that the difference in the filtering performance of Periodic and ReTrie is higher for document set S_1. This was expected, since the incoming documents where chosen to contain many words from the underclustered queries. Thus, achieving a better indexing for these queries is expected to improve the filtering performance of Periodic over ReTrie. Similarly, on set S_2, which is chosen to fit our standard document setting, the difference between the two algorithms is smaller.

6.9 Summary of Results

The experiments conducted in this section demonstrate the strengths and weaknesses of all algorithms. When no frequency information of word occurrences

in documents is available, BestFitTrie performs as much as 20% faster than prior work in the literature (achieving also a higher throughput rate). Additionally, BestFitTrie remains relatively unaffected to the increase in document size, compared to the rest of the algorithms. This improvement comes at the cost of a small constant increase in storage cost compared to brute force approach, and a small increase in query insertion time, which is a standard tradeoff in such a setting. Additionally BestFitTrie presents a sensitivity in the increase of the matching percentage, compared to the rest of the algorithms, but experiments show that it remains faster despite this increase.

When word frequency information is available, variations of the original algorithms (namely BestFitTrie-irank and PrefixTrie-irank) are reported to perform faster. A new algorithm introduced, called LCWTrie, performs slightly better that the previous algorithms, with no extra storage cost associated. When utilizing a wider and more varied corpus that does not contain many similar queries, then LCWTrie performs even better by exploiting the word frequency information. Finally, algorithm ReTrie extends the proposed algorithms by resorting in a clustering quality metric, and reorganizing the query database when this metric drops below a threshold. While the rest of the algorithms give a greedy, static solution to the problem of query database organization, ReTrie considers the reorganization of the database to achieve better performance at filtering time. Experiments show that the parameters regulating this reorganization process (e.g., the clustering threshold) should be carefully tuned and tailored to the needs of the specific scenario. Should this be done, further improvements of around 15% compared to BestFitTrie are reported.

7. CONCLUSIONS AND OUTLOOK

In the last years, main memory has become competitive in price with the disk storage of a few years ago. Multi-gigabyte main memories have become easily affordable and expandable, allowing applications with as much as 1 or 2 GB of data in main memory to be built with relatively inexpensive systems. In this work we have presented efficient main-memory algorithms suitable for large scale information filtering with queries supporting Boolean and proximity operations on attribute values. Indexing millions of users queries requires only a few hundred megabytes, allowing us to store up to twenty to thirty million user queries on a single off-the-shelf machine. Additionally, extensive experimental evaluation shows that our algorithms are able to filter incoming documents up to 20% faster compared to known alternatives in the literature. In scenarios where word frequency information can be collected, variations of the original algorithm can exploit it to speed up filtering. Finally, since all proposed algorithms utilize a heuristic solution to query clustering, variations that reorganizes the query database when query clustering degrades has been proposed.

In the future, we plan to extend our work on the problem of reorganizing the query database to identify alternative criteria that will reflect the clustering quality achieved and trigger the reorganization process. Additionally, we would like to use the lessons learned in this work to extend our approach to richer data models and more expressive query languages. Our goal is to

look into information filtering with data models based on XML and queries based on XQuery/XPath with IR features (phrases, word proximity, similarity etc.) [Amer-Yahia et al. 2004; Chinenyanga and Kushmerick 2001; Fuhr and Großjohann 2004; Theobald and Weikum 2000].

ACKNOWLEDGMENTS

We would like to thank Evangelos E. Milios and his group at Dalhousie University for providing us the original Neural Network Corpus. Special thanks go to Theodoros Koutris and Paraskevi Raftopoulou for their help in the processing of the corpus and for reading previous versions of this article and providing useful comments on it.

REFERENCES

AEKATERINIDIS, I. AND TRIANTAFILLOU, P. 2005. Internet scale string attribute publish/subscribe data networks. In *Proceedings of the ACM 14th Conference on Information and Knowledge Management (CIKM05)*.

AGUILERA, M. K., STROM, R. E., STURMAN, D., ASTLEY, M., AND CHANDRA, T. 1999. Matching events in a content-based subscription system. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*. ACM, New York, 53–62.

AHO, A., HOPCROFT, J., AND ULLMAN, J. 1983. *Data Structures and Algorithms*. Addison- Wesley, Reading, MA.

AHO, A. V., SETHI, R., AND ULLMAN, J. 1986. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

ALTINEL, M., AKSOY, D., BABY, T., FRANKLIN, M., SHAPIRO, W., AND ZDONIK, S. 1999. DBIS-toolkit: Adaptable middleware for large-scale data delivery. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

ALTINEL, M. AND FRANKLIN, M. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th VLDB Conference*.

AMER-YAHIA, S., BOTEV, C., AND SHANMUGASUNDARAM, J. 2004. TeXQuery: A full-text search extension to Query. In *Proceedings of WWW*. ACM Press, 583–594.

AOE, J.-I., MORIMOTO, K., AND SATO, T. 1992. An efficient implementation of trie structures. *Softw.—Pract. Exper. 22,* 9, 695–721.

BAEZA-YATES, R. AND GONNET, G. 1996. Fast text searching for regular expressions or automaton simulation on tries. *J. ACM 43,* 6, 915–936.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison Wesley, Reading, MA.

BELKIN, N. AND CROFT, W. 1992. Information filtering and information retrieval: Two sides of the same coin? *Comm. ACM 35,* 12, 29–38.

BELL, T., CLEARY, J., AND WITTEN, I. 1990. *Text Compression*. Prentice-Hall publishers.

BELL, T. AND MOFFAT, A. 1996. The design of a high performance information filtering system. In *Proceedings of the ACM SIGIR*. 12–20.

BHARAMBE, A., AGRAWAL, M., AND SESHAN, S. 2004. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM*. Portland, Oregon, USA.

BHARAMBE, A., RAO, S., AND SESHAN, S. 2002. Mercury: A scalable publish-subscribe system for Internet games. In *Proceedings of the 1st International Workshop on Network and System Support for Games (Netgames)*. Braunchweig, Germany.

CALLAN, J. 1996. Document filtering with inference networks. In *Proceedings of the ACM SIGIR*.

CALLAN, J. 1998. Learning while filtering focuments. In *Proceedings of the ACM SIGIR*. 224–231.

CALLAN, J., CROFT, W., AND HARDING, S. 1992. The INQUERY retrieval system. In *Proceedings of the 3rd International Conference on Database and Expert Systems Applications*. Springer-Verlag, 78–83.

CAMPAILLA, A., CHAKI, S., CLARKE, E., JHA, S., AND VEITH, H. 2001. Efficient filtering in publish subscribe systems using binary decision diagrams. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*. IEEE Computer Society, 443–452.

CARZANIGA, A., ROSENBLUM, D.-S., AND WOLF, A. 2001. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst. 19,* 3, 332–383.

CARZANIGA, A., ROSENBLUM, D. S., AND WOLF, A. L. 2000. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*. 219–227.

CHAN, C.-Y., FELBER, P., GAROFALAKIS, M., AND RASTOGI, R. 2002. Efficient filtering of XML documents with XPath expressions. In *Proceedings of ICDE*. 235–244.

CHANG, C.-C. 2001. Query and data mapping across heterogeneous information sources. Ph.D. thesis, Stanford University.

CHANG, C.-C., GARCIA-MOLINA, H., AND PAEPCKE, A. 1996. Boolean query mapping across heterogeneous information sources. *IEEE Trans. Knowl. Data Eng. 8,* 4, 515–521.

CHANG, C.-C. K., GARCIA-MOLINA, H., AND PAEPCKE, A. 1999. Predicate rewriting for translating Boolean queries in a heterogeneous information system. *ACM Trans. Inform. Syst. 17,* 1, 1–39.

CHINENYANGA, T. T. AND KUSHMERICK, N. 2001. Expressive retrieval from XML documents. In *Proceedings of SIGIR'01*.

COHEN, W. W. 2000. WHIRL: A word-based information representation language. *Artif. Intell. 118,* 1-2, 163–196.

COMER, D. 1981. Analysis of a heuristic for trie minimization. *ACM Trans. Datab. Syst. 6,* 3, 513–537.

COMER, D. AND SETHI, R. 1977. The complexity of trie index construction. *J. ACM 24,* 3, 428–440.

CRESPO, A. AND GARCIA-MOLINA, H. 2002. Routing indices for peer-to-peer systems. In *ICDCS*.

DE LA BRIANDAIS, R. 1959. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*. 295–298.

DENNING, P. 1982. Electronic junk. *Comm. ACM 25,* 3, 163–165.

DEVROYE, L. 1992. A study of trie-like structures under the density model. *Annals Appl. Prob. 2,* 2, 402–434.

DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. 1984. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 92–95.

DIAO, Y., ALTINEL, M., FRANKLIN, M., ZHANG, H., AND FISCHER, P. 2003. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Datab. Syst.*

DONG, L. 2002. Automatic term extraction and similarity assessment in a domain specific document corpus. M.S. thesis, Department of Computer Science, Dalhousie University, Halifax, Canada.

FABRET, F., JACOBSEN, H. A., LLIRBAT, F., PEREIRA, J., ROSS, K. A., AND SHASHA, D. 2001. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of ACM SIGMOD*.

FLAJOLET, P. 1983. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica 20*, 345–369.

FLAJOLET, P. AND PUECH, C. 1986. Partial match retrieval of multidimensional data. *J. ACM 33,* 2, 371–407.

FOLTZ, P. AND DUMAIS, S. 1992. Personalized information delivery: An analysis of information filtering methods. *Comm. ACM 35,* 12, 51–60.

FRANKLIN, M. AND ZDONIK, S. 1998. "Data in Your Face": Push technology in perspective. *SIGMOD Record (ACM Special Interest Group on Management of Data) 27,* 2, 516–519.

FRANTZI, K., ANANIADOU, S., AND MIMA, H. 2000. Automatic recognition of multiword terms:the c-value/nc-value method. *JODL 5,* 2.

FREDKIN, E. 1960. Trie memory. *Comm. ACM 3,* 9, 490–499.

FUHR, N. AND GROSJOHANN, K. 2004. XIRQL: An XML query language based on information retrieval concepts. *ACM Trans. Inform. Syst. 22,* 2, 313–356.

GARCIA-MOLINA, H. AND SALEM, K. 1992. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng. 4,* 6, 509.

GEDIK, B. AND LIU, L. 2003. PeerCQ: A decentralized and self-configuring peer-to-peer information monitoring system. In *Proceedings of the the 23rd International Conference on Distributed Computing Systems*.

GREEN, T. J., MIKLAU, G., ONIZUKA, M., AND SUCIU, D. 2003. Processing XML streams with deterministic automata. In *Proceedings of the International Conference on Database Technology*. 173–189.

GUPTA, A., SAHIN, O. D., AGRAWAL, D., AND ABBADI, A. E. 2004. Meghdoot: Content-based publish/subscribe over P2P networks. In *Proceedings of ACM/IFIP/USENIX 5th International Middleware Conference*.

HULL, D., PEDERSEN, J., AND SCHÜTZE, H. 1996. Method combination for document filtering. In *Proceedings of the ACM SIGIR*. 279–287.

IDREOS, S., KOUBARAKIS, M., AND TRYFONOPOULOS, C. 2004a. P2P-DIET: An extensible P2P service that unifies ad-hoc and continuous querying in super-peer networks. In *Proceedings of the ACM SIGMOD Conference*. 933–934.

IDREOS, S., KOUBARAKIS, M., AND TRYFONOPOULOS, C. 2004b. P2P-DIET: One-time and continuous queries in super-peer networks. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT)*. 851–853.

JACQUET, P. AND SZPANKOWSKI, W. 1991. Analysis of digital tries with Markovian dependency. *IEEE Trans. Inform. Theor. 37,* 5, 1470–1475.

KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*. 654–663.

KNUTH, D. 1973a. *The Art of Computer Programming*. Vol. 3: Sorting and Searching. Addison-Wesley, Reading, MA.

KNUTH, D. 1973b. *The Art of Computer Programming*. Vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, MA.

KOUBARAKIS, M., KOUTRIS, T., TRYFONOPOULOS, C., AND RAFTOPOULOU, P. 2002. Information alert in distributed digital libraries: The models, languages, and architecture of DIAS. In *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*. 527–542.

KOUBARAKIS, M., SKIADOPOULOS, S., AND TRYFONOPOULOS, C. 2006. Logic and computational complexity for Boolean information retrieval. *IEEE Trans. Knowl. Data Eng. 18,* 12, 1659–1666.

KOUBARAKIS, M., TRYFONOPOULOS, C., IDREOS, S., AND DROUGAS, Y. 2003. Selective information dissemination in P2P networks: Problems and solutions. *SIGMOD Record, Special Issue on Peer-to-Peer Data Management 32,* 3, 71–76.

KOUBARAKIS, M., TRYFONOPOULOS, C., RAFTOPOULOU, P., AND KOUTRIS, T. 2002. Data models and languages for agent-based textual information dissemination. In *Proceedings of the 6th International Workshop on Cooperative Information Agents (CIA)*. Lecture Notes in Artificial Intelligence, vol. 2446. Springer, 179–193.

LUHN, H. 1958. A business intelligence system. *IBM J. Reasear. Devel. 2,* 4, 314–319.

MILIOS, E., ZHANG, Y., HE, B., AND DONG, L. 2003. Automatic term extraction and document similarity in special text corpora. In *Proceedings of the 6th Conference of the Pacific Association for Computational Linguistics (PACLing)*. 275–284.

MORITA, M. AND SHINODA, Y. 1994. Information filtering based on user behaviour analysis and best match text retrieval. In *Proceedings of the ACM SIGIR*. 272–281.

NAVARRO, G. AND BAEZA-YATES, R. 1997. Proximal nodes: A model to query document databases by content and structure. *ACM Trans. Inform. Syst. 15,* 4, 400–435.

NGUYEN, B., ABITEBOUL, S., G.COBENA, AND PREDA, M. 2001. Monitoring XML data on the Web. In *Proceedings of the ACM SIGMOD Conference*. Santa Barbara, CA, USA.

NILSSON, S. AND KARLSSON, G. 1999. IP-address lookup using LC-tries. *IEEE J. Select. Areas Comm. 17,* 6, 1083–1092.

PETERSON, J. 1980. Computer programs for detecting and correcting spelling errors. *Comm. ACM 23,* 12, 676–686.

PFEIFER, U., FUHR, N., AND HUYNH, T. 1995. Searching structured documents with the enhanced retrieval functionality of freeWAIS-sf and SFgate. *Comput. Netw. ISDN Syst. 27,* 6, 1027–1036.

PIETZUCH, P. AND BACON, J. 2002. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*.

RAFTOPOULOU, P., PETRAKIS, E. G., TRYFONOPOULOS, C., AND WEIKUM, G. 2008. Information retrieval and filtering over self-organising digital libraries. In *Proceedings of the 12th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. 2001. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM Conference*.

REGNIER, M. AND JACQUET, P. 1989. New results on the size of tries. *IEEE Trans. Inform. Theor. 35,* 1, 203–205.

RIVEST, R. L. 1976. Partial-match retrieval algorithms. *SIAM J. Comput. 5,* 1, 19–50.

ROWSTRON, A. AND DRUSCHEL, P. 2001. Pastry: Scalable, distributed object location and routing for large-scale- peer-to-peer storage utility. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Paltforms (Middleware'01)*.

ROWSTRON, A., KERMARREC, A.-M., CASTRO, M., AND DRUSCHEL, P. 2001. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the 3rd International COST264 Workshop*, J. Crowcroft and M. Hofmann, Eds.

SEVERANCE, C. AND PRAMANIK, S. 1990. Distributed linear hashing for main memory databases. In *Proceedings of the International Conference on Parallel Processing*. 92–95.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM Conference*.

SUSSENGUTH, E. 1963. Use of tree structures for processing files. *Comm. ACM 6,* 5, 272–279.

TAM, D., AZIMI, R., AND JACOBSEN, H.-A. 2003. Building content-based publish/subscribe systems with distributed hash tables. In *Proceedings of the 1st International Workshop On Databases, Information Systems and Peer-to-Peer Computing*.

TANG, C. AND XU, Z. 2003. pFilter: Global information filtering and dissemination using structured overlays. In *FTDCS*.

TERPSTRA, W., BEHNEL, S., FIEGE, L., ZEIDLER, A., AND BUCHMANN, A. 2003. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*.

THEOBALD, A. AND WEIKUM, G. 2000. Adding relevance to XML. In *WebDB (Selected Papers)*. 105–124.

THEOBALD, M., SCHENKEL, R., AND WEIKUM, G. 2005. An efficient and versatile query engine for TopX search. In *Proceedings of the 31st International Conference on Very Large Databases (VLDB)*.

TRYFONOPOULOS, C., IDREOS, S., AND KOUBARAKIS, M. 2005a. LibraRing: An architecture for distributed digital libraries based on DHTs. In *Proceedings of the 9th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*. 25–36.

TRYFONOPOULOS, C., IDREOS, S., AND KOUBARAKIS, M. 2005b. Publish/subscribe functionality in IR environments using structured overlay networks. In *Proceedings of the 28th Annual International ACM SIGIR Conference*. 322–329.

TRYFONOPOULOS, C. AND KOUBARAKIS, M. 2002. Selective dissemination of information in P2P systems: Data models, query languages, algorithms and computational complexity. Tech. Rep. TR-ISL-02-2003, Department of Electronic and Computer Engineering, Technical University of Crete.

TRYFONOPOULOS, C., KOUBARAKIS, M., AND DROUGAS, Y. 2004. Filtering algorithms for information retrieval models with named attributes and proximity operators. In *Proceedings of the 27th Annual International ACM SIGIR Conference*. 313–320.

TRYFONOPOULOS, C., ZIMMER, C., KOUBARAKIS, M., AND WEIKUM, G. 2007. Architectural alternatives for information filtering in structured overlay networks. *IEEE Intern. Comput. 11,* 4, 24–34.

YAN, T. AND GARCIA-MOLINA, H. 1994a. Index structures for information filtering under the vector space model. *Proceedings of the 10th International Conference on Data Engineering*, 337–347.

YAN, T. AND GARCIA-MOLINA, H. 1994b. Index structures for selective dissemination of information under the Boolean model. *ACM Trans. Datab. Syst. 19,* 2, 332–364.

YAN, T. AND GARCIA-MOLINA, H. 1999. The SIFT information dissemination system. *ACM Trans. Datab. Syst*.

YANG, B. AND GARCIA-MOLINA, H. 2003. Designing a super-peer network. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*.

YOCHUM, J. A. 1985. A high-speed text scanning algorithm utilising least frequent trigraphs. In *Proceedings of the IEEE Symposium on New Directions in Computing*.

ZHANG, Y. AND CALLAN, J. 2001. Maximum likelihood estimation for filtering thresholds. In *Proceedings of the ACM SIGIR*.

ZIMMER, C., TRYFONOPOULOS, C., BERBERICH, K., KOUBARAKIS, M., AND WEIKUM, G. 2008. Approximate information filtering in peer-to-peer networks. In *Proceedings of the 9th Web Information Systems Engineering (WISE) Conference*.

ZIMMER, C., TRYFONOPOULOS, C., AND WEIKUM, G. 2007. MinervaDL: An architecture for information retrieval and filtering in distributed digital libraries. In *Proceedings of the 11th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*. 148–160.

ZIMMER, C., TRYFONOPOULOS, C., AND WEIKUM, G. 2008. Exploiting correlated keywords to improve approximate information filtering. In *Proceedings of the 31st Annual International ACM SIGIR Conference*.

Received February 2006; revised July 2007; accepted June 2008