University of the Peloponnese
Department of Informatics and Telecommunications
Th. Kolokotronis Campus, 22100, Tripoli

Parallel Algorithms for Efficient Information Filtering

Giorgos Parras

M.Sc. Thesis

Tripoli, November 2020

Parallel Algorithms for Efficient Information Filtering.

Giorgos Parras.

M.Sc. Thesis.

Software and Database Systems Lab.

Department of Informatics and Telecommunications, University of the Peloponnese

University of the Peloponnese
Department of Informatics and Telecommunications
Ac. Vlachou, 22100, Tripoli
Tel. 2710-230177
Fax. 2710-372160

Parallel Algorithms for Efficient Information Filtering

Giorgos Parras

M.Sc. Thesis

Supervisor:

Spiros Skiadopoulos, Professor, University of the Peloponnese.

Examination committee:

Costas Vassilakis, Professor, University of the Peloponnese.

Christos Tryfonopoulos, Associate Professor, University of the Peloponnese.

Acknowledgements

The completion of this work closes the circle of my postgraduate studies and also marks the completion of my research on a part of the Information Filtering field. This research would not be possible to complete without the support of my supervisor Prof. *Spiros Skiadopoulos*, who guided me in this effort. I would also like to extend my sincerest thanks to Prof. *Costas Vassilakis*, Assoc. Prof. *Christos Tryfonopoulos* and Dr. *Lefteris Zervakis* for their valuable advice and comments, together with my supervisor, in writing this thesis and valuable corrections to achieve these comprehensive results. The knowledge and the help they gave me over information dissemination and filtering were instrumental in the completion of this project.

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

Περίληψη

Σε ένα σύστημα φιλτραρίσματος πληροφορίας, οι χρήστες μπορούν να εγγράφονται σε ένα διακομιστή, δημιουργώντας συνεχή ερωτήματα που εκφράζουν τις ανάγκες τους. Τα ερωτήματα αυτά στοχεύουν στην ανάκτηση σχετικών κειμένων που έχουν δημοσιευθεί στους διακομιστές. Πιο συγκεκριμένα, όταν νεα κείμενα δημοσιεύονται τα ερωτήματα που ικανοποιούν αυτό το έγγραφο εντοπίζονται, ενημερώνοντας κατάλληλα τους χρήστες.

Δοσμένης μιας βάσης δεδομένων db και ενός εισερχόμενου κειμένου d το σύστημα φιτραρίσματος πληροφορίας βρίσκει όλα τα ερωτήματα $q \in db$ τα οποία ταιριάζουν με το d. Στην παρούσα εργασία επικεντρωνόμαστε σε ερωτήματα που εκφράζονται στο μοντέλο $\mathcal{AWP}$. Το μοντέλο δεδομένων $\mathcal{AWP}$ βασίζεται σε γνωρίσματα που περιέχουν κείμενο και η γλώσσα ερωτήσεων περιλαμβάνει λογικούς τελεστές και τελεστές εγγύτητας.

Επίσης, εξετάζουμε την αποτελεσματικότητα των τεχνικών παραλληλοποίησης των διαδικασιών φιλτραρίσματος της πληροφοριας. Για το σκοπό αυτό χρησιμοποιούμε κατάλληλες δομές δεδομένων, μεθόδους δεικτοδότησης και τεχνικές παραλληλοποίησης. Χρησιμοποιώντας τους προαναφερθέντες μηχανισμούς, οι μέθοδοι παραλληλοποίησης επιτυγχάνουν βελτίωση μεγαλύτερη του 98% στην απόδοση του φιλτραρίσματος για μεγάλες βάσεις δεδομένων (εως 3 εκατομμύρια ερωτήματα) όπου οι ερωτήσεις των χρηστών εκφράζονται στο μοντέλο $\mathcal{AWP}$.

Abstract

In the information filtering paradigm, clients subscribe to a server with continuous queries that express their information needs. Such queries aim to retrieve relative documents that are published on the server. More specifically, whenever a new document is published on the server, the continuous queries satisfying this document are found and notifications are sent to the respective clients.

More formally, given a database of continuous queries db and an incoming document d, an information filtering process finds all queries $q \in db$ that match d. We concentrate on queries that are expressed in the $\mathcal{AWP}$ data model. This model is based on named attributes with values of type text, and its query language includes Boolean and word proximity operators.

In this thesis, we consider the efficient parallelization of the information filtering procedures. To this end, we employ appropriate data structures, indexing methods and parallel techniques. Using the aforementioned machinery, our parallel methods achieve an improvement of more than 98% in filtering performance for large databases (up to 3 million queries), expressed in the $\mathcal{AWP}$ model.

Contents

List of figures

List of tables

List of abbreviations

AD	model predictive control
AWP	Attributes with Word Patterns
$AWPS$	Attributes with Word Patterns Similarity
BFT	BestFitTrie
CPU	Central Processing Unit
CA	Clustering Array
DAL	Distinct Attribute List
DWL	Distinct Word List
IF	Information Filtering
IR	Information Retrieval
LSI	Latent Semantic Indexing
OT	Occurrence Table
$P2P$	Peer-to-Peer
PT	PrefixTrie
RT	ReTrie
SEO	Search Engine Optimization

SDI	Selective Dissemination of Information
$StaR$	Statistical Reorganisation
WP	Word Pattern
WD	Word Directory
XML	eXtensible Markup Language

Chapter 1

Introduction

This thesis addresses the efficient parallelization of information filtering in publish/subscribe systems. In this chapter, we define the problem, highlight our approach and present our contributions.

1.1 Problem definition

Lately, we have witnessed an exponential growth of the internet and of the amount of information created, stored and requested on a daily basis. Users are often overwhelmed by the vast amount of existing information and call for information filtering mechanisms that will allow them to focus only on the relevant information.

Search engines (like Google, Yahoo and Bing) are the main applications used for retrieving information in the internet. They are based on information retrieval techniques and they are the most common example of the respective field. Although being very popular and helpful, they do not completely address the problem of relevant information finding. By providing a query input field, the user manually starts a search (usually using small, general keywords) without knowing whether relevant information exists or not and without being aware of the possible available and useful information due to the nature of the query structure, e.g., wrong or different query syntax can lead to loss of useful information.

After submitting the query, depending on its definition, search engines may often

provide huge results that cannot be managed effectively by the user. Also, black-hat SEO techniques (disapproved practices that could increase a page's ranking in a search engine result page, e.g., link manipulation, link farms, keyword stuffing) [52], can affect the search results order and eventually mislead the user.

More importantly, search engines are not able to inform the user of new available information. Additionally, the vast flow of newly created information significantly reduces the time frame that a query result remains valid. This means that a query result may be outdated within minutes or even seconds after its execution (since new information may be added or existing may be modified of even be deleted).

This creates the need of a system able to inform users when something that interests them becomes available. Information filtering, also referred to as *publish/subscribe* or *continuous querying* or *information push*, solves this problem by dynamically notifying the user, at the time new information becomes available. This key feature makes information filtering equally important to information retrieval and search engines.

Information filtering system users can express their information interests by subscribing to a server with *continuous queries* or *profiles*. These queries are specified using a well-defined language tailored to express the information needs of users. In an information filtering scenario, a user posts a query to the system in order to receive notifications when certain events of interest occur, e.g., a document, matching the stored *user query*, is added to the system. Whenever a new information reaches the system, it is validated with indexed queries and the matching users are instantly notified.

Existing indexing and filtering algorithms are sequential. The goal of this thesis is to adopt and extend those algorithms in order to take advantage of the multi-core capabilities of modern processors and to effectively solve the problem of *filtering parallelization* in each individual server. Based on the above, the problem of information filtering examined in this thesis may be defined as follows:

> *Given a database of continuous queries db and a document d, find all queries* $q \in db$ *that match d, using parallelization techniques.*

1.2 Solution outline

This thesis is focusing on the algorithms and the data structures used for indexing user queries in publish/subscribe systems and on the acceleration of the filtering process by taking advantage of the use of parallelization techniques.

The process of filtering causes the most work in the server, therefore it has to be efficient in order to handle millions of user queries in real-time. We focus on the parallelization of the matching process (information filtering), with all the information being in text format and using a data model that is quite common in information retrieval, called $\mathcal{AWP}$ (Attributes with Word Patterns) [44].

This model is based on attributes with text values and the query language supports attributes, along with comparison operators (like "equals" and "contains") and word proximity operators from the Boolean model of information retrieval [18].

The algorithms are based on the prior work of Tryfonopoulos et al. [70, 71] and on the work of Zervakis et al. [80]. The basic idea behind the algorithms that we will present, is the use of *tries* in order to index the queries and to achieve the best clustering possible. Better clustering is going to help us during the filtering process in order to minimize the number of visits to the forest nodes and thus taking less time to compute the answer.

In our work, we concentrate on extending the presented algorithm implementation by parallelizing the indexing and filtering process to suit modern multi-core processors. Such an improvement is critical as filtering algorithms are expected to process high volumes of incoming information as efficiently as possible.

1.3 Contributions

A major part of the work conducted in the field of publish/subscribe systems. The algorithms used are based on the work of Garcia-Molina [74] as well as on the extensions of the algorithms presented by Tryfonopoulos et al. [70, 71]. Specifically, we will extend the use of the PrefixTrie algorithm, that is an extension of the *Tree* of Yan and Garcia-Molina [74] for attributes.

Also, we extend the use of the algorithms BestFitTrie and ReTrie as they are presented in Tryfonopoulos et al. [71], which is an approach for better organization and reorganization of the forest respectively, both targeting on better query grouping. Finally, we extend the use of the StaR algorithm, the first algorithm in literature to consider database reorganization through appropriate word/query statistics for efficient filtering.

We propose methods to improve the performance of the filtering procedure and we concentrate on solving the parallelization problem efficiently.
Finally, we present measurements of the times of each algorithms using single core and multi-core examples.

In the light of the above, our contributions are:

- We investigate the effectiveness of the parallel techniques in both indexing and filtering functions.
- We extend existing algorithms implementation from single-core to multi-core environments while exploiting full CPU capacity, based on the machine's characteristics. We identify one parallelization option and experimentally evaluate its performance.

1.4 Thesis structure

The rest of the work is organized as follows. Chapter 2 surveys related work in the fields of information filtering on single-core and multi-core environments and techniques on query indexing and publication filtering.

Chapter 3 presents the data model and algorithms developed to solve the problem of indexing user queries, describing the indexing Algorithms (BestFitTrie, PrefixTrie, ReTrie, StaR) and it's variants, developed to solve the indexing and filtering problem.

Chapter 4 gives the experimental evaluation of the developed parallel versions of the algorithms using real-world data sets and comparing them against their sequential/serial version.

Finally, Chapter 5 gives conclusions and future directions for our work.

Chapter 2

Related work

In this chapter, we present related work in information filtering systems and parallelization techniques. At first, we briefly provide a background discussion on tries and on information filtering. Subsequently, we present works that focus on single-core an multi-core environments and techniques. Finally, we present previous works on multi-threading techniques in the information filtering field.

2.1 Tries

The concept of *tries* was originally developed in the work of R. de la Briandais [25], but the actual term *tries* was coined by Frednik [36] and was derived from the word *retrieval*. Tries are used in a wide range of applications including, dictionary management [2, 7, 42], text compression [10], natural language processing [8, 63], pattern recognition [33, 68], IP routing [61], or even searching for reserved words in a compiler [3]. The range of applications over which they can be applied, rank them as data structures of general purpose with properties that have become known through various surveys [27, 32, 41, 42, 67, 68].

There are several ways on how to implement the trie nodes and the choice depends on the type of application that use them. However, two are the most common ways to implement tries. The first is by using arrays on the size of the alphabet (array tries) [36] and the second is by using linked lists with non-empty elements as roots of subtrees

[25, 43].

The tries based on arrays are best suited to a relatively small alphabet. Lists are more suited for larger alphabet sizes or nodes that have a small number of children (in such case the array representation would consist of many null pointers). There are two main ways to reduce the size of a trie, either by reducing the size of the nodes or by reducing the number of nodes needed to represent sets of words.

Compact tries [69] are variations that aim at reducing the total number of nodes used to represent a set of words. This is achieved by compressing sequences of nodes that lead to a leaf, thus, reducing the degree of branching in a single node. Another concept for reducing the size of the trie is to address the indexing of words as a whole and not as a sequence. This way the size of nodes can be influenced, resulting in a smaller forest. Nevertheless, Comer and Sethi [23] showed that the problem of finding the smallest possible tree is *NP-complete* and therefore several heuristic methods for reducing the size of a static tree have been proposed. Such an example can be found in the work of Comer [22].

In our approach, we use the concept of tries based on linked lists to implement data structures which are going to store the user queries. The algorithms we will present are using tries based on lists and on techniques used by compact tries, in order to be able to identify the common elements of the queries and thus to make better node grouping.

2.2 Information filtering

Information retrieval and *information filtering* are often referred as two sides of the same coin [9]. 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), the techniques and the algorithms devised to satisfy these information needs, differ significantly.

Information filtering, also known as *publish/subscribe*, *continuous querying*, or *information push*, is equally important to one-time querying, since users are able to

subscribe to information sources and be notified when documents of interest are published. In an information filtering scenario, a user posts a subscription (or continuous query) to the system to receive notifications whenever certain events of interest take place (e.g., when a paper on distributed systems becomes available).

One of the early works done on the field of selective dissemination of information, is the work of Luhn [51], which describes an "Business Intelligence System". In this system, the user describes his interests, creating a profile, and then the text selection system offers lists of new texts. From this list, the users have the ability to choose and order the texts that interest them. At that time, this process of selection of texts presented, was described as *selective dissemination of new information*. The term *information filtering* was coined later by Denning [26], where it was described the need for the existence of a system which will filter incoming e-mails in order of significance.

In their early approaches, scientists focused on the correct representation of the interests of the user [55] and on the optimization of the filtering process [39]. The work of Morita and Shinoda [55] focused on the use of techniques that observe the behavior of users while indexing subsets of terms in order to determine which texts interest a real user. The work of Hull [39] studied the filtering, using methods based on machine learning techniques, where combinations of strategies were used to increase the accuracy of filtering (and of the results delivered to the user). Other techniques include the use of statistical data for filtering incoming documents such as LSI-SDI [34] that makes use of the LSI method in order to filter incoming documents. Latent Semantic Indexing (LSI) is an extension of the standard vector retrieval method where associations among terms and texts are calculated and exploited in retrieval. The assumption is that there is some underlying or "latent" structure in the pattern of word usage across documents and that statistical techniques can be used to estimate this latent structure. A description of terms, documents, and user queries based on the underlying latent semantic structure is used for representing and retrieving information.

One of the first works that studied the issue of *performance factor* in information filtering systems is the work of Bell and Moffat [11], which describes an information filtering system, capable of filtering large volumes of information of data. The researchers

developed a server which was receiving incoming texts with great pace, and proposed algorithms supporting the vector space model of information retrieval by improving the SQI algorithm of Yang and Garcia-Molina [73].

Another influential system is InRoute [13]. InRoute is based on inference networks and focuses on filtering efficiency. The InRoute system creates documents and query networks and uses belief propagation techniques to filter incoming documents.

Other works in the area focused on adaptive filtering [14, 82] and how queries, that were represented in the vector space model, have adjusted according to the documents that have been filtered in the past.

Besides the statistical approaches described above, filtering systems based on the Boolean model of information retrieval have also been developed. A representative example is LMDS [77], which uses least frequent trigrams to allow faster processing of incoming documents. LMDS indexes queries of users under the trigram with the smaller number of appearances. The documents are also represented as a sequence of trigrams. During the process of filtering, a table determines which queries match the incoming document and since false positive results may incur, a second stage is required to determine the actual matches.

Later on, similar works focused on information filtering that support data models with attributes and query languages that implement arithmetic and string comparison operators (e.g., Le Subscribe [30] and the monitoring subsystem of Xyleme [60]). Another system worth mentioning is the system presented by Campailla et al. [15], because it uses queries based on attributes, but goes beyond conjunctive queries which is the standard class of queries that other systems use [30]. Most recent works focus on supporting documents in XML format and queries that are subsets of XQuery or XPath (e.g., XFilter [6], YFilter [28], Xtrie [17] and *xmltk* [37].

One of the main issues in publish/subscribe systems is to determine the interests of the user and the best possible way to represent them. Thus, we find approaches such as those of Nanas et al. [56, 57] for creating profiles representing users. By using genetic and machine learning algorithms, profiles are created close to user's interests. Additionally, the user rates the final information delivered to him, indirectly indicating

which queries will be used for the next generation of queries.

Chang et al. [19] relying on the fact that a user's interests change over time, developed a personalized filtering system focusing on the ability to easily reflect these changes. Using a data mining system, based on the Apriori algorithm [1], they developed the *updatable tree* which is an index structure that supports incremental updates. They focused on recognizing both long-term and short-term interests of each user by taking weights of keywords in queries into consideration. Categorizing the users' interests aimed at a better query indexing and made it easier to reorganize the system when changes occur regularly. Other approaches stir away from the classical model of query representation as a sequence of simple terms and focus on relationships between words [56]. In these approaches, a network of word nodes with weights is created, to better represent the interests of each user and to increase the accuracy of the results returned to him. As part of this effort, for better and comprehensive representation of information, methods were developed for the description of the user's interests based on ontologies [81]. Park and Chung [62] also developed a broker that accepts queries in SPARQL, a language for expressing queries in a form suitable to be filtered by documents represented by ontologies. An extension of SPARQL with full-text operators was proposed by Zervakis et al. [79], aiming at more expressive continuous queries that are able to support versatile user needs.

Other works [48, 83, 84] focus more on the problem of the document filtering and divide it in two stages in order to accelerate the process and to provide greater accuracy in the results brought to users. In the first stage irrelevant documents are excluded from entering the system while in the second stage a filtering technique, based on a pattern matching method, is applied to the remaining texts (which is a relatively a smaller number). Their aim is to achieve greater precision in identifying those that match the queries and reduce the number of non-relevant documents that arrive to the final user. Following the assumption that word patterns offer more information than a model based on a set of words [53] Nanas and Vavalis [58] tried to identify semantic similarities in documents and user queries. In the same context, other researches were also initiated [4, 46, 47] receiving more into account the negative user response to documents while

combining this response with terms pattern recognition in order to effectively reduce the noise that reaches users.

2.2.1 Information filtering in databases

The majority of the work on information filtering regarding the database literature, has its origins on the work of Franklin and Zdonik [35] where the term *selective dissemination of information* (SDI) is also used. Their previous work also appears in the DBIS system [5]. The term *publish/subscribe systems*, that comes from the area of distributed systems, is also used by database researchers. Another system with great influence is SIFT [74, 75], where publications are simple documents and queries are sets of words. The SIFT system was the first that emphasized on the indexing of the queries/profiles in order to cope with a large amount of document arrival rate [74].

A more recent fully-functional, content-based information filtering system is PING, as proposed by Chantzios et. al. [20]. PING aims to showcase the realisability of information filtering and to explore and test the suitability of the existing technological arsenal for information filtering tasks. The system is entirely based upon open-source tools and it is customisable enough to be adapted for different textual information filtering tasks. PING puts emphasis in user profile expressivity, intuitive UIs, and timely information delivery.

2.2.2 Information filtering in distributed systems

We will mention some of the most representative information filtering systems developed in recent years. Driven by the lack of large-scale systems Carzaniga et al. [16] developed SIENA, a distributed system that provides user-notification services that are based on events. The SIENA system makes use of a data model based on attribute-value pairs, that offers the opportunity to express notifications, subscriptions and advertisements.

This is the first distributed system which focuses both at providing expressiveness in the way users request the information and to offer solutions related to a wide range

of users in a wide space such as the Internet. Building on top of SIENA's main idea, Koubarakis et al. [44] developed the DIAS system as well as the P2P-DIET system [40, 45] and used $\mathcal{AWP}$ and $\mathcal{AWPS}$ data models whereas the creation of which was based on ideas of information retrieval. These two systems attempt to combine ideas from information retrieval and databases in order to provide them in a single framework, while maintaining the characteristics of SIENA relating to the information filtering side.

Another important contribution of P2P-DIET is that it demonstrates how to support ad-hoc or one-time query scenarios, while supporting two similar protocols, that of super-peer systems [76] and the pub/sub features (SIENA) [16]. Also, the iClusterDL system [65] demonstrates how to use an unstructured network, called *Semantic Overlay Network* [24], to support functions of both information retrieval and information filtering in an electronic library.

2.3 Information filtering parallel techniques

As the trend of representing information in XML rises, algorithms are developed to support documents and queries in this form ([21, 38, 50]), while Grummt [38] emphasizes at the parallel query and document processing.

An interesting approach to document filtering is also given by Vanderbauwhede et al. [72] as they implement the procedure of filtering, using Field Programmable Gate Arrays (FPGA) in order to accelerate the most demanding routines. Similarly, other work [31, 64] try to exploit the power of modern processors by using *threads* [49, 54] and by parallelizing the algorithms that are responsible for the filtering.

The work of Farroukh et al. [31], showed a 74% reduction of the average matching time when using eight processors. In the work of Qian et al. [64] experiments showed that relatively independent matching algorithm produced similar throughput as complete independent matching algorithm and the average matching time of collaborative matching algorithm was reduced by approximately 77% as the number of threads increases from one to six.

Chapter 3

Data model and algorithms

In this chapter, we present the $\mathcal{AWP}$ data model (Section 3.1) we used to define both the user queries and the documents. We also present the algorithms, which make use of the data model, for indexing the queries and matching the incoming documents. We follow the presentation of [44, 71, 78], on top of which this thesis was based on.

3.1 The AWP model

$\mathcal{AWP}$ is a data model widely used in the area of information filtering. It can be used to represent and query textual information under the Boolean model, using attributes with values of type text, witch are used to encode textual information in a notification (e.g., author and title).

The query language of the $\mathcal{AWP}$ model, supports Boolean and word proximity operators, as well as comparison operators *equals* and *contains*, just as the model of Chang et al. [18] does.

3.1.1 Definition of documents

Documents in $\mathcal{AWP}$ consist of *attributes* with values of plain *text*. Let Σ be a finite alphabet. A *word* is a finite non-empty sequence of letters belonging to Σ. As a *vocabulary* we define a set of finite number of words denoted by $\mathcal{V}$.

In the examples of this thesis, the dictionary $\mathcal{V}$ represents the dictionary of the English language.

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

Essentially, s consists of a finite sequence of words belonging to $\mathcal{V}$ and the expression $s(i)$ gives us the i-th word of s. The value s can be used to represent finite-length strings of words separated by blanks. In order to refer to the length of a text value s, we will use the notation $|s|$.

Example 1 *The following sequence of words "will study the data structures", is represented by a text value s of length $|s| = 5$ and $s(1) = will$, $s(2) = study$ etc.*

In the rest of this section, we are going to use the s symbol when referring to text values while our definitions will abide by the mathematical definitions quoted above. Next we define $\mathcal{A}$ as a set of attributes which we will call *attribute universe*.

Practically, the attributes are used to organize information into fields. Thus, for example, attributes of a document are the author's name, the title, the summary, the body text etc. The attributes may derive from a collection of values (*namespaces*), for example from the set Dublin Core Metadata Element[1].

Definition 2 *A document d consists of a set of attribute-value pairs (A, s) where $A \in \mathcal{A}$, s is a text value and all attributes are unique and distinct.*

From now on, when we use the notation $A(d)$ we will refer to a unique text value s of document d, such that $(A, s) \in d$.

Example 2 *Consider the following set of pairs in a document:*

$$d = \{(AUTHOR, \text{“Hector Garcia-Molina”}),$$
$$(TITLE, \text{“Selective dissemination of information in databases”}),$$
$$(ABSTRACT, \text{“In this paper we will study the most widespread ...”})\}$$

Based on the above, we have $AUTHOR(d) =$ “Hector Garcia-Molina”

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

3.1.2 Definition of queries

In order to define queries in $\mathcal{AWP}$, we will use the concept of the word we have given in Section 3.1.1, together with the concept of interval in order to define proximity formulas and word patterns. These notions, along with the concept of attributes, are necessary to define queries.

Let $\mathcal{I}$ be a set of *(distance) intervals*, defined as follows:

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

Definition 3 *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 $\mathcal{I}$.

The operator $\prec_i$ used in Definition 3 is called *proximity operator* and is used to define the concepts of *order* and *distance* between the words in a text document. Using the results of these operators we limit the bounds between two words. Thus for example, the type $w_1 \prec_{[l,u]} w_2$ indicates that the word w_1 is placed before the word w_2 and separated by at least l and at most u other words.

Example 3 *Consider the following proximity formulas:*

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

The proximity formula:

$$\textsf{information} \prec_{[0,0]} \textsf{retrieval}$$

indicates that the word retrieval *appears exactly before the word* information*. This is a way to encode the string* information retrieval *in $\mathcal{AWP}$. We can also create larger expressions using proximity operators. For example, in the proximity formula:*

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

the word topics *is constrained to precede the word* information *by at least 0 and up to 5 words. Similarly the word* information *must appear just before the word* retrieval.

Definition 4 *A* word pattern *is a conjunction of words that belong to* $\mathcal{V}$*, proximity formulas and boolean operators.*

Example 4 *The following example is a word pattern:*

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

In the following definitions, the symbol $\sqsupseteq$ should read as *contains*, and the expression $B \sqsupseteq wp$ is a way to say that the value of the attribute B contains a pattern of words as specified by wp.

Definition 5 *A query is a notation of the form:*

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

where each $A_i, B_i \in \mathcal{A}$*, each* s_i *is a text value that belongs to a document and every* wp_i *is a word pattern.*

Example 5 *The following formula is a query:*

$$\begin{aligned} &\textit{AUTHOR} = \text{“Hector Garcia-Molina”} \wedge \\ &\textit{TITLE} \sqsupseteq (\textit{selective} \prec_{[0,0]} \textit{dissemination} \prec_{[0,3]} \textit{information}) \wedge \textit{structures} \end{aligned}$$

Although the queries of a user may be unrelated and involve a variety of topics, all queries are formed based on the $\mathcal{AWP}$ data model. From now on when we refer to a user query we will refer to a set of queries expressed in $\mathcal{AWP}$.

3.1.3 Query answering

We will now describe the query semantics in the $\mathcal{AWP}$ data model. In the case of information filtering, the answer to a query essentially means that the incoming document matches the query of a user. To better understand this concept, we define it formally in two steps. First, Definition 6 shows when a text value satisfies a word pattern and then, using the Definition 7, we respond to when a document satisfies a query.

Definition 6 *Let s be a text value and wp be 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 $w_1 \prec_{i_1} \cdots \prec_{i_{n-1}} w_n$ then $s \models wp$ iff there exist $p_1, \ldots, p_n \in \{1, \ldots, |s|\}$ such that, $s(p_j) = w_j$ and $p_j - p_{j-1} \in i_j$ for all $j = 2, \ldots, n$*
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 6 *The text value $s =$ "applications of selective dissemination of information" satisfies the word pattern of Example 4:*

$$\mathit{applications} \wedge (\mathit{selective} \prec_{[0,0]} \mathit{dissemination} \prec_{[0,3]} \mathit{information})$$
$$\mathit{since}\ s(1) \wedge s(3) \prec_{[0,0]} s(4) \prec_{[0,3]} s(6)$$

Definition 7 *Let d be a document and ϕ a query. The concept of d satisfying the 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 7 *Let us consider a document d of the Example 2. The value of attribute AUTHOR in document d is "Hector Garcia-Molina" and the value of attribute TITLE satisfies the word pattern*

$$(\mathit{selective} \prec_{[0,0]} \mathit{dissemination} \prec_{[0,3]} \mathit{information}) \wedge \mathit{databases}$$

according to Definition 6. Thus, d satisfies the query of Example 5

$$\mathit{AUTHOR} = \text{"Hector Garcia-Molina"} \wedge$$
$$\mathit{TITLE} \sqsupseteq (\mathit{selective} \prec_{[0,0]} \mathit{dissemination} \prec_{[0,3]} \mathit{information}) \wedge \mathit{structures}$$

according to Definition 7.

3.1.4 The expressive power of AWP

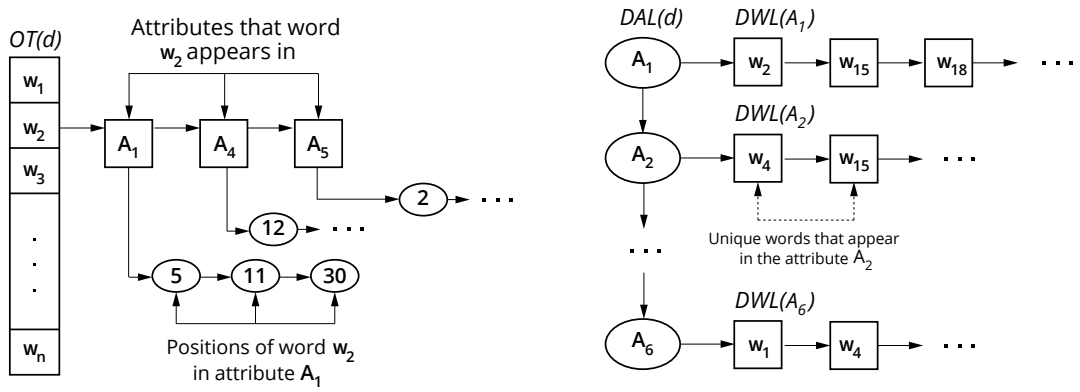
The proximity operator $\prec_i$, that is used in this thesis, have more potential to express information than the traditional information filtering model which makes use of the operator kW which means "the first word must precede the second word k number of words" [18]. Thus, for example, the expression $w_1\, kW\, w_2$ in [18] is equivalent to the expression $w_1 \prec_{[0,k]} w_2$ in the model $\mathcal{AWP}$ which we use in this thesis. Additionally, the operator kW does not have enough expressive power to express $w_1 \prec_{[l,u]} w_2$ for $l > 0$ or $u = \infty$, i.e., the first word not to be exactly next to the second and the distance of the second word from the first to be indifferent. Our model cannot express the operator kN of [18] with meaning "the first word should be at a distance k from the second, regardless of the display order". $\mathcal{AWP}$ model can, however, make use of disjunction and the sentence $w_1 kN w_2$ can be converted to $w_1 \prec_{[0,k]} w_2 \vee w_2 \prec_{[0,k]} w_1$.

In conclusion, the proximity operators of $\mathcal{AWP}$ have the same expressive power with kN, regarding the distance between words, but lack in power when we seek the existence of the *window* containing the words.

In the implementation of the algorithms, we have covered a subset of the $\mathcal{AWP}$ data model and chose not to support the proximity operators.

3.2 Algorithms for filtering

In this chapter, we will describe the algorithms we used in order to solve the filtering problem of the incoming documents [71, 75, 78]. In Section 3.2.1 we will present the BestFitTrie algorithm and its data structures. The PrefixTrie algorithm will be presented in Section 3.2.2. PrefixTrie is an extension of Tree algorithm that uses alphabetical sorting for the sets of words entering the *forest* of tries. In Section 3.2.4 we will present the ReTrie algorithm. ReTrie is a variant of BestFitTrie that reorganizes the queries based on a *clustering ratio*. Finally, in Section 3.2.5 we will present the StaR, an algorithm that utilises word statistics in order to reorganize the queries.



(a) Occurrence Table $OT(d)$ (b) Distinct Attribute List $DAL(d)$

Figure 3.1: Data structures for each document

3.2.1 The Algorithm BestFitTrie

The first algorithm we used, in order to solve the filtering problem, is the BestFitTrie algorithm, as described in the works [71, 78]. BestFitTrie algorithm uses some specially designed structures to represent queries and documents. Next, we will describe the data structures of the algorithm.

3.2.1.1 Data structures for documents

Every document that is published, must be represented by an appropriate data structure in order for the matching process to be completed as soon as possible. BestFitTrie stores published documents using two main data structures (see also Figure 3.1):

- an array for displaying words called *Occurrence Table* which is denoted as $OT(d)$ and
- a dynamically linked list used for the document's attributes, called *Distinct Attribute List*, which is denoted as $DAL(d)$.

More specifically, as shown in Figure 3.1a, the data structure $OT(d)$ is a hash table which contains every single word of the document as keys, and each element points

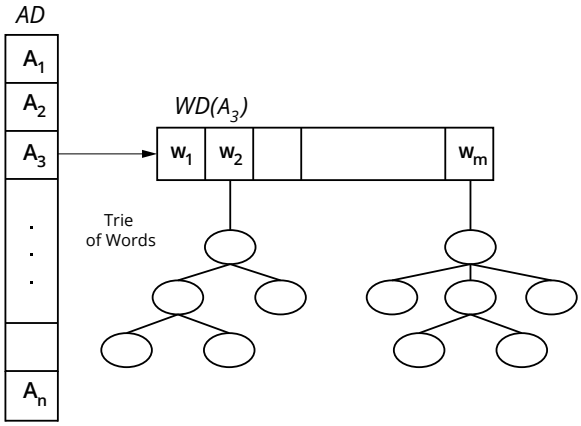


Figure 3.2: Data structures for user queries. Query index with word directory $WD(B_i)$.

to a list of attributes d in which the word appears. Each node of the list, stores the positions of the word within the attribute.

Figure 3.1b shows the linked list $DAL(d)$ which stores the unique attributes of the document, and these attributes store a list of $DWL(A)$ with the unique words shown in them.

3.2.1.2 Data structures for queries

The indexing of the queries is accomplished using respective structures in order to be easier to compare them with the incoming documents. An array of attributes AD (Attribute Directory) is used, in which each element stores a unique attribute A. Each $AD(A)$ indexes a set of queries wp (as defined in Section 3.1.2) and for every query it applies $A \sqsupseteq wp$.

In order to properly arrange the words of a word pattern set ($words(wp)$), we use an array of word entries. This *word directory*, named WD, 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 $words(wp)$.

3.2.1.3 Query insertion

We will analyze the process of importing queries and how each is effectively stored for the match to be completed. An inserted query is of the form:

k	Query $A_i \sqsupseteq wp_j$	Identifying subsets
1	$A_i \sqsupseteq$ *information*	{information}
2	$A_i \sqsupseteq$ *retrieval* $\wedge$ *information*	{information, retrieval}
3	$A_i \sqsupseteq$ *information* $\wedge$ *filtering*	{information, filtering}
4	$A_i \sqsupseteq$ *models* $\wedge$ *proximity* $\wedge$ *operators* $\wedge$ *retrieval* $\wedge$ *information*	{retrieval, models}, {proximity, operators}, {information, models}
5	$A_i \sqsupseteq$ *filtering* $\wedge$ *information* $\wedge$ *selective* $\wedge$ *dissemination*	{selective, dissemination}, {filtering, information, selective, dissemination}, {dissemination}, ...
6	$A_i \sqsupseteq$ *boolean* $\wedge$ *information* $\wedge$ *filtering* $\wedge$ *dissemination*	{boolean}, {dissemination, information}, ...

Table 3.1: Identifying subsets examples

$$A_1 \sqsupseteq wp_1 \wedge \cdots \wedge A_m \sqsupseteq wp_m$$

For each attribute A_j where $1 \leq j \leq m$, we compute wp_j and insert the words in the forest of tries that stores the words $WD(A_j)$ (Figure 3.2).

The main idea behind this data structure is to store sets of words compactly by exploiting their common elements in order to achieve better clustering, that will lead to better performance in document matching. This way, memory space is better optimized, as we create fewer nodes, compared with a data structure that does not use common elements grouping methods.

Definition 8 *Let S be a set of non-empty sets of words and $s_1, s_2 \in S$, where $s_2 \subseteq s_1$. We say that s_2 is an identifying subset of s_1 if and only if $s_2 = s_1$ or there is no $r \in S$ such that $r \neq s_1$ and $s_2 \subseteq r$.*

According to the above definition, the sets of identifying subsets of two sets of words $s1$ and $s2$ with respect to a set S is the same if and only if s_1 is identical to s_2.

Example 8 *Table 3.1 shows some examples to make these concepts better understood. In each line we give a query in the form specified by the $\mathcal{AWP}$ model where it applies*

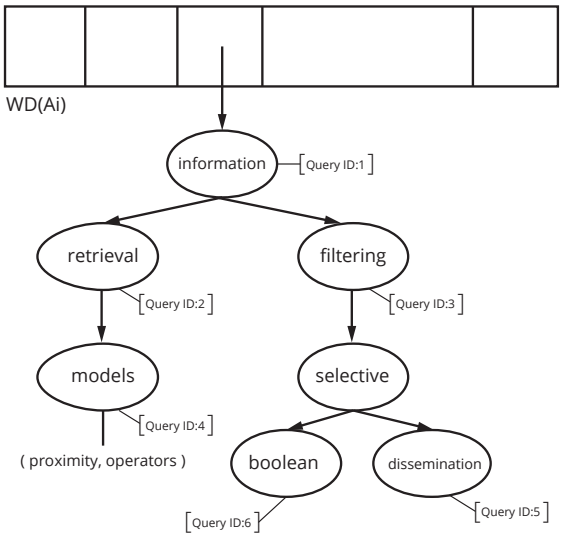


Figure 3.3: BestFitTrie trie example

that $A \sqsupseteq wp_j$ *and some identifying sets of* $words(wp_j)$ *where* $S = \{words(wp_k) : 1 \leq k \leq j-1\}$.

Each query j is represented by a word pattern wp for which $A \sqsupseteq wp_j$ is consisting of a set of words $words(wp_j)$.

Figure 3.3 shows an example of a trie created by BestFitTrie for the queries of Table 3.1, together with the $Query(n)$ and $Remainder(n)$ lists. Queries are denoted with brackets and remainders are shown in parentheses.

The creation and use of the structure $Remainder(n)$ is to allow the delayed creation of new nodes. This way, we index the words in lists and we wait for other word sets to appear. This results in new nodes to be placed in a better order, serving both the old and the newly introduced sets.

A set of words $words(wp_j)$ is organized in $WD(A)$ as follows:

Let S be a set of words already stored in $WD(A)$. To insert a new set of words s, BestFitTrie chooses the most appropriate trie T between the forest of tries of $WD(A)$. Within this trie, the most appropriate location to insert s is identified, taking into account the current organization of all the words in the forest (word directory). The criteria used to make the choice of the location of queries will be presented below.

Each trie T of $WD(A)$ has the following properties:

- Each node of T stores a set of words and other data items related to this set of words. We denote as *sets-of-words*(T) the set of all words stored in the trie T.

- The root of T is at a depth 0 and it always stores sets of words with an identifying subset of cardinality one. In general, a node n, located at depth i, stores sets of words with an identifying subset of cardinality $i + 1$.

- Each node n belonging to T, that stores a set of words s, is represented by a data structure that stores 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 words appearing on the path from the root to node n.

  - $Query(n)$: a linked list that contains the unique identifier of every query q that is contained in the word pattern wp, for which $\{w_0, \ldots, w_i\}$ is the identifying subset of *set-of-words(T)*.

  - $Remainder(n)$: if node n is a leaf, then this field is a linked list of words s that are not included in $\{w_0, \ldots, w_i\}$. If n is not a leaf (all words of s are contained in $\{w_0, \ldots, w_i\}$), $Remainder(n)$ is empty.

  - $Children(n)$: a linked list of pairs (w_{i+1}, ptr), where w_{i+1} is a word of $\{w_0, \ldots, w_i, w_{i+1}\}$ and is an identifying subset for the sets of words stored at w_i and ptr is a pointer to the node n', where $Word(n') = w_{i+1}$.

- The sets of words stored in node n of T are equal to $\{w_0, \ldots, w_n\} \cup Remainder(n)$, where $w_0, \ldots, w_n$ are the words that we find on the path from the root of T to n. A identifying subset of the words stored at node n is $w0, \ldots, w_n$.

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 9 *Let s be a set of words indexed at node n of the 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 that we find in the path from the root of T to node n. The clustering ratio of s in T,*

denoted as $ClusteRat(s,T)$*, is defined as* $ClusteRat(s,T) = \frac{|\{w_0,\ldots,w_n\}|}{|s|}$. $\{w_0,\ldots,w_n\}$ *denotes the number of words from the root* T *to node* n *and* $|s|$ *denotes the total number of words in the set* s*.*

$ClusteRat(s,T)$ is used to quantify how well the set of words s is clustered in trie T. It practically indicates how many words of set s are going to be indexed in nodes and how many of them will stay in the list $Remainder(n)$. By definition, clustering ratio takes values from the interval $0 < ClusteRat(s,T) \leq 1$. Generally, $ClusteRat(s,T)$ near 0 indicates that the query is poorly indexed and a lot of words do not create nodes and remain in the list $Remainder(n)$. A clustering ratio near 1 indicates that the majority of the words is indexed in nodes.

The algorithm for inserting a new set of words s in a word directory is as follows. The first set of words that is inserted, will create a new trie root using a randomly selected word, while the rest will be stored in the *Remainder* list in order for new nodes to be created later on. The second set of words will consider being stored at the existing trie or create a trie of its own. The algorithm BestFitTrie is searching the forest of $WD(A)$ for every trie with root that belongs to s such that, if s was indexed there, $ClusteRat(s,T')$ would be maximized.

BestFitTrie identifies the optimal node by performing a depth-first search (in depth $|s|-1$) in *all* candidate tries. If many nodes that maximize $ClusteRat(s,T')$ are found, then BFT randomly selects one among them. The path from the root to the node n is extended by adding new nodes containing the words in $\tau = (s \backslash \{w_0,\ldots,w_n\}) \cap Remainder(n)$.

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

In Figure 3.4, we present the pseudo-code that BestFitTrie uses to index the

```
   Algorithm: BestFitTrie
1  match ← Null
   // for all query attributes
2  foreach attribute A ∈ q do
3      curClusterRat ← 0
4      curPosition ← Null
       // for all candidate tries
5      foreach trie T such that root(T) ∈ s do
           // for all possible storage positions in candidate tries
              perform a DFS
6          foreach node n ∈ T such that word(n) ∈ s do
7              calculateclusterRat(s)
               // if a better position is found make a note of it
8              if clusterRat(s) > curClusterRat then
9                  curClusterRat ← clusterRat(s)
10                 curPosition ← n
       // of s cannot be indexed in any existing trie
11     if curPosition = Null then
           // create a new trie with a random word from s as a root
12         create a trie T with root(T) ← w_i ∈ s
13         curPosition ← root(T)
           // put the rest of the words in s in the
              Remainder(curPosition)
14         Remainder(curPosition) ← s\{w_i}
15     else
16         store s at node curPosition
           // put the rest of the words in s in the
              Remainder(curPosition)
17         Remainder(curPosition) ← s\{w_0, ..., w_n}
```

Figure 3.4: Pseudocode for query insertion under Algorithm BestFitTrie

user's queries. The time complexity of BestFitTrie is linear in the size of the dictionary.

3.2.1.4 Filtering incoming documents

When a new document d is inserted in the forest, BestFitTrie fills the data structures previously described. Thus, for each attribute A presented in $DAL(d)$ and for each word w of $DWL(A)$, an in-depth search is performed in tries $WD(A)$ with root w. Only tries

```
   Algorithm: FILTER
1  match ← Null
   // for all documents attributes
2  foreach attribute A ∈ DAL(d) do
       // for all tries with the distinct words in attribute as roots
3      foreach trie T such that root(T) = w ∈ DWL(A) do
4          foreach node n ∈ T do
               // if the word is contained in d
5              if word(n) ∈ OT(A) then
                   // if the words of the Remainder(n) (if any) are also
                      contained in d
6                  if remainder(n) ⊆ OT(A) then
                       // the query ids of this node are added in the
                          potentially matching queries
7                      match ← match ∪ query(n)
                       // traverse the trie in DFS
8                      n ← children(n)
9              else
                   // if the word is not contained in d no need to search
                      the subtree
10                 prune n
```

Figure 3.5: Pseudocode for filtering incoming documents

and their subtrees containing the words belonging to the document's attribute $A(d)$ are examined, and by using the hash table $OT(d)$ this is achieved much faster. For each node n of the trie, the list $Query(n)$ gives us the unique identifiers which are stored in a data structure. Once the process is completed, the document is checked if it has matched all the attributes of each query and that their users are notified of the inserted document. This process is repeated for all the words of $DWL(A)$ and for all the attributes of $DAL(d)$.

Figure 3.5 shows the pseudo-code for filtering of incoming documents by BESTFIT-TRIE.

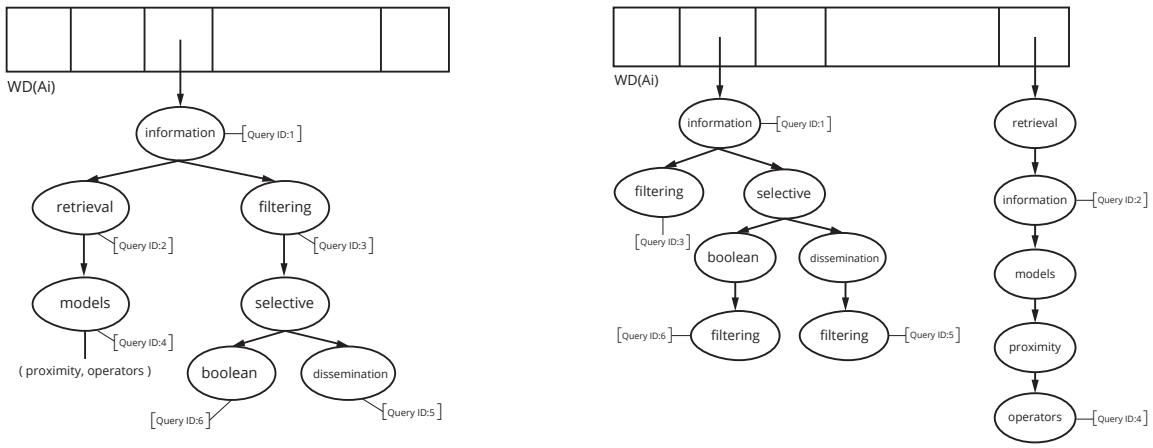
3.2.2 The Algorithm PrefixTrie

To evaluate the performance of BestFitTrie we chose to implement the Algorithm PrefixTrie as described in [71, 78]. PrefixTrie, following the logic of Tree [74], uses tries in order to index queries. This logic was based on the development of SIFT system, as indicated in work [74], where it was observed that when users subscribe on systems in the same area or topic, they tend to use the same or similar words and expressions for their needs. This could lead in a large number of queries created, using similar words. Thus, PrefixTrie, which is an extension of Tree algorithm [74], uses alphabetical sorting for the sets of words entering the forest and attempts to put these sets in tries where they will be stored more compactly. As a result, grouping improved the overall speed of the process and contributed to saving disk space.

Since PrefixTrie examines only the prefixes of word sequences in lexicographic order, to identify common parts, it misses many opportunities for clustering.

BestFitTrie constitutes an improved version of PrefixTrie, as it uses the PrefixTrie's basic idea with some improvements. BestFitTrie handles each query as a set of words and not as a sorted sequence and searches exhaustively the forest of tries to discover the best place to introduce a new set of words. Finally, the use of the $Remainder(n)$ structure offers the possibility of not generating nodes out of words directly but to wait until a new set of words to be inserted. These heuristic methods, used by BestFitTrie, offer an organization that mainly depends on the order in which the queries will enter into the forest. In order to solve this problem we will present two *reorganization* methods that are referred in [71] and in [80].

A common assumption between BestFitTrie and PrefixTrie is that queries will have many words in common. This scenario is 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 or elections) or even unexpected 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 avoid a significant processing load by



(a) BestFitTrie (b) PrefixTrie

Figure 3.6: BestFitTrie vs. PrefixTrie for the queries of Table 3.1

exploiting the similarities between queries to provide fast filtering times.

3.2.3 Reorganization

Algorithms BestFitTrie and PrefixTrie use heuristic methods to identify and cluster similar queries, in order to achieve better performance during the matching of the incoming documents. These heuristics provide an organization of queries that is dependent on the order of insertion of the queries in the forest. In this chapter, we will study methods to improve the queries' clustering.

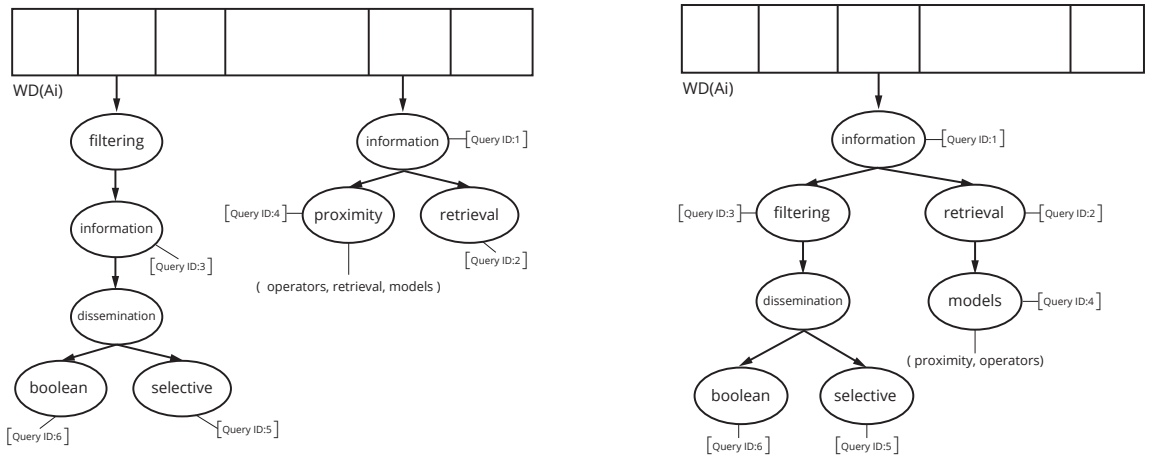
3.2.3.1 The order of queries insertion

From the way we organize the queries in the forest (using BestFitTrie), the order in which these queries are inserted into the forest is of fundamental importance, since it drastically affects the creation of new tries. These tries will then be used as the basis on which the upcoming queries will be indexed.

Example 9 *Consider the queries shown in Table 3.1. Figure 3.6a presents how the forest is formed if the queries are introduced in the order listed in Table 3.1. However, if we change the order of the queries insertion,* BestFitTrie *would create an entirely different forest. Inserting queries in the order of Table 3.2, results to the forest of Figure 3.7a.*

Order	Queries
1	$s_5 = \{filtering, information, selective, dissemination\}$
2	$s_3 = \{information, filtering\}$
3	$s_6 = \{boolean, information, filtering, dissemination\}$
4	$s_1 = \{information\}$
5	$s_4 = \{models, proximity, operators, retrieval, information\}$
6	$s_2 = \{retrieval, information\}$

Table 3.2: Queries insertion order example



(a) $WD(B)$ based on Example 9

(b) $WD(B)$ based on Example 10

Figure 3.7: Different query insertion order

As shown in Example 9, for the same set of words but using a different insertion order, we receive two different forests with different number of nodes, clustering ratio and therefore different performance results. The problem is to find the order of the insertion of the queries that optimizes query execution. We will propose two different ways to address this problem and improve the performance of our algorithms. If we consider the problem of clustering as a search problem, with the search space representing all the possible organizations of nodes that can be achieved, then BestFitTrie provide us with a greedy heuristic witch may not be considered optimal but one of the acceptable solutions.

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. When we seek the best

possible performance in the reorganization process, we must decide the appropriate criterion used for these reorganizations to occur. It is necessary to define what we will be reorganizing and how often.

1. **Reorganization based on intervals**: Surely the simplest approach is to reorganize the index of queries on specific intervals (i.e., after a specific number of query insertions). This approach can result in a better-formed database as often as desired, but ignores the clustering criteria, while invoking reorganization events when not needed.

2. **Reorganization based on clustering ratio**: Another option is for us to reorganize the forest when the clustering ratio gets below a specific rate.

3. **Reducing branching ratio**: A third approach would be to consider and reduce the fan-out (branching ratio) of trie nodes. This would reduce the number of subtrees and could lead in smaller document filtering times.

Every technique has its advantages and disadvantages. Some of these techniques are used for both reorganization algorithms (ReTrie and StaR) that we will present in the next section.

3.2.4 Reorganization using ReTrie

We consider Algorithm ReTrie [71, 78, 80]. ReTrie is a variant of BestFitTrie that reorganize the queries that have a clustering ratio below a specified threshold. It is a combination of the above techniques (1 and 2), in order to achieve better reorganization results. The main idea is as follows, we reorganize our forest when a particular number Q of queries is reached, but we will individually examine the clustering ratio for each query and only when it falls below a threshold c that we set, will the reorganization occur. Before describing ReTrie in more detail, we will determine which queries are *under-clustered*, i.e., are considered to be below the clustering ratio limit.

Definition 10 *Let s be a set of words that have been indexed in node n of trie T. The set s is considered* under-clustered *iff $ClusterRat(s,T) < c$ where $0 \leq c \leq 1$ is a clustering threshold called the* minimum clustering ratio*.*

In order to record the clustering ratio of each set of words s, ReTrie utilizes a *clustering array* (CA) which contains an entry for every set of word of $WD(B)$. Each CA entry contains a pointer to the positions of s (of $WD(B)$) and a number that corresponds to $ClusteRat(s,T)$. When a new set of words s is indexed in node n of the trie, then the clustering ratio of s stored in CA is initialized based on the Definition 9. If the $Remainder(n)$ list is expanded to create new nodes, then the clustering ratios of the other sets of words stored at n are updated based on CA. If *Remainder* list is not expanded, no other update is needed to the array CA.

The pseudo-code of the algorithm ReTrie that is presented in Figure 3.8, searches and repositions the sets of words that are below threshold c that we have set and are badly clustered.

All under-clustered sets of words are identified by scanning the CA array. For every s where $ClusterRat(s,T) < 0$, ReTrie is scanning for all the nodes of the forest $WD(B)$ that could index s and result in a $ClusterRat(s,T)$ greater than c. Essentially, using the mechanisms of BestFitTrie and making the insertion from the beginning of the query, the algorithm also chooses to make the insertion in the new node only when the node offers a high $ClusterRat(s,T)$ than the existing clustering ratio. Finally, if there is a shift of s, an update of CA also occurs, and the set of words is removed from the previous position that it was indexed at.

Algorithm ReTrie has the potential to improve the clustering of the queries, unlike BestFitTrie where not all the queries have the same clustering opportunities. This can be explained as follows: When a set of words s corresponding to formula $B \sqsupseteq wp$ needs to be indexed, the clustering algorithm searches every trie of the forest $WD(B)$ for a node such that the clustering ratio $ClusterRat(s,T)$ is maximized.

It is easy to understand that the search for the optimal position is influenced by the number of tries and nodes. The higher the number of candidate positions to insert s, the higher the probability that s will be better indexed. This is shown in Figure 3.9.

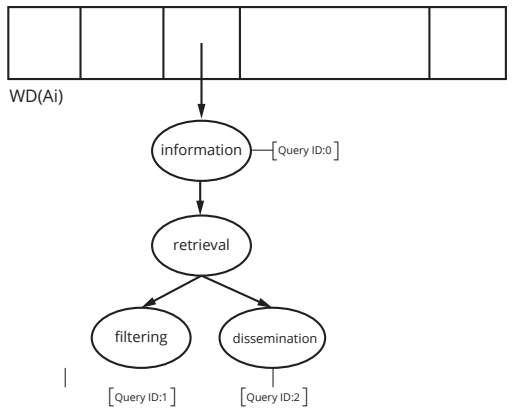
```
   Algorithm: ReTrie
   // for all document attributes
1  foreach attribute A do
       // for all stored sets of words where N is the number of the
          sets
2      for i = 0 to N do
           // identify under-clustered ones
3          if CA[i].clusterRat < c then
4              let s be the i-th set of words
               // for all candidate tries
5              foreach trie T such that root(T) ∈ s do
                   // for all possible storage positions in candidate
                      tries
6                  foreach node n ∈ T such that word(n) ∈ s do
7                      calculateClusterRat(s, T)
                       // if a better position is found make a note of it
8                      if ClusterRat(s, T) > curClusterRat then
9                          curClusterRat ← clusterRat(s, T)
10                         curPosition ← n
           // if the best position found is better than the initial
11         if curPosition ≠ CA[i].position then
               // move s there
12             move s to curPosition
               // and update CA
13             CA[i].position ← curPosition
14             CA[i].position ← curClusterRat
```

Figure 3.8: Pseudocode for query insertion under Algorithm ReTrie

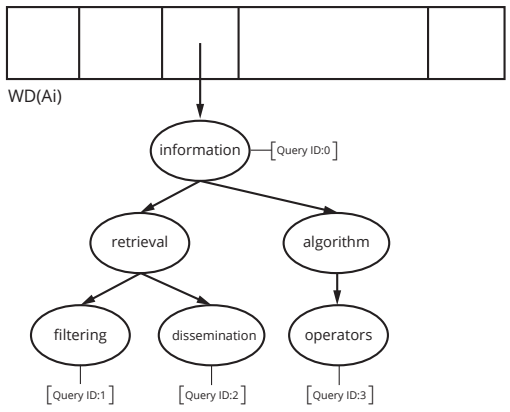
Consider the forest shown in Figure 3.9a, consisting of a single trie T and indexing three sets of words:

- $s_0 = \{information\}$
- $s_1 = \{information, retrieval, filtering\}$
- $s_2 = \{information, retrieval, dissemination\}$

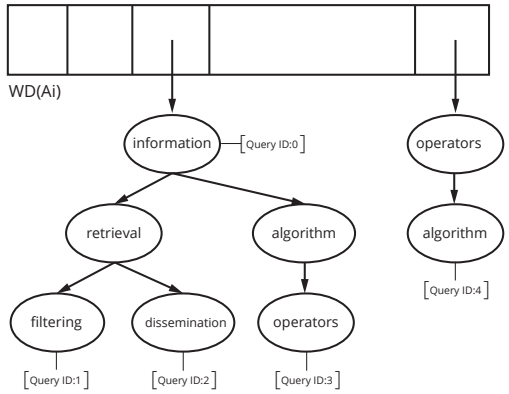
When a new set of words $s_3 = \{information, algorithm, operators\}$ is inserted into the forest, it is clustered only under one word (Figure 3.9b). The clustering ratio of



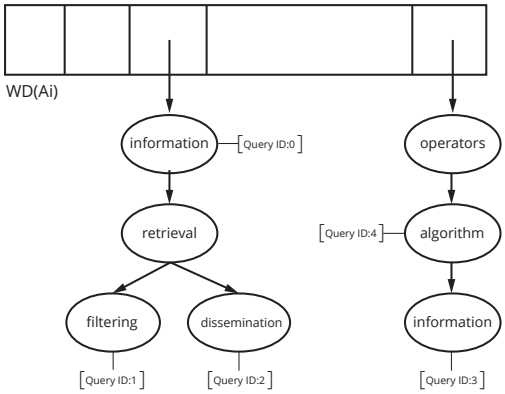
(a) $WD(B)$ with only one trie and three sets of words (s_0, s_1 and s_2)



(b) $WD(B)$ after insertion of $s_3 = \{information, algorithm, operators\}$



(c) $WD(B)$ after the insertion of $s_4 = \{algorithm, operators\}$



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

Figure 3.9: Query insertions and reorganization achieved by ReTrie

s_3 is $ClusterRat(s_3, T) = 0.33$. Upon arrival of $s_4 = \{algorithm, operators\}$, a new trie is created, since s_4 cannot be indexed under an existing trie, creating the forest of tries shown in Figure 3.9c). It is obvious though, that there is a better position for s_3 to be indexed and this position is along with s_4, as shown in Figure 3.9d), where $ClusterRat(s_3, T) = 0.66$. We notice that the words *retrieval* and *operators* of the forest $WD(B)$, as shown in Figure 3.9c), appear in two nodes each and that redundancy in nodes is the main problem that slows down the filtering process. After reorganizing the forest of Figure 3.9d) the redundant nodes of these two words are removed. Generally, it is impossible to remove all redundant nodes in the forest (e.g., the node *information*), but we focus our efforts on minimizing nodes by reorganization.

3.2.5 Reorganization using StaR

Studying ReTrie we understand that the order in which the queries are inserted in our forest, is an important factor in the organization of the index as shown in Example 9. The tries that will be created in the early stages play a crucial role because they determine the positions of the following queries. We also observed that as often as we call ReTrie after a number of queries, it cannot find any position that offers higher clustering ratio than the existing and the reposition of a large number of queries inside the forest may damage the clustering of the trie from which they are removed. These conclusions were derived from studying the perfect choice of parameters for ReTrie [78].

It is readily understood that the key to the proper clustering of the tries is based on the correct insertion of the queries.

More precisely, the ReTrie algorithm firstly organises queries into tries and then maintains a data structure that monitors the number of poorly clustered queries in the forest. These purely clustered queries are those that are not indexed in the best possible position according to the existing tries. When a certain threshold of poorly clustered queries is reached, the reorganisation process is triggered and all poorly indexed queries are examined and re-indexed. By choosing to reposition only poorly indexed queries, the ReTrie algorithm misses many available reorganisation options and is bound to use the existing tries. This means that new trie creation is very rare. On the contrary, the StaR algorithm reorganises all newly inserted queries by employing query ordering and word frequency heuristics to avoid exploring all possible query organisations.

For that reason, we used the algorithm StaR [71, 80], that can affect the order of the queries insertion in the forest. The main idea behind the algorithm is as follows: The forest accepts queries without being aware of the significance of the words inserted. It would be useful if we could delay the creation of the tries until a good sample of the significance of the words could be provided. Then, by exploiting the information for the significance of the words, the forest would proceed to the appropriate index reorganization.

In the first stage, StaR gets the queries and indexes them in the same way as

keyword k	$freq(k)$
information	6
retrieval	2
filtering	3
models	1
proximity	1
operators	1
selective	1
dissemination	2
boolean	1

Table 3.3: Statistics Table. Scoring Example in Table 3.4

BestFitTrie but in a temporary forest. The purpose of the temporary forest is to index the queries until they pass to the final forest. This way, both users that have their queries in the final forest and those who have recently introduced queries in the forest but they are not have passed to the final stage of the organization, will be kept informed. Additionally, we maintain the matching speed at the same level as with BestFitTrie until we have the final forest resulting from the StaR algorithm. Along with the insertion of new queries in the forest, StaR maintains a hash table named *Statistics* which stores the number of appearances (*frequency*) for each unique word (keyword) of the set of word patterns, that have appeared in the forest by this time. An example of a statistics table can be seen in Table 3.3 below.

The query itself is also stored in another table in the memory in order to be rated at a later time, without having to search the entire forest to recover it.

Before describing the second phase of the algorithm, we will give the definition for the *scoring* that every s is receiving based on the statistics of words.

Definition 11 *Let $s = \{w_0, \ldots, w_n\}$ be a set of words and $freq(w)$ be the frequency of appearances of a word. The* score *of s is denoted as:*

$$score_s = \sum_{i=0}^{n} freq(w_i)$$

Practically, score *is the sum of the word appearances of the set of words s as this is*

Query	Score
$s_1 = \{information\}$	6
$s_2 = \{retrieval, information\}$	8
$s_3 = \{information, filtering\}$	9
$s_4 = \{models, proximity, operators, retrieval, information\}$	11
$s_5 = \{filtering, information, selective, dissemination\}$	12
$s_6 = \{boolean, information, filtering, dissemination\}$	12

Table 3.4: Scoring of queries in Table 3.1

stored in table Statistics.

In the second phase after a number of Q queries have been inserted into the forest, wherein Q is a parameter that sets the beginning of the operation, StaR starts importing them in the final forest. Using the *statistics* table, StaR calculates the score of each query as indicated by Definition 11, sorts the terms based on the *Statistics* table and organizes its insertion order to the final stage. The insertion order and the clustering of the terms can be made either in descending or ascending order of score. StaR may insert first the queries with low rating (i.e., with low frequency) and then those higher rating (i.e., with higher frequency). Alternatively, it can first enter the queries with the highest score, i.e., those containing words that appear more often in the entire dictionary. In Chapter 4 we will further analyze the advantages and disadvantages of the ascending and descending insertion order and how this choice affects the creation of the forest.

Previously, we mentioned that except for the queries we also sort their terms in order to create better tries. Thus, the creation of any new root for a trie T in $WD(B)$ is not random; when creating a new trie we always set the first word as root (even if this word represents a word with the lowest frequency of the higher one), while the way new tries are created is affected by the terms order in the $Remainder(n)$ list. So when the list $Remainder(n)$ is about to create more than one new nodes, it creates them reflecting their in-between ranking in the hierarchy of parent-child nodes. Finally, StaR inserts the queries exactly in the manner described earlier for BestFitTrie.

```
   Algorithm: StaR
1  queryCounter ← 0 // query counter
2  tempForest ← Null // create temporary forest
3  finalForest ← Null // create final forest
   // for each inserted query
4  foreach wp do
       // if buffer is not full
5      if queryCounter%bufferSize then
           // add the query in the temporary forest exactly as
              BestFitTrie
6          tempForest.add(wp) // update the Statistics array
7          Statistics.add(wp) // store the query in order to score it
              later
8          queryTable.add(wp) // update the counter that another query
              has been inserted in the temporary forest
9          queryCounter++
10     else
11         Sort(queryTable, Statistics) // Sort the queries based on
              Statistics
12         finalForest ← queryTable // insert the queries to the final
              forest
13         tempForest ← Null // empty the temporary forest
```

Figure 3.10: Pseudocode for query insertion under Algorithm StaR

The pseudo-code of the algorithm StaR is presented in Figure 3.10.

Example 10 *In Example 9 we saw that the order of insertion is essential for the creation of the trie. If we use* StaR *to insert the queries of Example 9 in the same order, then Figure 3.7a is representing the* temporary forest. *While the* table of statistics *is formed as follows:*

StaR *is able to sort the set of words s of each query in ascending or descending order, based on the Statistics hash table 3.3 (i.e., the frequency of appearances of each word, leads to the creation of two variations of algorithm* StaR *identified as* StaR-F, StaR-R*), and must choose to organize the insertion order of all queries in the final forest in ascending or descending order of* $Score_s$ *(i.e., the score of queries leads to the creation of two variations of algorithm* StaR *identified as* StaR-H, StaR-L*). This gives us four possible combinations for the query organization before they enter the final*

*forest (*StaR-H StaR-F*,* StaR-H StaR-R*,* StaR-L StaR-F*,* StaR-L StaR-R*).*

In this example, we choose to sort the words of each query in descending order of appearances while we insert queries in descending order of Score (i.e., StaR-F*,* StaR-H*). Based on the above we have the following order of insertion:*

- $s_6 = \{information, filtering, dissemination, boolean\}$
- $s_5 = \{information, filtering, dissemination, selective\}$
- $s_4 = \{information, retrieval, models, proximity, operators\}$
- $s_3 = \{information, filtering\}$
- $s_2 = \{information, retrieval\}$
- $s_1 = \{information\}$

Having described our problem and the algorithms used, we can now analyze the algorithms performance in the next chapter.

3.2.6 Parallelization of the indexing and filtering process

An elegant way of enhancing the performance of all algorithms is by parallelizing the filtering process. Such an improvement is critical as filtering algorithms are expected to process high volumes of incoming information as efficiently as possible. Here we identify one proof-of-concept parallelization variation for all Algorithms described.

Document parallelization (MtF) is a straightforward solution where each one of the available threads ϑ_d is assigned to execute the filtering process for a sub-set of incoming documents $\{D_j, \ldots, D_m\}$. The filtering procedure is executed as described in Algorithm 3.5. Thus, the forest can be searched simultaneously by more than one threads.

In the same spirit, we also used query parallelization (MtP) where each one of the available threads ϑ_p is assigned to execute the indexing process for a sub-set of queries $\{P_j, \ldots, P_m\}$. The indexing procedure is executed as described in Figure 3.4. The indexing of the queries though, cannot be executed simultaneously because the forest of tries is actually one global variable that can be manipulated only by the current active thread. *Mutual exclusion* was used, in order to prevent race conditions.

Chapter 4

Experimental evaluation

In this chapter, we discuss the experimental evaluation of the Algorithms BestFitTrie [71, 78], described in Section 3.2.1, PrefixTrie [74] described in Section 3.2.2, Retrie [71] and StaR [71, 80], presented in Section 3.2.4 and Section 3.2.5 respectively. At first, we are going to demonstrate the data and the parameters used in our evaluation process, the technical configuration of our algorithms and the implementation of the experiments. Finally, we present and analyze the results obtained from the evaluation of the algorithms.

4.1 Data and query sets

To evaluate the proposed algorithms, sets of incoming documents are required. For our experiments, we use two sets of documents described in the following sections.

4.1.1 The *DBpedia* corpus

The first set of documents used in our experiments is based on the *DBpedia* corpus. It consists of a wide and thematically unfocused set of documents, contains more than $3.7M$ documents, has a total vocabulary of $3.14M$ words and its average document size is 53 words. Each document is an extended Wikipedia abstract downloaded from the *DBpedia* website (http://wiki.dbpedia.org/Downloads39). Table 4.1 summarizes some key characteristics of the *DBpedia* corpus.

Description	Value
Vocabulary size	$3.14M$
Average document size (words)	53
Maximum document size (words)	14.425
Minimum document size (words)	1
Maximum word size (letters)	57
Minimum word size (letters)	1

Table 4.1: Characteristics of the DBpedia corpus.

These documents topics, in the DBPedia collection, are of general content but they will help us evaluate how documents from different topics are clustered in a forest.

The collection of words follows a *Zipf* distribution, i.e., few words have high frequency of appearance in the collection and most of the words have small frequency of appearance. The Zipf distribution appears in most collections of natural language texts, thus, the DBpedia corpus is a very good choice.

4.1.2 The neural networks corpus

The second set of documents is composed of research papers in the area of Neural Networks and we will refer to them as the *NN* corpus. The set is downloaded from ResearchIndex and it was originally used in [29]. The *NN* set is composed of 10400 documents and contrary to *DBpedia* is a collection of documents of the same area and interest. Table 4.2 provides some key characteristics of the *NN* collection. *NN* collection also follows the *Zipf* distribution.

4.2 Queries creation

There is not a public available set of queries for the sets of documents. Thus, we created a set of queries in order to use them as *users queries*. To this end, we edited the two corpuses (*DBpedia, NN*), removing the most common words along with everything that it could be considered as noise (e.g., very long, rare or misspelled words).

Description	Value
Vocabulary size	323.902
Average document size (words)	3.151
Maximum document size (words)	56.434
Minimum document size (words)	1
Maximum word size (letters)	35
Minimum word size (letters)	1

Table 4.2: Some of NN set's characteristics

After editing these collections, we end up with vocabularies that were used for the creation of different users queries, in order to test different filtering scenarios for our algorithms. For every collection, we chose random terms from the available vocabularies for user queries and for every set of queries we created different subsets with different average length of terms.

Using this procedure, we have created *3 Million* queries.

4.2.1 First query collection

The *first query collection* contains queries formed by conjunctions of different terms; each term conjunct is selected equiprobably among the set of words forming the *DBpedia* corpus vocabulary ($3.14M$) and the set of wikipedia document titles. Due to the nature of the *DBpedia* corpus and the corresponding vocabulary size, the constructed queries are expected to cover a wide variety of topics and, thus, share few common words between them. This restricts clustering opportunities and makes this setting a *stress test* for the filtering performance of the algorithms, as they are forced to identify and exploit the few commonalities between the indexed queries. For this query set, we select $500K$ documents' extended abstracts from *DBpedia* and use them as the incoming documents.

	Collection	
Description	First	Second
Vocabulary size	$650K$	$551K$
Words	$25M$	$15M$
Average query length	$3-5$	$3-5$
Complex terms per query length	50%	0%

Table 4.3: Characteristics of first and second query collections.

4.2.2 Second query collection

The *second query collection* is constructed by selecting $50K$ thematically related extended abstracts from *DBpedia*. As these queries become more focused and the vocabulary of the query database is restricted, more clustering opportunities appear. In this setting, the performance of the different algorithms is expected to be similar, as all will exploit the many clustering opportunities offered. Notice that the $500K$ incoming documents utilized in this section are the same ones used in the first query collection, since we aim to study the behavior of our algorithms when varying the query set.

4.2.3 Third query collection

The aim of third and final collection is to measure how the algorithms behave when using a great volume of irrelevant terms (noise) along with very specific terms and how each algorithm will perform.

This collection collection consists of queries from the dictionary resulting from the NN documents and has $62K$ unique words. Based on this, we created $1.5M$ queries without using complex terms. Subsequently we created an additional $1.5M$ queries with common words from the DBpedia collection. The size of this single dictionary is $13,910$. Finally, these $3M$ queries were indexed and we matched all of the NN collection of documents, namely $10,400$ documents related exclusively to the neural networks.

4.3 Evaluation criteria

The criterion that summarizes our algorithms' performance is the time the incoming documents need to be matched. The measurements were performed in order to determine the performance of the Algorithms BestFitTrie, PrefixTrie, ReTrie and StaR, in single-core and multi-core environments. We also use the size of incoming queries and documents and the number of threads in order to evaluate the performance of each algorithm when those variables are altered.

The time shown in the graphs is wall-clock time and the results of each experiment are averaged over 5 executions to eliminate any fluctuations in time measurements. Thus, we will present our results focusing on the time difference in indexing and filtering processes between sequential and parallel execution and between our algorithms.

In addition, we are going to present the variation resulting to the optimal performance per thread in relation to the overall improvement in the filtering process.

4.4 Experimental setup

All the algorithms shown in the experiments of this section were implemented in C++. For the parallelization of the indexing and filtering process, we used the <thread> C++ library. We used a Blade server with 4 Xeon 2.13GHz processors, 8 cores, 64 threads and a main memory of $264GB$ running Debian Linux. The time shown in the graphs is wall-clock time and the results of each experiment are averaged over 5 executions to eliminate any fluctuations in time measurements.

4.4.1 Algorithm parameters and configuration

There is a number of parameters, which affect the performance of the presented algorithms that have to be determined and set. For our evaluation, we use a clustering ratio of 0.8 for ReTrie, while query reorganization for under-clustered queries is invoked every $I_P = 125K$ query insertions. Regarding StaR, in our experiments we are going to use the StaR-LR variation and the reorganization will be always occurring

Parameter	Description	Baseline value	Range
I_P	Number of incoming queries	$500K$	$500K - 3M$
I_D	Number of incoming documents	$500K$	$500K$
DB	Number of queries indexed in the database	$3M$	$500K - 3M$
ϑ	Number of threads used for the indexing and filtering process	1	$1 - 80$

Table 4.4: Parameters' description, their baseline values and their range.

at $500K$ queries. All the above parameters were selected based on the measurements of [78]. Finally, for the parallelization of the indexing and filtering process we utilized a set of 4, 16, 25, 50 and 80 threads.

The baseline values for each tunable parameter in the experimental evaluation are:

1. The number of incoming queries $I_P = 500K$
2. The number of incoming documents $I_D = 500K$
3. The query database size $DB = 3M$
4. The threads used in the filtering process $\vartheta = 1$

Table 4.4 summarizes the parameters examined in our experimental evaluation. For more details about the parameter setting we refer the interested reader to [71, 74].

4.5 Results for first collection

In this section, we are going to discuss the results we received from our first collection. More specifically, we will evaluate BestFitTrie, PrefixTrie, ReTrie and StaR algorithms using the *DBpedia* collection as described in Section 4.1.1. We will present the indexing and filtering times of the algorithms for both serial and multithreading versions.

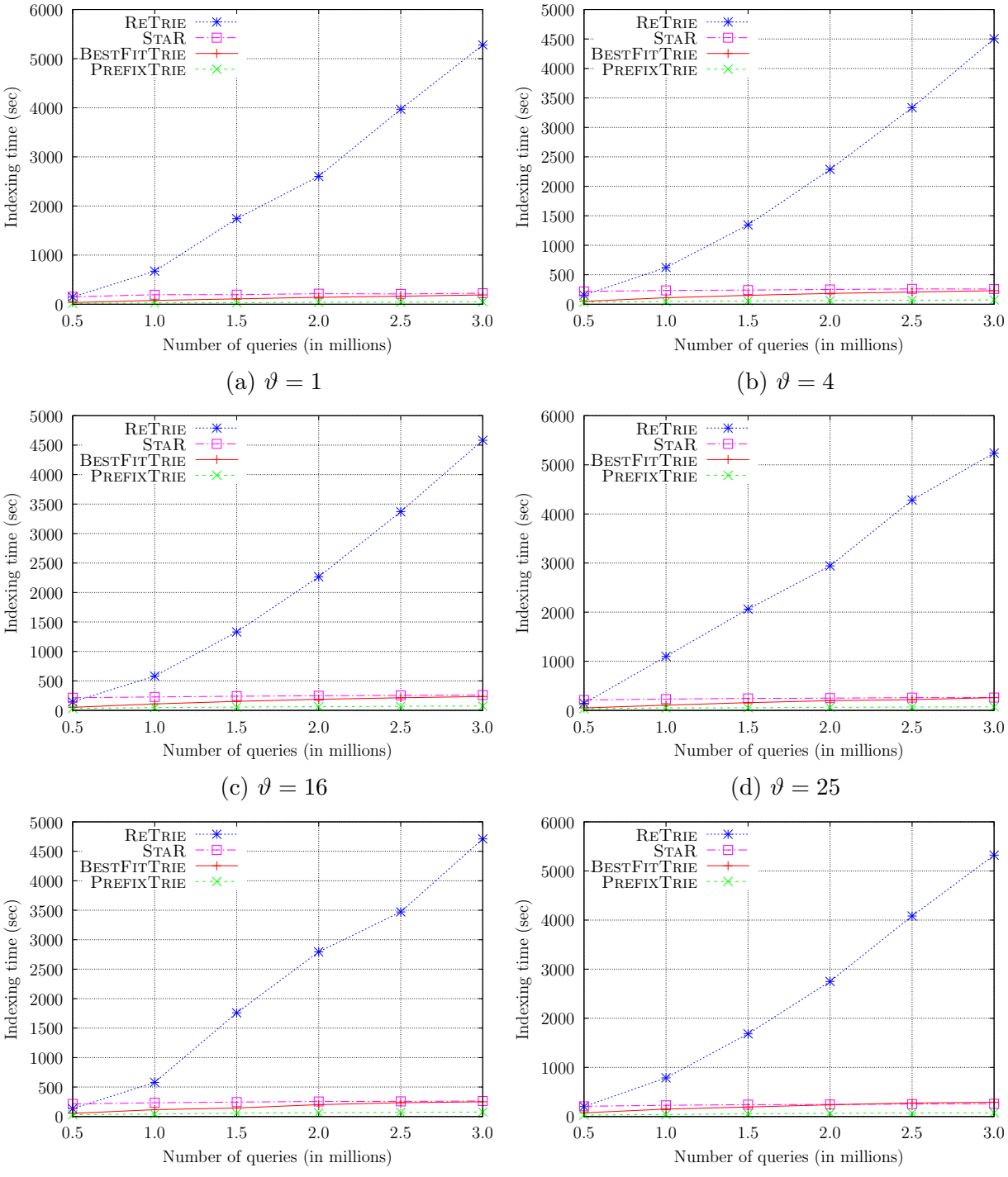
We also present the most important results to understand how the algorithms perform when using threads and without them. For more information regarding the way the algorithms create the forest you can refer to [78].

4.5.1 Evaluating indexing time

This section presents the time every algorithm requires to index 3 million queries. As expected, the time needed for indexing is increased as the number of queries is increasing.

In all our experiments we found that despite the number of threads created for indexing the queries, almost all our algorithms took noticeably more time to index the number of queries compared to their serial versions (Figure 4.2). This is a logical consequence of the manner that the four algorithms are inserting the queries in the forest, which is in fact one global variable. That means that only one thread at a time can manipulate this variable. By using mutual exclusion (mutexes), in order to prevent race conditions and to ensure the integrity of the final results, the forest of queries is locked by each thread every time queries are inserted in the forest, essentially making the various threads waiting for the occasional active thread to finish its work, resulting in worse insertion times than a serial insertion of queries. Our results showed from almost identical times, up to 2 times slower indexing when using multiple threads.

The only exception was ReTrie, were we actually had a mediocre improvement in most of the variations of the algorithm, especially when using 4, 16, 25, and 50 threads. The indexing performance in ReTrie increased from $23,43\%$ to $41,26\%$ compared to the serial version of the algorithm. In Figure 4.2 we can see all the algorithms serial times compared to their multithreading versions when indexing 3 million queries at once. In Figure 4.1 we can see a comparison of all algorithms per number of threads used. ReTrie is also the algorithm that takes the most time to index the specific queries due to the reorganization process, followed by StaR and BestFitTrie. On the other hand, due to the usage of alphabetical sorting, PrefixTrie is the algorithm with the smallest time difference and indexing time amount in general but this also results in having the worse filtering times from all the other algorithms.

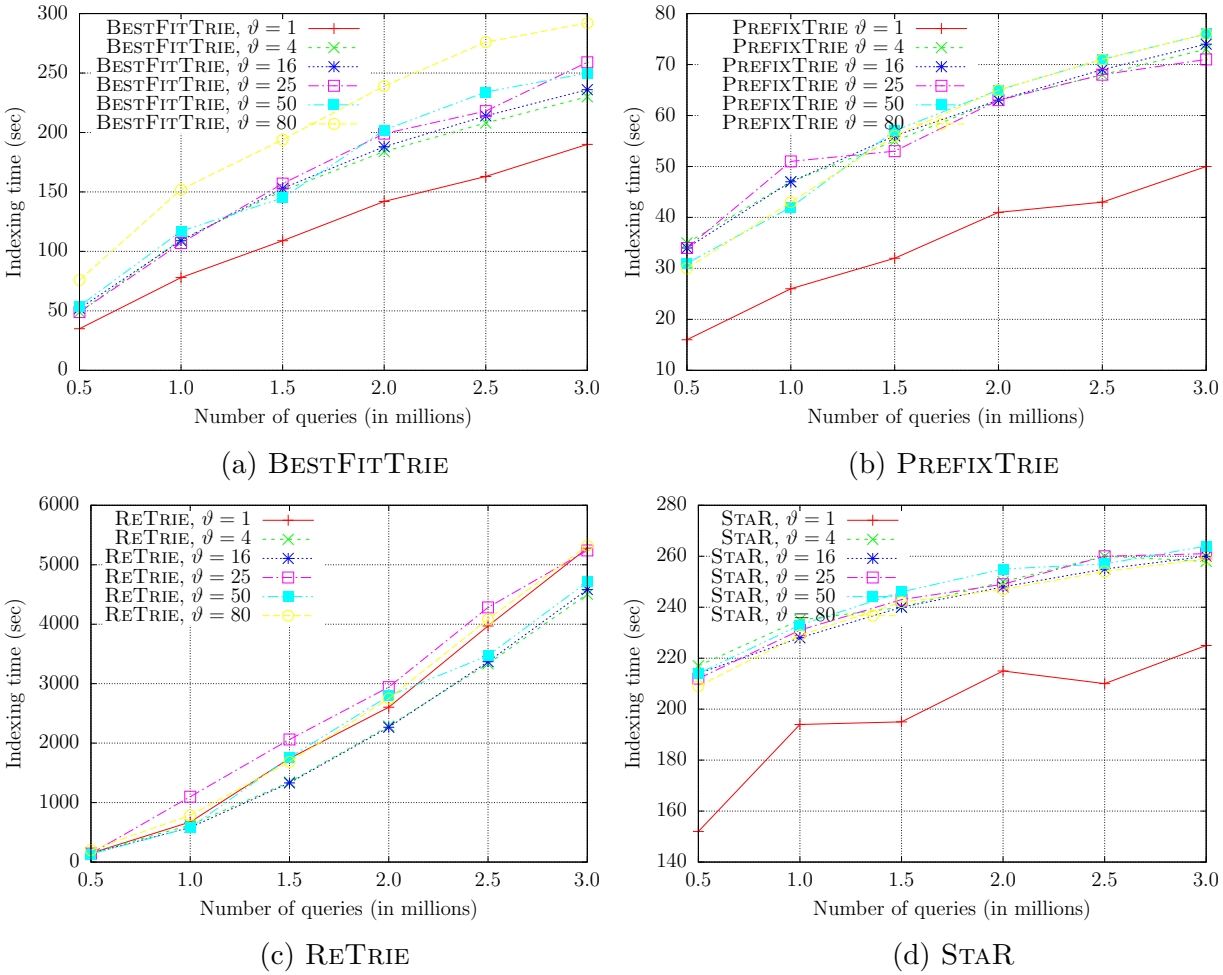


(a) $\vartheta = 1$ (b) $\vartheta = 4$

(c) $\vartheta = 16$ (d) $\vartheta = 25$

(e) $\vartheta = 50$ (f) $\vartheta = 80$

Figure 4.1: Evaluating indexing time for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$), when varying the database size from $0.5M$ to $3M$ and $I_P = 500K$. In all cases, PrefixTrie has the fastest performance, BestFitTrie and StaR are very close while ReTrie by far is the slowest.



(a) BestFitTrie (b) PrefixTrie

(c) ReTrie (d) StaR

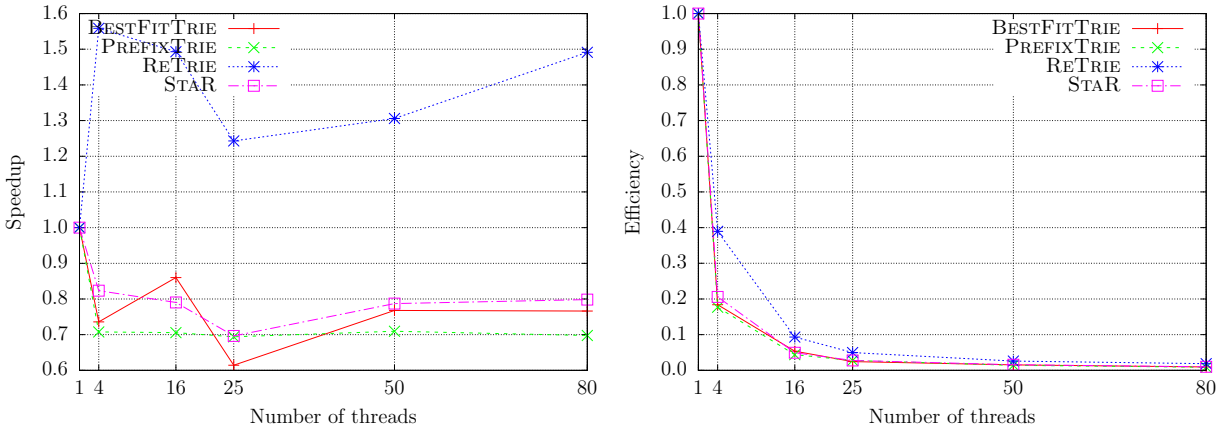
Figure 4.2: Evaluating indexing time for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$), when varying the database size from $0.5M$ to $3M$ and $I_P = 500K$. In most cases, a single thread ($\vartheta = 1$) has the fastest performance. Increasing the number of threads leads to slower performance for all algorithms but ReTrie.

4.5.1.1 Indexing speedup and efficiency

Speedup and efficiency are two performance metrics used in parallel applications (R.Rocha and F. Silva). Speedup $S(\vartheta)$ is a measure of performance. It measures the ratio between the sequential execution time and the parallel execution time. Speedup $S(\vartheta)$ is defined as follows:

$$S(\vartheta) = \frac{T(1)}{T(\vartheta)}$$

where $T(1)$ is the execution time with one processing unit and $T(\vartheta)$ is the execution time using ϑ number of threads.



(a) Comparing speedup

(b) Comparing efficiency

Figure 4.3: Comparing speedup and efficiency of indexing, for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$), when varying the database size from 0 to $3M$ and $I_P = 3M$.

Efficiency $E(\vartheta)$ is a measure of the usage of the computational capacity. It measures the ratio between performance and the number of resources available to achieve that performance. Efficiency $E(\vartheta)$ is defined as follows:

$$E(\vartheta) = \frac{S(\vartheta)}{\vartheta} = \frac{T(1)}{\vartheta * T(\vartheta)}$$

where $S(\vartheta)$ is the speedup for ϑ number of threads.

In Figure 4.3 we present speedup and efficiency for the indexing process of all algorithms and for a several number of threads. In Figure 4.3a we can see that the only algorithm exceeding speedup 1 is ReTrie, who had a mediocre improvement in most of the variations of the algorithm. All the other algorithms are presenting a speedup of less than 1, indicating that the serial version of the algorithms had the best performance most of the times.

In Figure 4.3b we can see that the efficiency of all the algorithms is declining while the number of threads is increasing, suggesting that the serial version of the algorithms had the best efficiency.

4.5.2 Evaluating filtering time

This section discusses the results of the filtering time required to match an incoming document against a database of queries as well as the optimal number of threads

Filtering Time Improvement - 80 Threads				
Algorithm	Min Improvement	Max Improvement	Speedup (Min)	Speedup (Max)
BestFitTrie	96,22%	97,93%	26,46	48,41
PrefixTrie	96,89%	97,84%	32,18	46,23
ReTrie	96,56%	98,28%	29,08	58,12
StaR	96,53%	97,41%	28,84	38,62

Table 4.5: Filtering time improvement and speedup (times faster) for all algorithms.

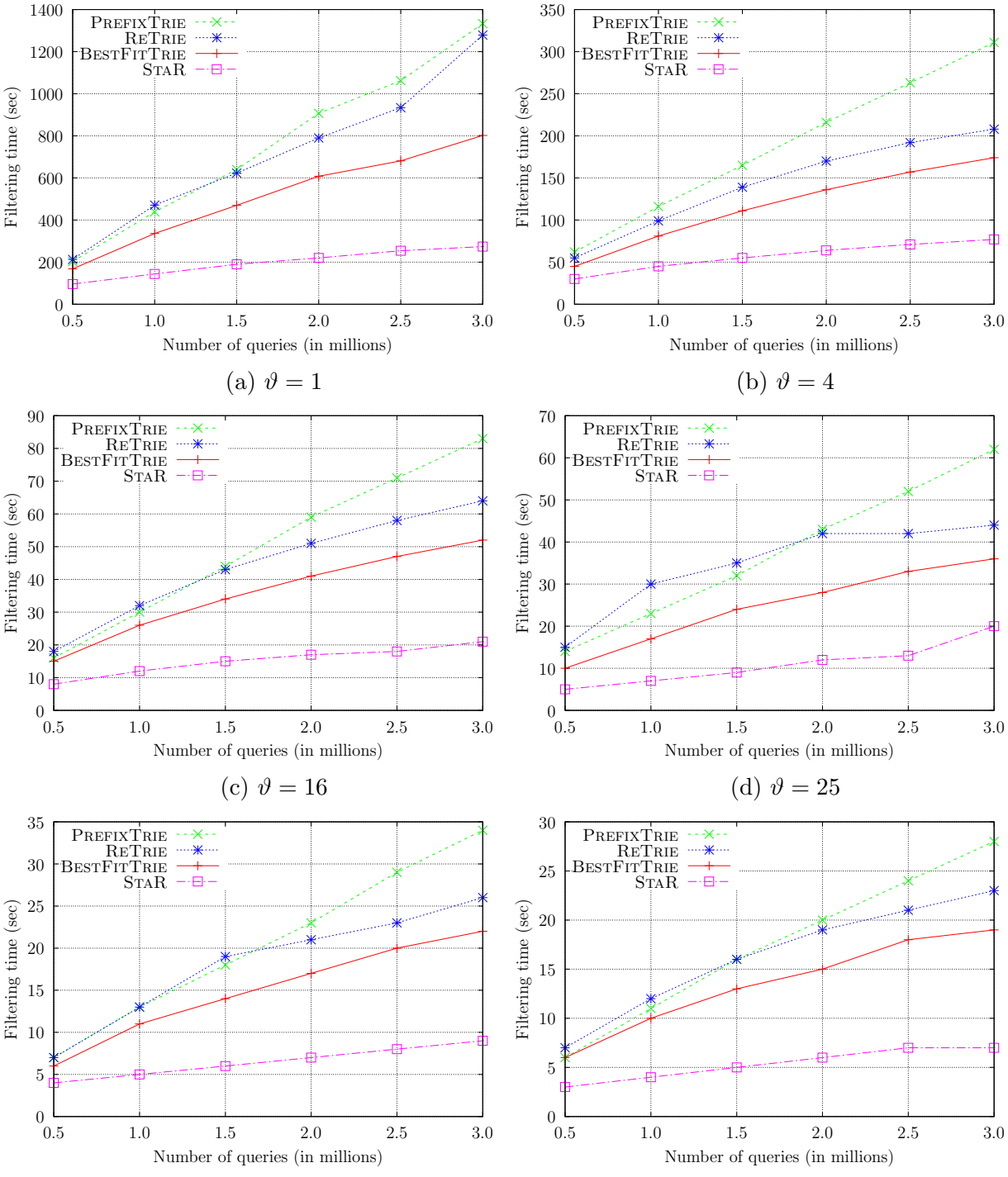
providing the best algorithms' performance.

In order to compare the filtering time of the original algorithms with their multithreading versions, we will use our results from all the algorithms variations, using the number of threads providing the lowest filtering times and the best speedup and efficiency.

Our experiment was conducted using 4, 16, 25, 50 and 80 threads. Although hardware concurrency function suggested that the number of threads supported was 64 threads (as a hint), we have found that using a number of 80 threads had the best filtering times in our system in almost all cases.

Figure 4.4 shows the filtering time of all algorithms per number of threads and Figure 4.5 shows the filtering time per algorithm when varying the number of threads ϑ ($\vartheta = 1$ refers to the serial version of the algorithm).

Table 4.5 shows how much the filtering time was improved when using 80 threads for all our algorithms. It also showcase the minimum and maximum time improvement and how many times faster (speedup) each algorithm completed the filtering process.

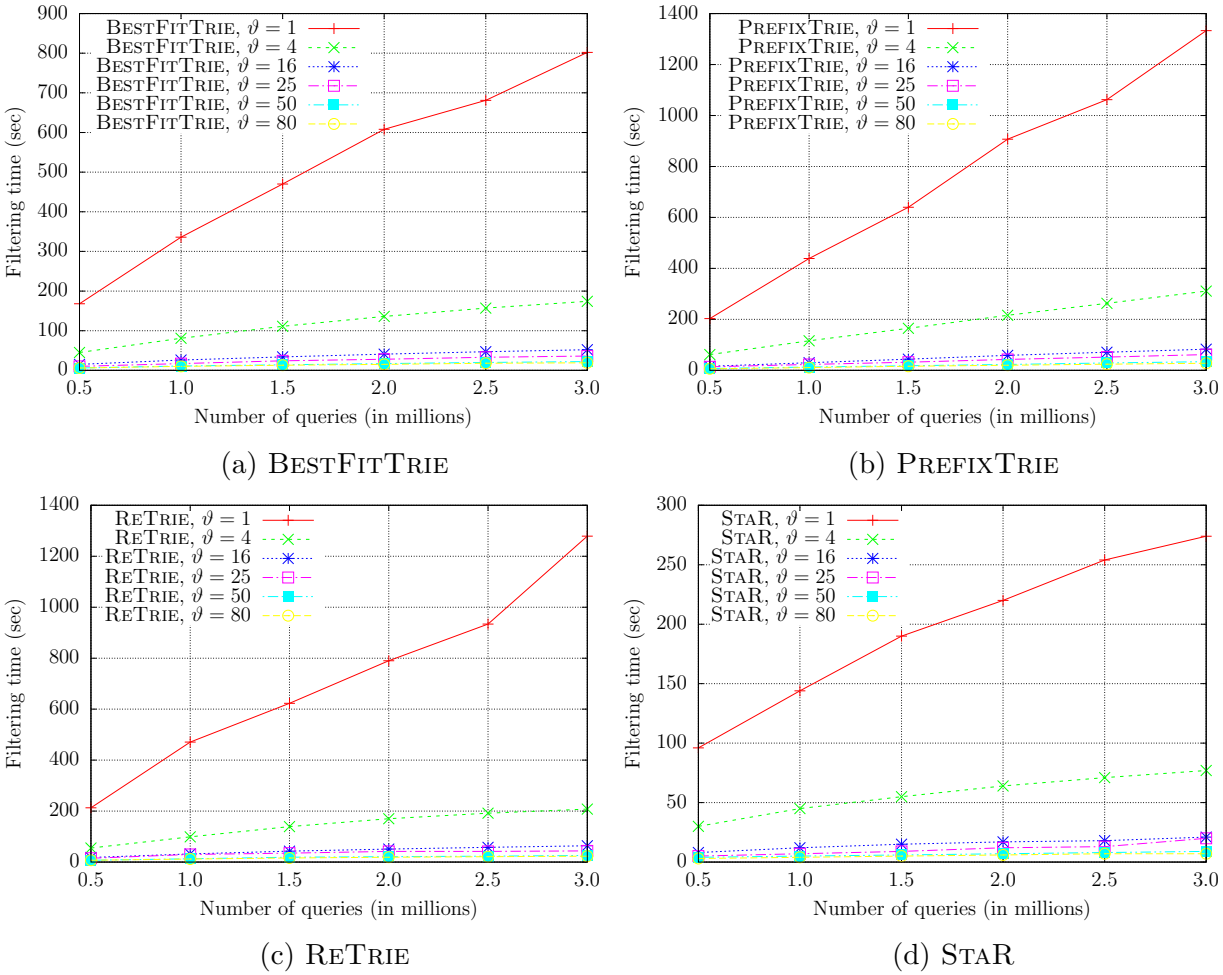


(a) $\vartheta = 1$ (b) $\vartheta = 4$

(c) $\vartheta = 16$ (d) $\vartheta = 25$

(e) $\vartheta = 50$ (f) $\vartheta = 80$

Figure 4.4: Evaluating filtering time efficiency for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$), when varying the database size from $0.5M$ to $3M$ and $I_P = 500K$. In most cases, StaR has the fastest performance, followed by BestFitTrie, ReTrie and PrefixTrie.



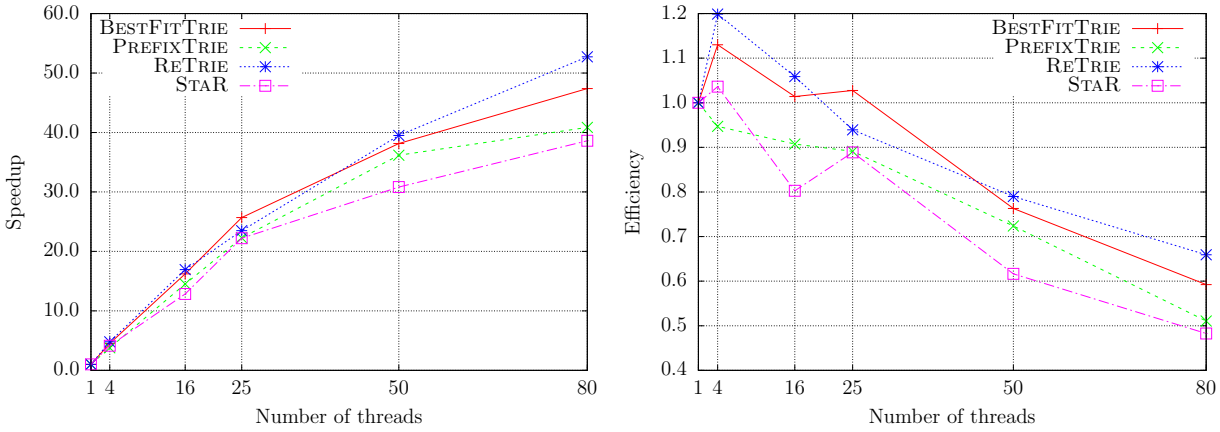
(a) BestFitTrie (b) PrefixTrie

(c) ReTrie (d) StaR

Figure 4.5: Evaluating filtering time, of all algorithms, when varying the size of query database DB and ϑ.

BestFitTrie algorithm filtering time was improved between $96,22\%$ and $97,93\%$ ($26,4$ to $48,4$ times faster execution respectively, see Figure 4.5a), compared to its serial version and PrefixTrie showed an improvement between $96,89\%$ and $97,84\%$ ($32,2$ to $46,2$ times faster, see Figure 4.5b). ReTrie algorithm filtering time was improved between $96,56\%$ and $98,28\%$ (29 to $58,1$ times faster, see Figure 4.5c). StaR algorithm filtering time was improved between $96,53\%$ and $97,41\%$ ($28,8$ to $38,6$ times faster, see Figure 4.5d).

We observe that the filtering time that all algorithms consume when using threads, is not noticeably affected as the number of queries increasing from $500K$ to 3 million. On the contrary, on the serial version of the algorithms, we can clearly see that the



(a) Comparing speedup

(b) Comparing efficiency

Figure 4.6: Evaluating speedup and efficiency for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$) in filtering, when varying the database size from 0 to $3M$ and $I_P = 3M$.

number of queries used for matching, dramatically affects the system when increased.

StaR is the algorithm with the lowest filtering time in general. Algorithms serial execution also provides the same results regarding the filtering time, due to the way the algorithms are indexing the queries and create the forest. For example, PrefixTrie uses alphabetical sorting for the queries entering the system and by using tries to store queries compactly misses clustering opportunities and thus creates a final forest that needs more time to be traversed, eventually affecting the filtering time in both scenarios.

4.5.3 Filtering speedup and efficiency

Examining the performance of the algorithms for a different number of threads, we observe that although best overall filtering times were achieved by increasing the number of threads, the efficiency of the algorithms dropped as the number of threads was increasing (Figure 4.6). In addition, the percentage of the total filtering improvement is logarithmic and constantly decreases as the number of threads increases, suggesting that maximum performance is reached for the specific system (Figure 4.7).

Table 4.6 presents the efficiency of ReTrie when $I_P = 2.5M$ for all values of ϑ. For example, ReTrie filtering time results, when using 4 threads in total, showed

Thread Performance in ReTrie			
No. of threads	Speedup(Max)	Efficiency	Time Improvement
4	6,00	150,00%	83,35%
16	27,91	174,43%	96,42%
25	34,02	136,07%	97,06%
50	52,43	104,86%	98,09%
80	58,12	72,63%	98,28%

Table 4.6: Speedup, efficiency and filtering time improvement of ReTrie for a query database of $DB = 2.5M$.

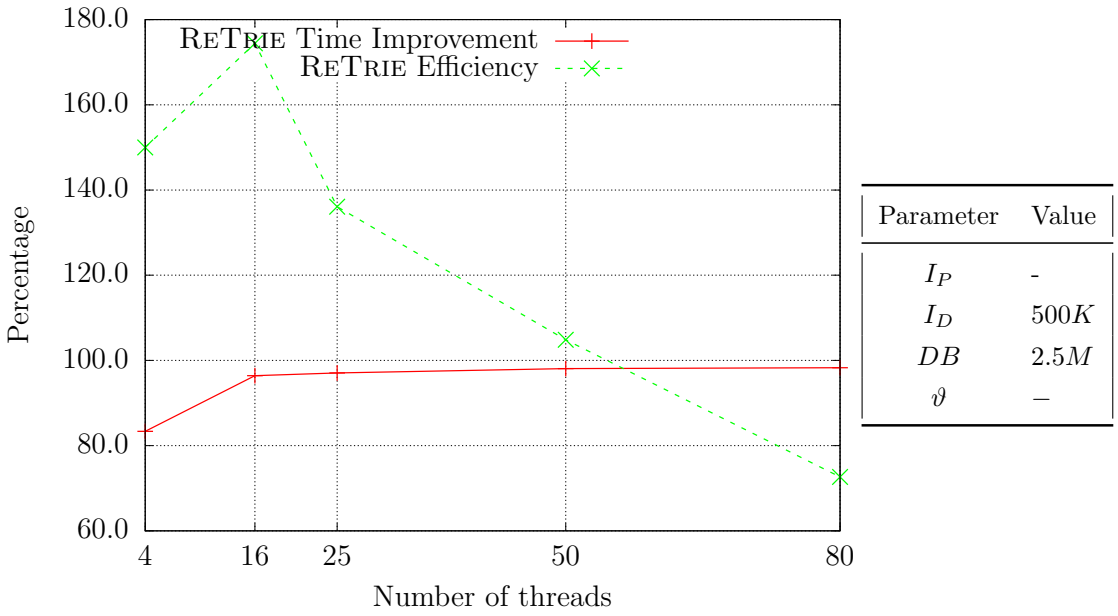


Figure 4.7: Comparing efficiency and filtering time improvement of ReTrie, when varying the number of threads ϑ, for a query database of $DB = 2.5M$.

$83,35\%$ improvement in total time but the efficiency was increased up to $150,00\%$ (by the assumption that 4 threads can ideally and theoretically complete the same task 4 times faster than 1 main thread), resulting in completing the task 6 times faster (speedup).

When efficiency is exceeding 100%, this is called *super-linear speedup*. One possible reason for super-linear speedup, in low-level computations, is the cache effect resulting from the different memory hierarchies of a modern computer. In parallel computing, not

only do the numbers of processors change, but so does the size of accumulated caches from different processors. With the larger accumulated cache size, more or even all of the working set can fit into caches and the memory access time reduces dramatically, which causes the extra speedup in addition to that from the actual computation. An analogous situation occurs when searching large data sets [12].

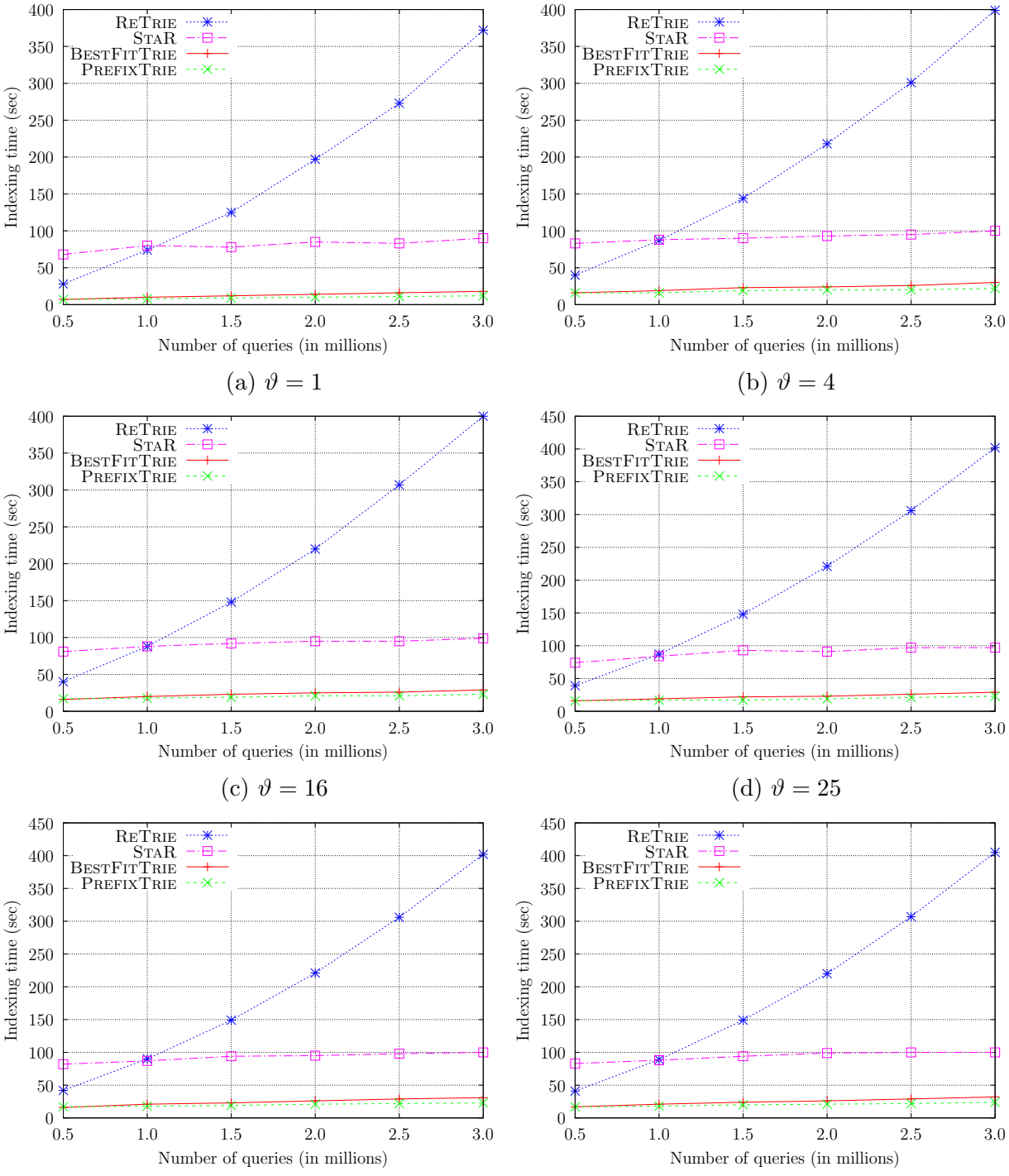
The usage of 16 threads showed 96, 42% faster filtering times and 174, 43% improvement in efficiency, resulting in 27, 91 times faster execution of the filtering process. The usage of 25 threads presented an improvement of 97, 06%, executing the filtering task 34, 02 times faster. From this point on, all our next measurements, using 50 and 80 threads, show that efficiency is declining (104, 86% and 72, 63% respectively) and the total time improvement is similar (98, 09% and 98, 28%) near 98%.

These results show that the best efficiency was achieved when using 16 threads. The use of 25 threads constitutes an intermediate solution, combining good filtering times with great efficiency, for this particular system.

4.6 Results for the Second Collection

In this section, we are going to discuss the results we received from our second collection, presenting the indexing and filtering times of the algorithms for both serial and multithreading versions.

The results from the second collection show that when the dictionary is small and the number of queries is high, there is no high variation in the time improvement between the algorithms. This is due to the high repetition of terms in the queries, leading to the creation of similar forests.

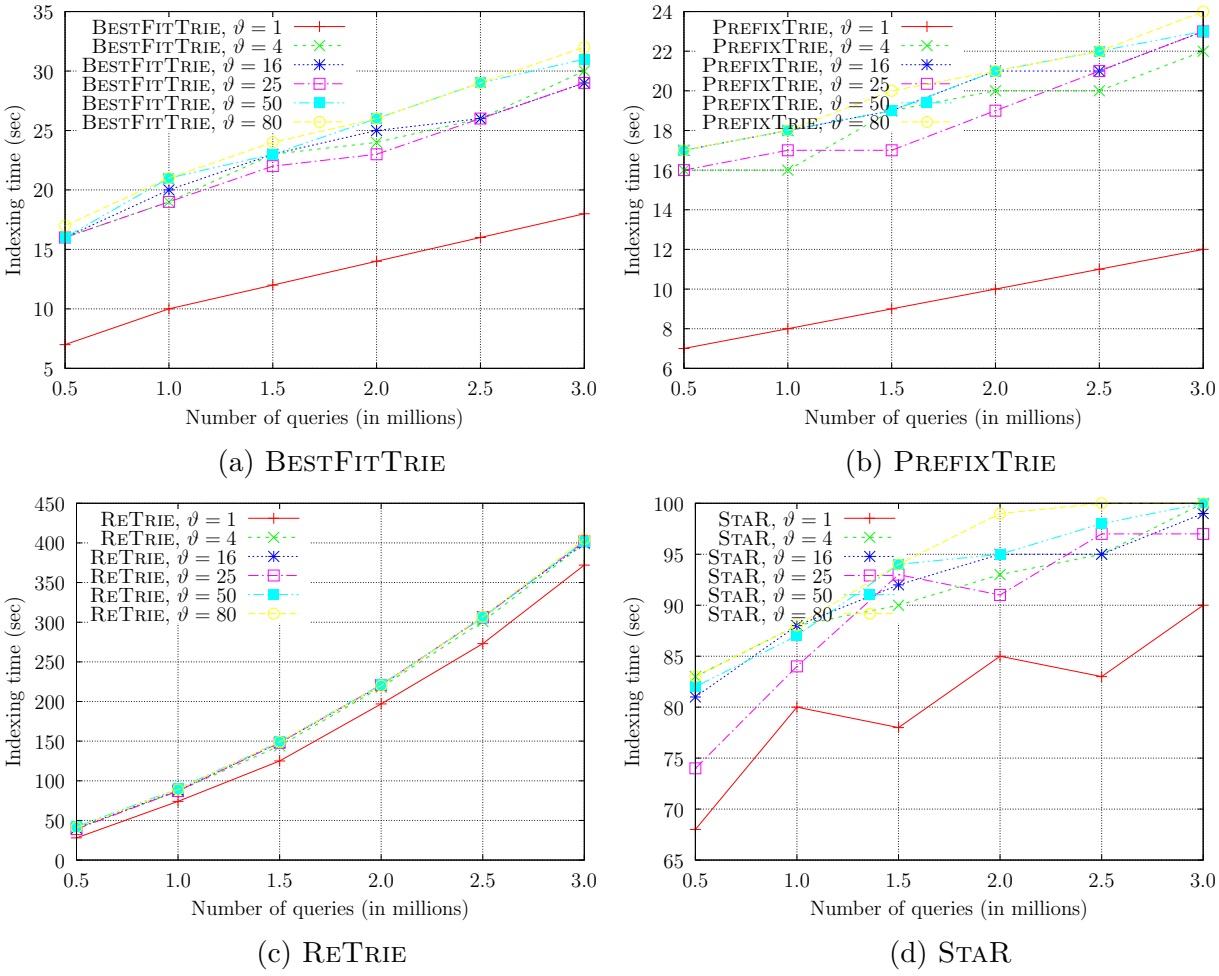


(a) $\vartheta = 1$ (b) $\vartheta = 4$

(c) $\vartheta = 16$ (d) $\vartheta = 25$

(e) $\vartheta = 50$ (f) $\vartheta = 80$

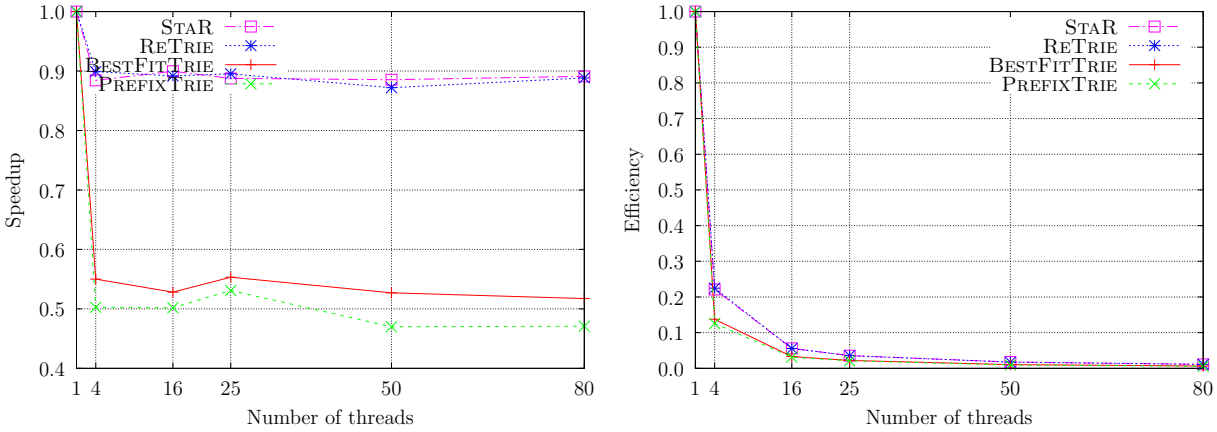
Figure 4.8: Evaluating indexing time of the *Second Collection*, for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$), when varying the database size from $0.5M$ to $3M$ and $I_P = 500K$. In all cases, PrefixTrie has the fastest performance followed by BestFitTrie. StaR and ReTrie are the slowest.



(a) BestFitTrie (b) PrefixTrie

(c) ReTrie (d) StaR

Figure 4.9: Evaluating indexing time for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$) of the *Second Collection*, when varying the database size from $0.5M$ to $3M$ and $I_P = 500K$. In all cases, ($\vartheta = 1$) has the fastest performance.



(a) Comparing speedup

(b) Comparing efficiency

Figure 4.10: Comparing speedup and efficiency of indexing of the *Second Collection*, for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$), when varying the database size from 0 to $3M$ and $I_P = 3M$.

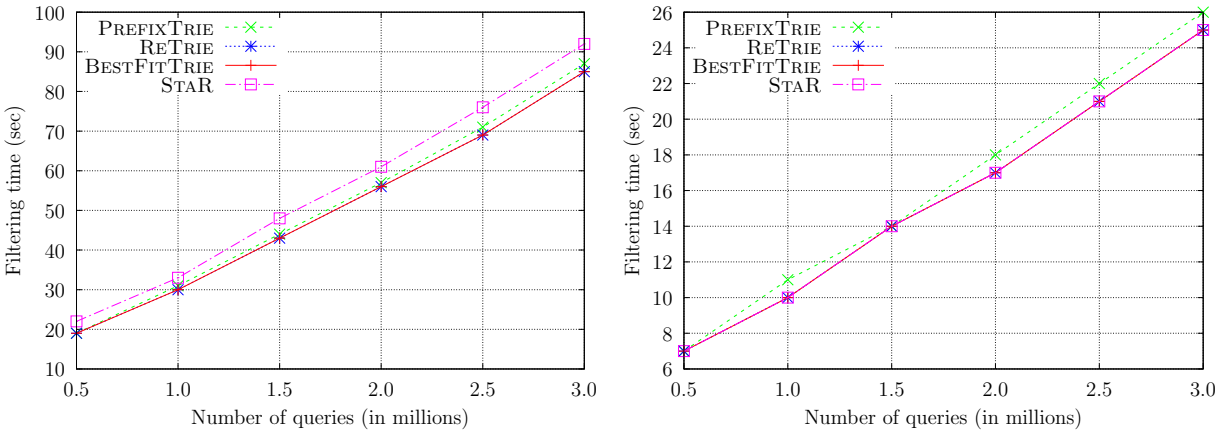
4.6.1 Evaluating indexing time

Despite the number of threads used, all algorithms took more time to index the queries. ReTrie was the slowest, followed by StaR, BestFitTrie and PrefixTrie. Comparing indexing time for several number of threads, we find that especially BestFitTrie and PrefixTrie took from 64% to 112% more time, depending on the number of threads used, compared to their serial versions. ReTrie and StaR took from 7% to 20% more time, compared to their serial versions.

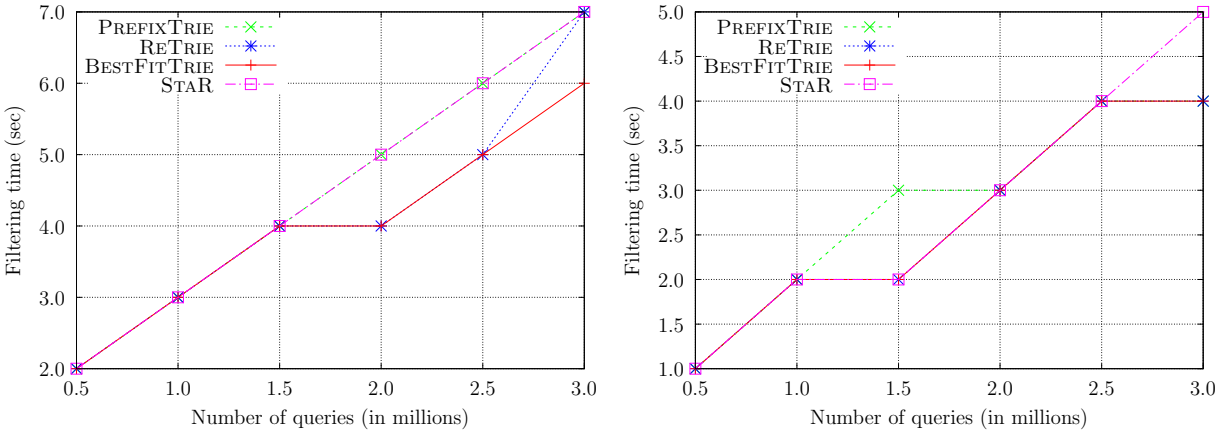
In Figure 4.8 we can see all the algorithms indexing times per number of threads used. In Figure 4.9 we can see all algorithms indexing times, compared to their multi-threading versions when indexing 3 million queries at once.

Speedup and efficiency of the indexing process of the *Second Collection*, is presented in Figure 4.10. In Figure 4.10a we can see that StaR and ReTrie have the best speedup, followed by BestFitTrie and PrefixTrie. Efficiency, as presented in Figure 4.10b, is similar for all algorithms and drops dramatically as the number of threads increases.

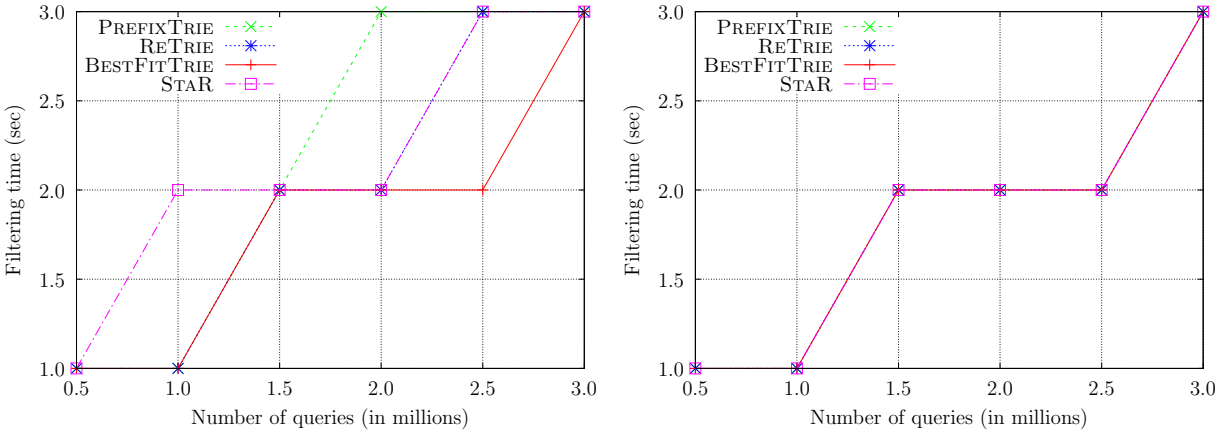
4.6.2 Evaluating filtering time



(a) $\vartheta = 1$ (b) $\vartheta = 4$

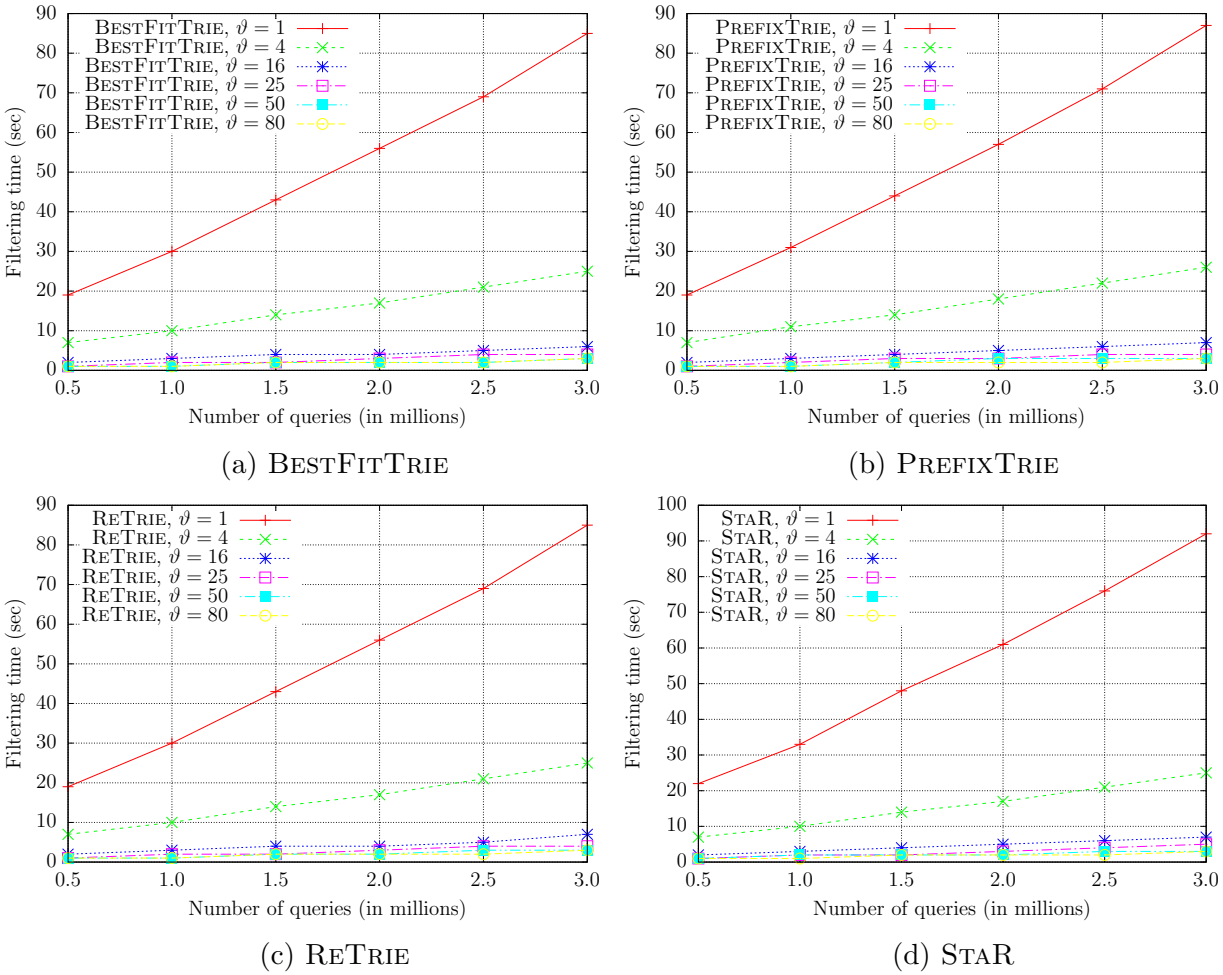


(c) $\vartheta = 16$ (d) $\vartheta = 25$



(e) $\vartheta = 50$ (f) $\vartheta = 80$

Figure 4.11: Evaluating filtering time efficiency for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$) of the *Second Collection*, when varying the database size from $0.5M$ to $3M$ and $I_P = 500K$. The performance of all algorithms is similar.



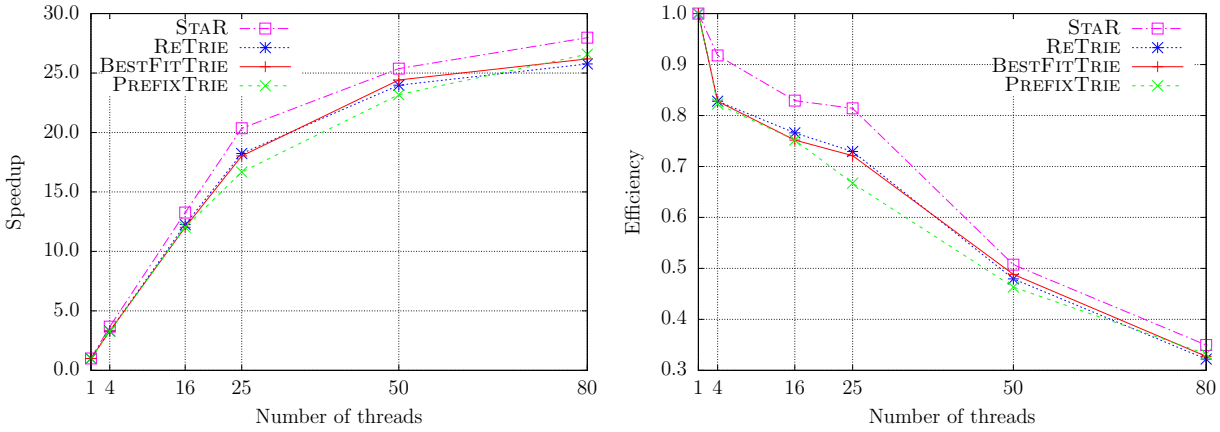
(a) BestFitTrie (b) PrefixTrie

(c) ReTrie (d) StaR

Figure 4.12: Evaluating filtering time, of all algorithms the *Second Collection*, when varying the size of query database DB and ϑ.

The filtering time needed was drastically decreased on all algorithms. The filtering time of the algorithms showed an improvement between $91,78\%$ to $96,48\%$. The only exception was when the algorithms used 4 threads. Then the filtering time improved from $70,09\%$ to $72,76\%$ on all algorithms. We also noticed that filtering performance was almost identical regardless of the algorithms used, with BestFitTrie and ReTrie completing the process slightly faster in most cases. Figure 4.11 presents all algorithms filtering time efficiency per number of threads. In Figure 4.12 we can see the comparison of filtering times when varying ϑ per algorithm. Figure 4.13 presents filtering speedup and efficiency of all algorithms.

Table 4.7 shows how much the filtering time was improved when using 80 threads



(a) Comparing speedup

(b) Comparing efficiency

Figure 4.13: Evaluating speedup and efficiency for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$) of the *Second Collection*, in filtering, when varying the database size from 0 to $3M$ and $I_P = 3M$.

Filtering Times Improvement - 80 Threads				
Algorithm	Min Improvement	Max Improvement	Speedup (Min)	Speedup (Max)
BestFitTrie	92,94%	96,24%	14,17	26,58
PrefixTrie	92,74%	96,24%	13,77	26,56
ReTrie	93,00%	96,30%	14,29	27,06
StaR	94,11%	96,48%	16,98	28,44

Table 4.7: Filtering time improvement and speedup for all algorithms - *Second Collection*

for all our algorithms, using the second query collection.

4.6.3 Filtering speedup and efficiency

Examining the speedup and efficiency, we observe that the efficiency is similar to the first collection. Efficiency is also declining as the total number of threads is increasing. Table 4.8 and Figure 4.14 present the efficiency and time improvement of algorithm StaR when varying ϑ.

These results show that the best efficiency was achieved when using 4 threads. All the algorithms had similarly decreased their performance when using more than 4

Thread Performance in StaR			
No. of threads	Speedup(Max)	Efficiency	Time Improvement
4	3,67	91,79%	72,76%
16	13,27	82,92%	92,47%
25	20,35	81,40%	95,08%
50	25,37	50,74%	96,06%
80	27,98	34,97%	96,43%

Table 4.8: Thread performance and filtering time efficiency of StaR, for a query database of $DB = 3M$ - *Second Collection*.

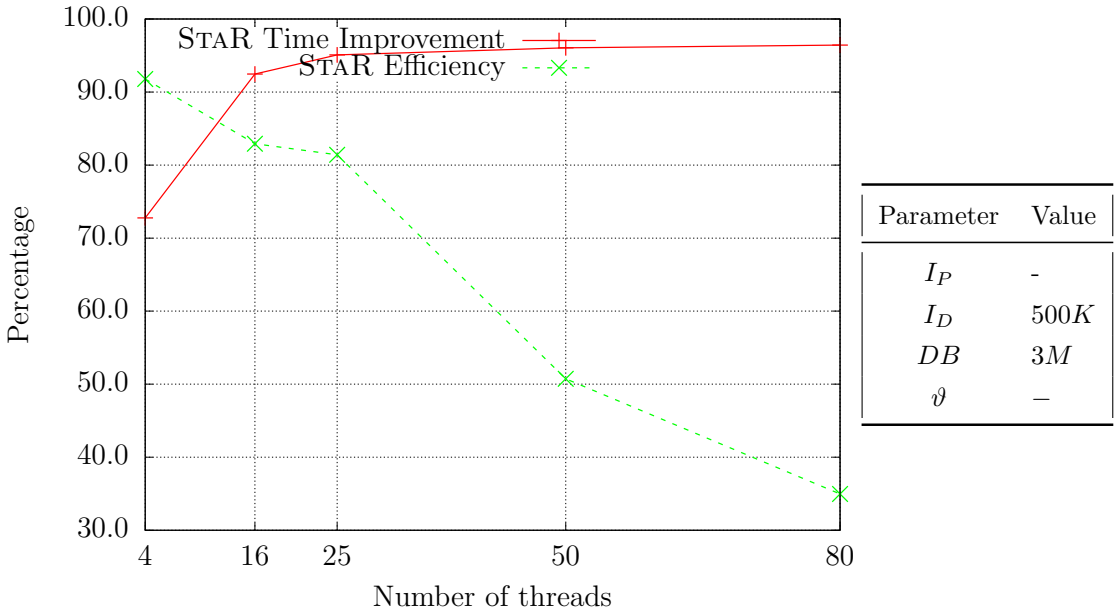


Figure 4.14: Comparing thread performance and filtering time efficiency of StaR, when varying the number of threads ϑ, for a query database of $DB = 3M$ of the *Second Collection*.

threads. The percentage of the total filtering time is also logarithmic and constantly decreases as the number of threads increases (Figure 4.14).

4.7 Results for the Neural Networks Collection

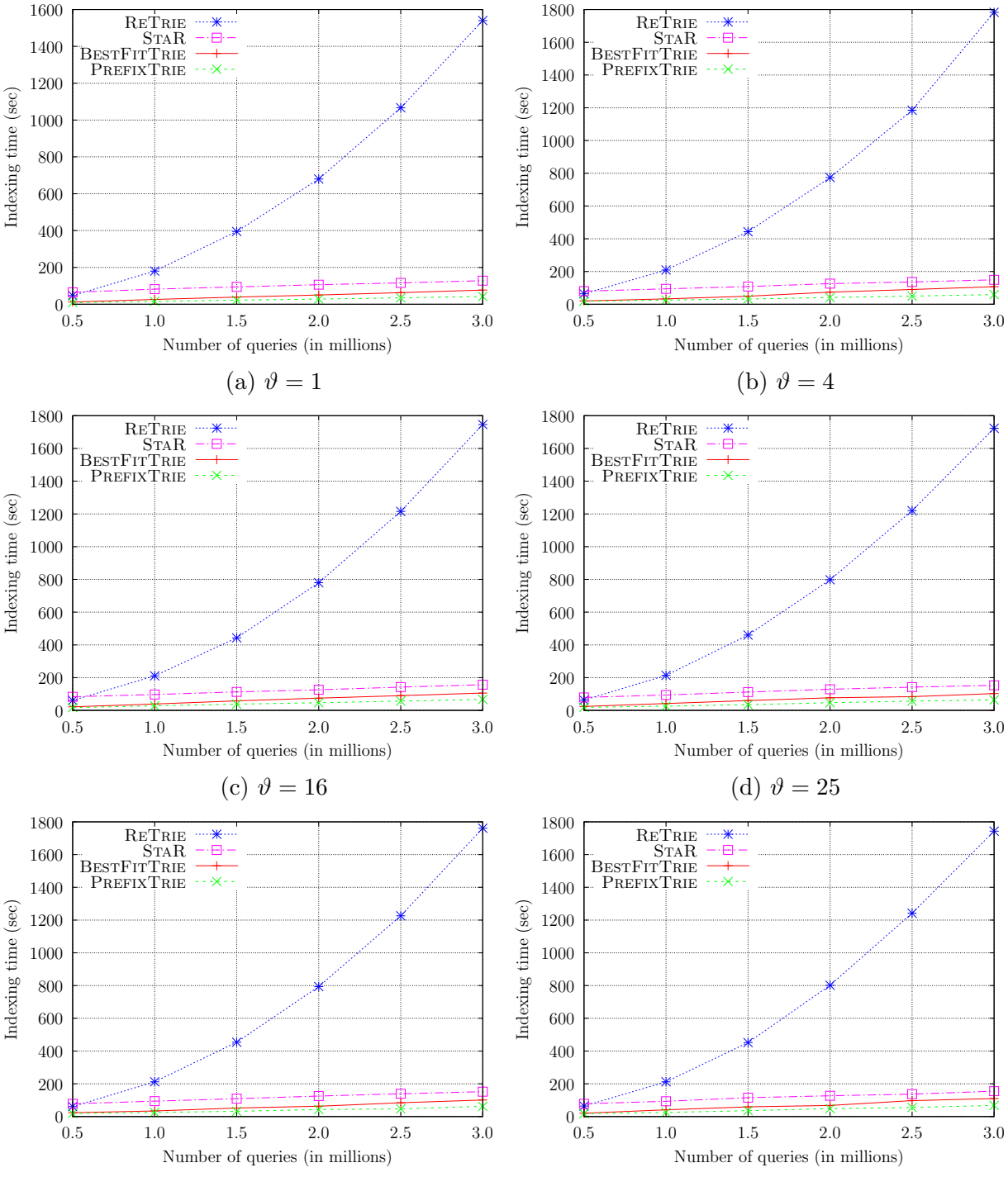
In this section, we are going to discuss the results we received from our third collection, presenting the indexing and filtering times of the algorithms for both serial and multi-threading versions. This is a focused data set about the same topic, against a collection of queries with a small vocabulary and a lot of noise.

4.7.1 Evaluating indexing time

ReTrie was again the algorithm that took the most time in indexing, followed by StaR, BestFitTrie and PrefixTrie. However, BestFitTrie and PrefixTrie where the algorithms having the worst performance in indexing, when compared to their serial versions. Both algorithms were slower than their serial version, taking from $20,49\%$ to $46,16\%$ more time to index the queries. On the contrary, ReTrie performed close to its serial version and one of the variations ($I_P = 1M$ and $\vartheta = 4$) presented an improvement of $3,23\%$. StaR was the algorithm with the best indexing times performing from $-2,57\%$ to $3,80\%$ in comparison with its serial version.

In Figure 4.15 we can see all the algorithms indexing times per number of threads used. In Figure 4.16 we can see all algorithms indexing times, compared to their multithreading versions when indexing 3 million queries at once.

Speedup and efficiency of the indexing process of the *Neural Networks Collection*, is presented in Figure 4.17. In Figure 4.17a we can see that StaR and ReTrie have the best speedup, followed by BestFitTrie and PrefixTrie. Efficiency is similar for all algorithms, as presented in Figure 4.17b and dramatically decreases as the number of threads increases.

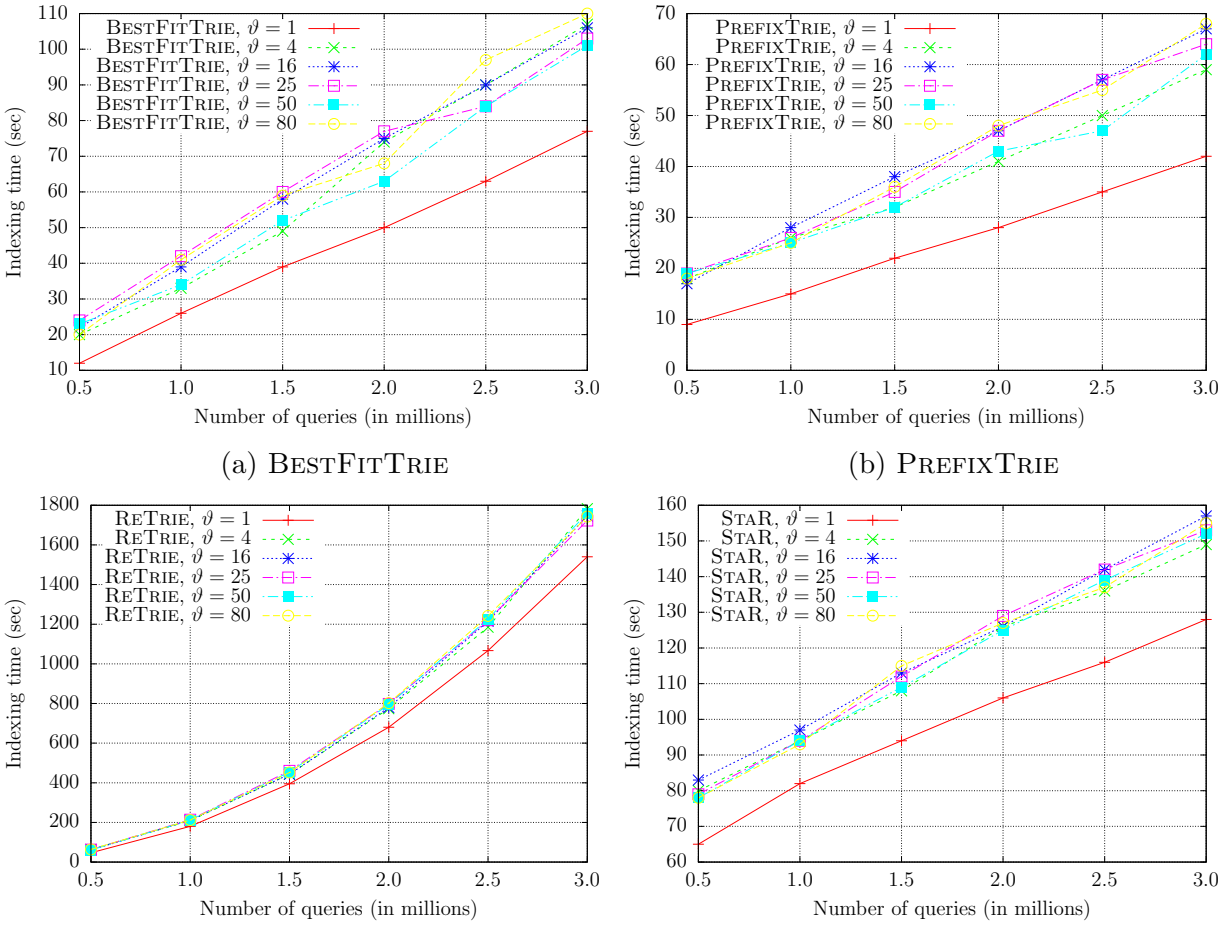


(a) $\vartheta = 1$ (b) $\vartheta = 4$

(c) $\vartheta = 16$ (d) $\vartheta = 25$

(e) $\vartheta = 50$ (f) $\vartheta = 80$

Figure 4.15: Evaluating indexing time of the *Neural Networks Collection*, for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$), when varying the database size from $0.5M$ to $3M$ and $I_P = 500K$. In all cases, PrefixTrie has the fastest performance followed by BestFitTrie. StaR and ReTrie are the slowest.



(a) BestFitTrie (b) PrefixTrie

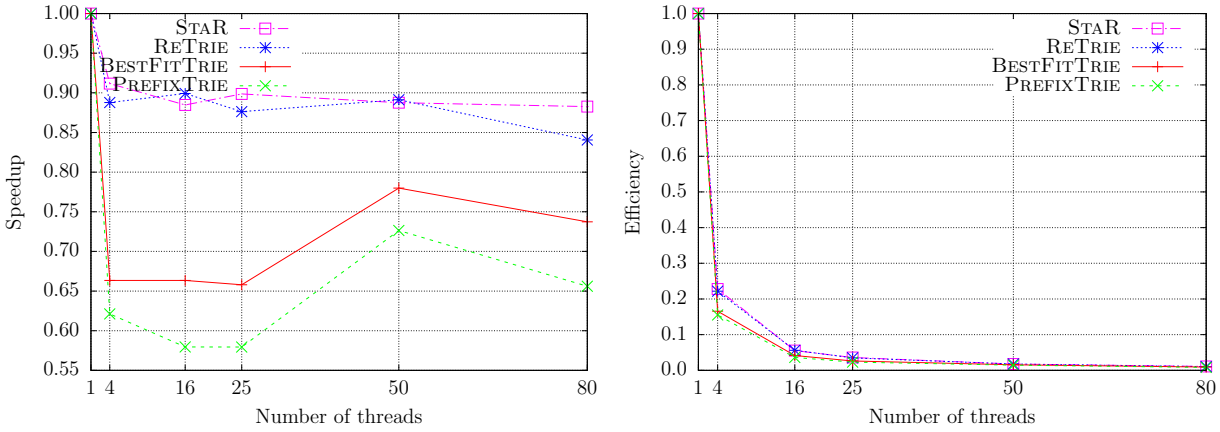
(c) ReTrie (d) StaR

Figure 4.16: Evaluating indexing time for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$) of the *Neural Networks Collection*, when varying the database size from $0.5M$ to $3M$ and $I_P = 500K$. In all cases, a single thread ($\vartheta = 1$) has the fastest performance.

4.7.2 Evaluating filtering time

The performance of the algorithms in the filtering process, showed a similar time improvement When using 4 threads, all the algorithms minimized the time needed $71,50\%$ in average. While the number of threads increased, the filtering time needed decreased between $91,37\%$ and $98,19\%$ for all algorithms. Finally, ReTrie was the algorithm that completed, almost always, the filtering process faster than the other algorithms due to the nature of this collection.

Figure 4.18 presents all algorithms filtering time efficiency per number of threads. In Figure 4.19 we can see the comparison of filtering times when varying ϑ per algorithm.



(a) Comparing speedup

(b) Comparing efficiency

Figure 4.17: Comparing speedup and efficiency of indexing of the *Neural Networks Collection*, for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$), when varying the database size from 0 to $3M$ and $I_P = 3M$.

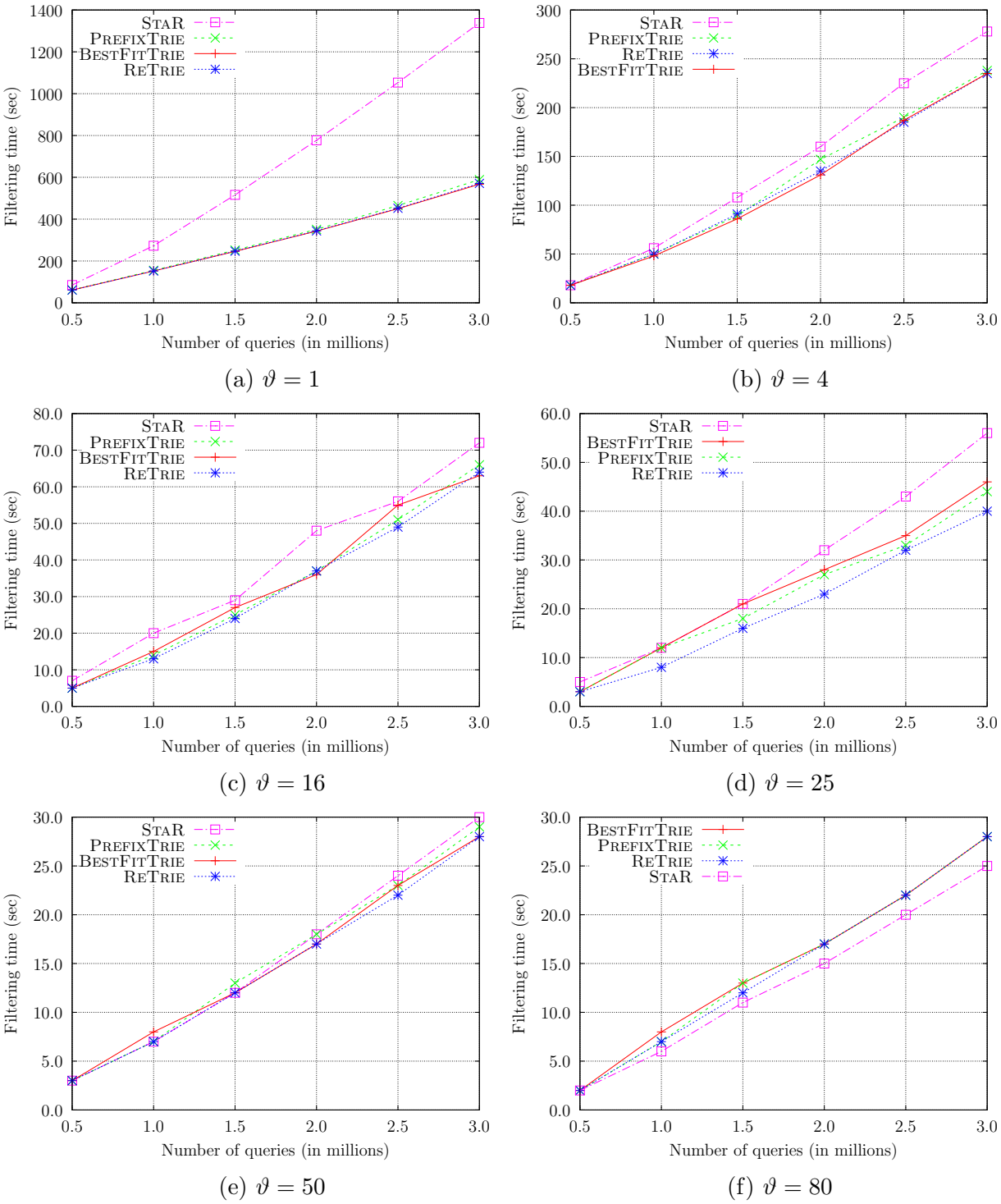
Filtering Times Improvement - 80 Threads				
Algorithm	Min Improvement	Max Improvement	Speedup (Min)	Speedup (Max)
BestFitTrie	94,69%	95,84%	18,84	24,04
PrefixTrie	94,72%	95,81%	18,95	23,87
ReTrie	94,86%	95,85%	19,44	24,12
StaR	97,00%	98,19%	33,34	55,18

Table 4.9: Filtering Time Improvement - *Neural Networks Collection*

Figure 4.20 presents filtering speedup and efficiency of all algorithms.

Table 4.9 shows how much the filtering time was improved when using 80 threads for all our algorithms, using the *Neural Networks Collection*.

4.7.3 Filtering speedup and efficiency

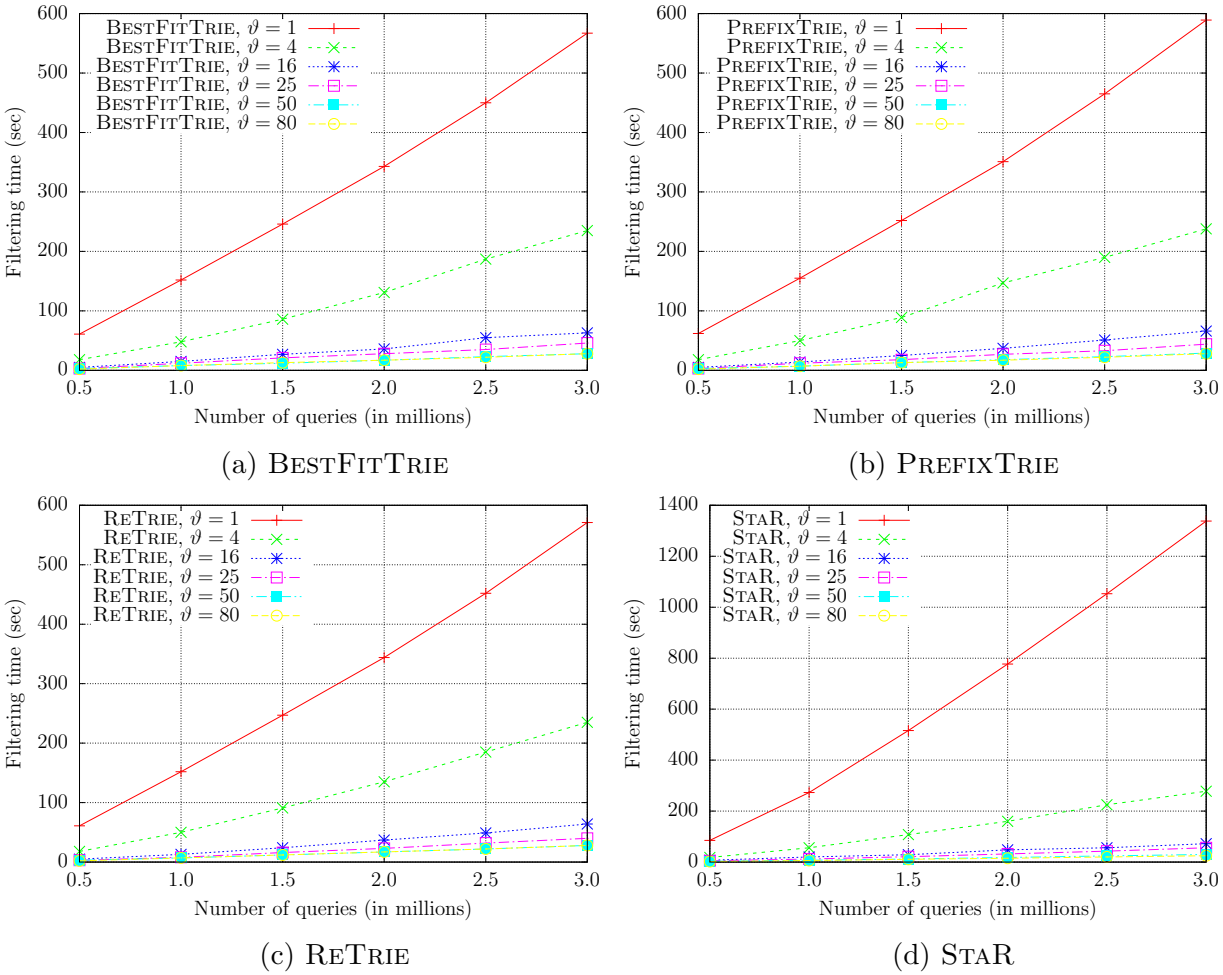


(a) $\vartheta = 1$ (b) $\vartheta = 4$

(c) $\vartheta = 16$ (d) $\vartheta = 25$

(e) $\vartheta = 50$ (f) $\vartheta = 80$

Figure 4.18: Evaluating filtering time efficiency for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$) of the *Neural Networks Collection*, when varying the database size from $0.5M$ to $3M$ and $I_P = 500K$. The performance of all algorithms is similar.



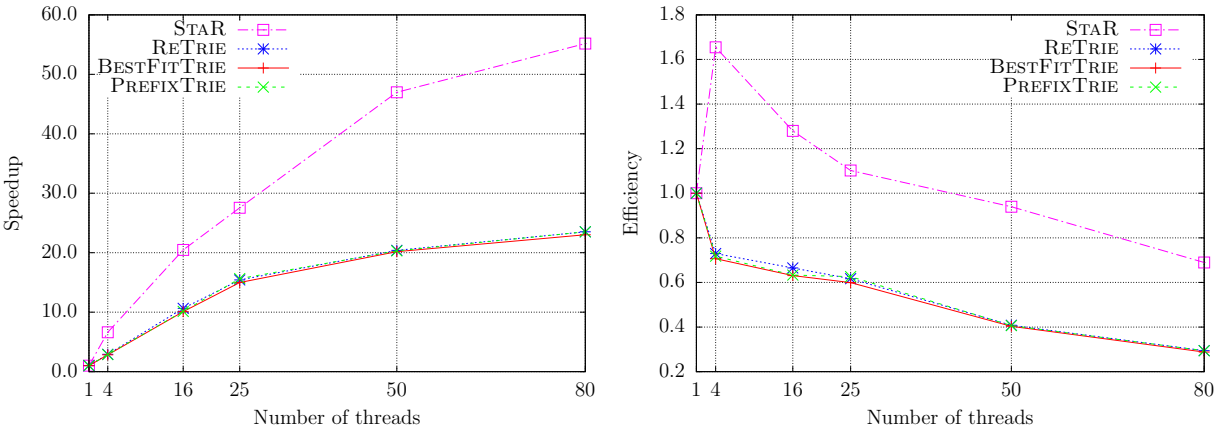
(a) BestFitTrie (b) PrefixTrie

(c) ReTrie (d) StaR

Figure 4.19: Evaluating filtering time, of all algorithms the *Neural Networks Collection*, when varying the size of query database DB and ϑ.

The best speedup and efficiency was achieved when using 4 threads. All the algorithms, except StaR, had a similar increase in speedup and efficiency. StaR was the algorithm with remarkably better speedup and efficiency in all cases (Figure 4.21). The rest of the algorithms had a similar efficiency between 70% to 73% when using 4 threads. Also, efficiency drops significantly as the number of threads increases.

Table 4.10 and Figure 4.21 present the efficiency and time improvement of algorithm StaR when varying ϑ.



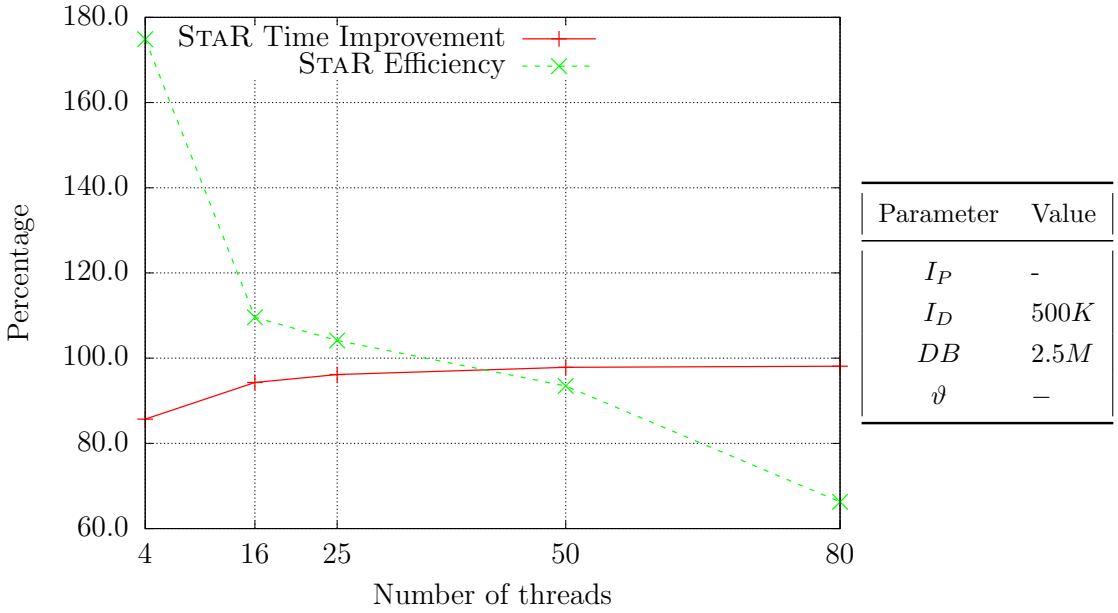
(a) Comparing speedup

(b) Comparing efficiency

Figure 4.20: Evaluating speedup and efficiency for several number of threads ($\vartheta \in \{1, 4, 16, 25, 50, 80\}$) of the *Neural Networks Collection*, in filtering, when varying the database size from 0 to $3M$ and $I_P = 3M$.

Speedup and Efficiency in StaR			
No. of threads	Speedup (Max)	Efficiency	Time Improvement
4	6,99	174,87%	85,70%
16	17,53	109,60%	94,30%
25	26,04	104,16%	96,16%
50	46,74	93,48%	97,86%
80	53,01	66,26%	98,11%

Table 4.10: Thread performance and filtering time efficiency of StaR, for a query database of $DB = 2.5M$ - *Neural Networks Collection*.



Parameter	Value
I_P	-
I_D	$500K$
DB	$2.5M$
ϑ	–

Figure 4.21: Comparing thread performance and filtering time efficiency of STAR, when varying the number of threads ϑ, for a query database of $DB = 2.5M$ of the *Neural Networks Collection*.

Chapter 5

Conclusions

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

5.1 Summary

In this thesis, we studied the problem of information filtering, given a database of user queries, in a publish/subscribe system.

In this publish/subscribe system, users can be enrolled in a server, expressing their interests in form of queries. When a document is published in the system, all the continuous queries that match the incoming document are located through the matching process and all the appropriate clients are notified.

By using parallelization techniques and by extending indexing algorithms, our goal is to effectively solve the problem of the *information filtering parallelization* in each individual server. The indexing algorithms were presented in Chapter 3. In Chapter 4 we evaluated our algorithms performance by comparing their serial and multithreading form/variation.

Due to its nature, the process of indexing the queries seemed to luck any noticeable improvement in decreasing the time needed to complete its operation. On the other hand, filtering time was reduced significantly in all our algorithms, in proportion with

the way each algorithm structures its forest of tries. Finally, we investigated the usage of multi threaded solutions to enhance the filtering performance of the algorithms BestFitTrie, PrefixTrie,ReTrie and StaR. We identified and assessed one proof-of-concept parallelization scenario, in order to increase the filtering performance of the algorithms.

5.2 Future directions

Interesting directions for future research involve *(i)* the adaptation of automata/graph-based techniques as in [59, 66] to Boolean information filtering and their comparison against trie-based approaches, *(ii)* the construction of information filtering ontology systems that will be able to filter ontology data in a streaming fashion.

Bibliography

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on very large databases*, pages 490–501, 1994.

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

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

[4] A. Algarni, Y. Li, Y. Xu, and R. Lau. An effective model of using negative relevance feedback for information filtering. *In CIKM*, pages 1605–1608, 2009.

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

[6] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th VLDB Conference*, 2000.

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

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

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

[10] T. Bell, J. Cleary, and I. Witten. *Text Compression.* PrenticeHall Publishers, 1990.

[11] T. Bell and A. Moffat. The Design of a High Performance Information Filtering System. In *Proceedings of the ACM SIGIR*, pages 12–20, 1996.

[12] J. Benzi and M. Damodaran. *Parallel Computational Fluid Dynamics 2007*, volume Parallel Three Dimensional Direct Simulation Monte Carlo for Simulating Micro Flows. Springer, Berlin, 2009.

[13] J. Callan. Document Filtering With Inference Networks. In *Proceedings of the ACM SIGIR*, 1996.

[14] J. Callan. Learning While Filtering Documents. In *Proceedings of the ACM SIGIR*, pages 224–231, 1998.

[15] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient Filtering in Publish Subscribe Systems Using Binary Decision Diagrams. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 443–452, Los Alamitos, California, 2001.

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

[17] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the 18th International Conference on Data Engineering*, pages 235–244, February 2002.

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

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

[20] T. Chantzios, L. Zervakis, S. Skiadopoulos, and C. Tryfonopoulos. Ping - a customizable, open-source information filtering system for textual data. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, (DEBS)*, Darmstadt, Germany, June 2019.

[21] R. Choi and R. Wong. Efficient xquery join processing in publish/subscribe systems. *In ADC*, pages 97–106, 2009.

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

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

[24] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-peer Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 23–24, Vienna, Austria, July 2002.

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

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

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

[28] Y. Diao, M. Altinel, M. F. H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-performance XML Filtering. *ACM Transactions on Database Systems*, 28(4):467–516, 2003.

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

[30] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of ACM SIGMOD*, 2001.

[31] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen. Parallel event processing for content-based publish/subscribe systems. *In DEBS*, 2009.

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

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

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

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

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

[37] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, pages 173–189, Siena,Italy, January 2003.

[38] E. Grummt. Fine-grained parallel xml filtering for content-based publish/subscribe systems. *In DEBS*, pages 219–228, 2011.

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

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

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

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

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

[44] M. Koubarakis, T. Koutris, C. Tryfonopoulos, and P. Raftopoulou. Information Alert in Distributed Digital Libraries: The Models, Languages, and Architecture of DIAS. In *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*, pages 527–542, Rome, Italy, September 2002.

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

[46] Y. Li, A. Algarni, S.-T. Wu, and Y. Xue. Mining negative relevance feedback for information filtering. *In Web Intelligence*, pages 606–613, 2009.

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

[48] Y. Li, X. Zhou, P. Bruza, Y. Xu, and R. Lau. A two-stage text mining model for information filtering. *In CIKM*, pages 1023–1032, 2008.

[49] S. Lippman, J. Lajoie, and B. Moo. *C++ Primer.* Addison-Wesley, 5th edition, 2013.

[50] J. Lu, X. Meng, and T. Ling. Indexing and querying xml using extended dewey labeling scheme. *Data Knowl. Eng.*, 70(1):35–59, 2011.

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

[52] R. Malaga. *Advances in Computer.*, volume 78:Search Engine Optimization—Black and White Hat Approaches. Elsevier Inc., New Jersey, USA, 2010.

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

[54] M. Mitchell, J. Oldham, and A. Samuel. *Advanced Linux Programming.* David Dwyer, 2001.

[55] M. Morita and Y. Shinoda. Information Filtering Based on User Behaviour Analysis and Best Match Text Retrieval. In *Proceedings of the ACM SIGIR*, pages 272–281, 1994.

[56] N. Nanas, S. Kodovas, and M. Vavalis. Revisiting evolutionary information filtering. *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[57] N. Nanas, S. Kodovas, M. Vavalis, and E. Houstis. Immune Inspired Information Filtering in a High Dimensional Space. In *Proceedings of the 9th International Conference on Artificial Immune Systems*, pages 47–60, Edinburgh, UK, July 2010.

[58] N. Nanas and M. Vavalis. A "bag" or a "window"of words for information filtering?. *In SETN*, pages 182–193, 2008.

[59] N. Nanas, M. Vavalis, and A. N. D. Roeck. A network-based model for high-dimensional information filtering. In *Proceeding of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2010, Geneva, Switzerland, July 19-23, 2010*, pages 202–209, 2010.

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

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

[62] M.-J. Park and C.-W. Chung. ibroker: An intelligent broker for ontology based publish/subscribe systems. *In ICDE*, pages 1255–1258, 2009.

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

[64] J. Qian, J. Yin, and J. Dong. Parallel matching algorithms of publish/subscribe system. *In ITNG*, pages 638–643, 2011.

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

[66] W. Rao, L. Chen, S. Chen, and S. Tarkoma. Evaluating continuous top-k queries over document streams. *World Wide Web*, 17(1):59–83, 2014.

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

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

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

[70] C. Tryfonopoulos, M. Koubarakis, and Y. Drougas. Filtering Algorithms for Information Retrieval Models with Named Attributes and Proximity Operators. In

Proceedings of the 27th Annual International ACM SIGIR Conference, pages 313–320, Sheffield, UK, July 2004.

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

[72] W. Vanderbauwhede, L. Azzopardi, and M. Moadeli. Fpga-accelerated information retrieval: High-efficiency document filtering. *In FPL*, pages 417–422, 2009.

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

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

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

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

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

[78] L. Zervakis. Profile Indexing Algorithms in Publish/Subscribe Systems. Bachelor thesis, Department of Informatics and Telecommunications, University of Peloponnese, Tripoli, Greece, 2011.

[79] L. Zervakis, C. Tryfonopoulos, S. Skiadopoulos, and M. Koubarakis. Full-text support for publish/subscribe ontology systems. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, Crete, Greece, May 2016.

[80] L. Zervakis, C. Tryfonopoulos, S. Skiadopoulos, and M. Koubarakis. Query reorganization algorithms for efficient boolean information filtering. *IEEE Trans. Knowl. Data Eng.*, 29(2):418–432, 2017.

[81] B. Zhang, J. Pan, J. Hu, Z. Liu, and R. Zhang. A case study on chinese text information filtering method based on user ontology model. *In ICYCS*, pages 1695–1700, 2008.

[82] Y. Zhang and J. Callan. Maximum likelihood estimation for filtering thresholds. In *Proceedings of the ACM SIGIR*, 2001.

[83] X. Zhou, Y. Li, P. Bruza, Y. Xu, and R. Lau. Rough sets based reasoning and pattern mining for a two-stage information filtering system. *In CIKM*, pages 1429–1432, 2010.

[84] X. Zhou, Y. Li, P. Bruza, Y. Xu, and R. Lau. Pattern mining for a two-stage information filtering system. *In PAKDD (1)*, pages 363–374, 2011.