Efficient Profile Indexing Algorithms for Information Dissemination

by

Yannis Drougas

A thesis submitted in fulfillment of the requirements for Diploma degree in Electronic and Computer Engineering

TECHNICAL UNIVERSITY OF CRETE
DEPARTMENT OF ELECTRONIC AND COMPUTER ENGINEERING
INTELLIGENT SYSTEMS LABORATORY

July, 2003

Abstract

The quantity of information in the web is huge and rapidly increasing. People that want to use this information have to cope with the problem of information overload. As a result, information querying and retrieval relies on search engines and other specialized systems designed for this task. However, the user still has to spend much time in order to seek for the information he (or she) is interested in and to filter out unwanted information. A solution to this problem is the use of selective dissemination of information. The idea of selective information dissemination is that users express their desire and preferences for information by posting profiles (or long-standing queries) to a computer system. The system informs the user about any incoming information matching his (or her) profile.

This dissertation deals with the problem of the dissemination of textual information. More specifically, we study the data models and query languages suitable for such an application. Next, we propose some main memory algorithms that support these languages. We implement these algorithms and calculate their space and time complexities. Finally, we end up with their experimental evaluation.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Manolis Koubarakis, Thodoris Koutris and Christos Tryfonopoulos for providing me with material covering the exact definitions of the data models used in this dissertation. This material has been used verbatim in Chapter 3.

Chapter 1

Introduction

The quantity of information in the web is huge and rapidly increasing. People that want to use this information have to cope with the problem of information overload. As a result, information querying and retrieval relies on search engines and other specialized systems designed for this task. However, the user still has to spend much time in order to seek for the information he (or she) is interested in and to filter out unwanted information. A solution to this problem is the use of selective dissemination of information. The idea of selective information dissemination is that users express their desire and preferences for information by posting profiles (or long-standing queries) to a computer system. The system then, informs the user about any incoming information matching his (or her) profile.

1.1 Overview

Dissemination of selected information is a field of more and more scientific research during the last years. This dissertation implements a part of a *peer-to-peer (P2P)* system like the one presented in Figure 1.1. We consider a P2P network of *middle agents*, storing user profiles that represent user needs for information. End agents connect with the middle agent network, to submit either information or a profile. By submitting a profile, the user requests the system to send to him any document that complies ("matches") with the rules declared through some appropriate language in the profile, as soon as the document is published to the system.

In this dissertation, we propose some algorithms and data structures appropriate to be used by any of the middle agents to determine the elements of a set of profiles that match an incoming document. Specifically, we implement methods to support such functionality under the profile languages $\mathcal{AWP}$ and $\mathcal{AWPS}$ proposed in [32], as well as a boolean profile language. The methods presented in this dissertation utilize main memory for the storage of the profiles.

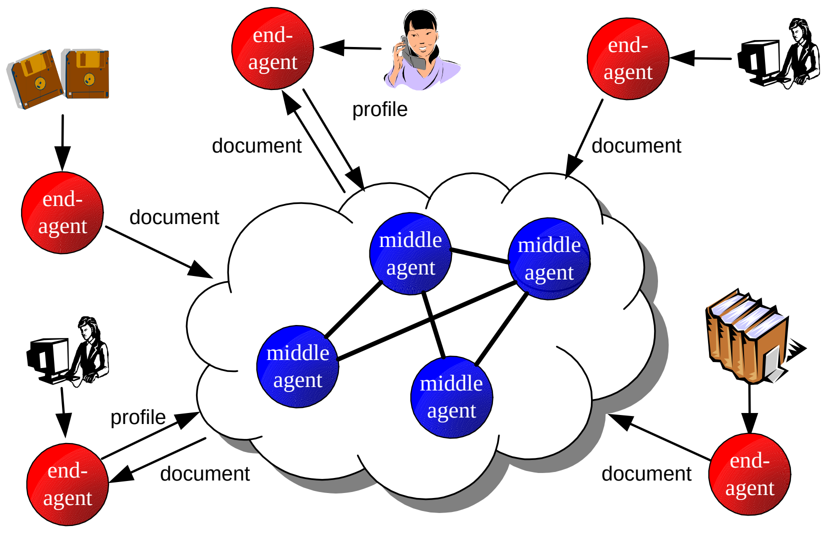


Figure 1.1: A distributed P2P agent architecture for information dissemination

1.2 Organization of the Dissertation

In Chapter 2 we introduce systems relative to the aforementioned scenario and methods that propose better profile handling. In Chapter 3, we present appropriate languages for information dissemination, originally defined in [32]. In Chapter 4, 5 and 6, we present algorithms and data structures that offer support for the languages of Chapter 3. Finally, we conclude this dissertation and propose future research in the area in Chapter 7.

Chapter 2

Related Work

In this chapter, we present some systems implemented for information dissemination. We present the context of each of these systems, the problems they try to solve, as well as their architecture. Finally, data models and languages relative to the ones of this thesis are reviewed.

2.1 Information Push Vs Information Pull

The concept adopted in information retrieval as well as in most database systems, is that of *information pull*. This means that the system maintains a database of documents (or, generally content). Whenever a user needs to search for a file he might be interested in, he connects to the system, submits a query and then, gets any relevant information according to the query he entered. On the other hand, the concept of Selective Information Dissemination is the storage of queries (which are now called *profiles*) and not of the documents. This alternate setting implements *information push*. Under information push, a user submits a profile into the system. After that, whenever a document that is relevant to the user's interests (expressed by his profile) arrives, a notification or the document itself is forwarded to the user. The process of the evaluation whether a document should be sent or not to a user given his profile, is called *filtering* or *matching*.

Information dissemination has another major difference with information retrieval: in information retrieval, when a query is entered, the system returns a ranked list of candidate documents that might interest the user. An answer set is almost always returned, no matter how relevant the documents in this set might be. But, in information dissemination, we do not have the luxury of bothering the user with useless documents. So, in the case where an information retrieval system would return a ranked list of documents, an information dissemination system returns only the these documents that worth to be seen by the user. These usually are the top-ranked documents of the list returned by the respective information retrieval system. There might not be any documents returned, if none of them is similar enough to what the user needs.

There are many difficulties in establishing a trustworthy query model through which a user can accurately express his demands. What is usually done, is ask the user to enter some keywords and then try to evaluate matching using these keywords. An alternative idea, is to employ *relevance feedback*, in addition to the above. This means that the user provides feedback to the system about how much he is interested in a returned document. The system then uses this information to adjust the user profile and maybe become more mature itself about the meaning of similar profiles.

Now that the concepts of information dissemination have been briefly discussed, let us present some representative systems for information dissemination along with their languages.

2.2 The SIFT Information Dissemination System

In this section, we present the Stanford Information Filtering Tool (SIFT) originally presented in [38]. SIFT was used for the dissemination of Usenet articles. It was serving 18400 users and 80000 profiles daily and evolved in a commercial system.

SIFT was originally based on a client-server model. The profile database was stored in a main server, that also had the responsibility of alerting the users for documents. A user submits his queries through email or through a web interface. The notifications are returned to him by email.

A user of SIFT can submit either boolean profiles containing conjunctions or negations (evaluated under the boolean model) [38, 37] and vector space profiles containing phrases and similarity thresholds (evaluated under the vector space model) [38]. In order to improve efficiency, SIFT employs various methods using inverted indices. The data structures that compose the profile database in SIFT were implemented under the perspective that user profiles will grow over time. The overall design and implementation targets efficiency and matching speed. Efficiency is also the metric for the evaluation of the system, rather than precision or recall (which are the basic evaluation metric in Information Retrieval). Another interesting characteristic of SIFT is the incorporation of a *similarity threshold* as part of a VSM profile. With this threshold, SIFT implements the idea described in Section 2.1, that an information dissemination system returns only documents worth to be seen. In the case of SIFT, only documents whose similarity (under VSM) is above the threshold are sent to the user. Under the boolean model, this is easier to implement, as the definition of matching is quite straightforward.

Distributed versions of SIFT [36] have also been studied. In a distributed version, multiple *distributed dissemination servers* take the place of the single dissemination server of the non-distributed version. Distribution relies on profile replication to more than one servers. A document is also posted in more than one servers. Each server matches an incoming document with only the profiles stored in it (which are obviously less than the profiles stored in the whole system) and sends the document to users that have sent the respective matching profiles. Note that a server cannot identify all the users of the whole system that are interested in a document. So, if we assume that a profile p matches a document d, in order for us to be sure that the user that posted p will get d, we have to send d to at least one of the servers that store p. In other words, if p was posted in a set of server P and d is sent to a set of servers D, then $P \cap D \neq \emptyset$ must hold. It is preferred that d will be sent to exactly one of the servers of P, to prevent the system from sending the document to the user more than once. After experimenting with various protocols, it was found that a grid organization of the peers with balanced sizes of P and D provide the best distribution of documents and profiles. Moreover, taking advantage of the possibility that two or more nearly located clients can be interested in the same document, network utilization may be reduced, by establishing a *profile group* with the profiles of these users. A delivery mechanism called *profile grouping* uses this information to send the document to a *distribution server* (also located near the clients and is connected in a high speed, such as a LAN, with them), which will then send the document to each of the clients.

2.3 The Hermes Notification Service

In this section, we present the Hermes notification service [16]. The setting in Hermes consists of users and digital libraries. The users wish to be notified about publications or other content available by a digital library, at the time this content is made available in it.

Hermes is a system that overcomes the heterogenities of the various digital libraries and integrates their alert services into a single service. A common interface is available to the users to express their demands on content. Also, Hermes supports an interface for the digital libraries themselves, through which they can join the system and offer, distribute or advertise their content. The user interface is web based, while notifications are sent to users by email. More protocols are planed to be implemented to send notifications to users.

The system is capable of integrating very diverse provider services. Specifically, the providers can be *active* or *passive*, depending on whether they have their own alert service or not. Moreover, they can be *cooperative* or *non-cooperative*. Cooperative providers send the notifications in an easy to parse format (e.g. XML), with well defined metadata. Non-cooperative providers on the other hand, send human readable notifications, such as emails or web pages. Specialized wrappers are required in order to parse information sent by non-cooperative providers.

The profiles a user can submit consist of a *query* and of a *notification policy* part. The notification policy part contains information for example about how often a batch of notifications for the profile should be sent (e.g. daily, weekly), about the protocol under which the notifications should be sent (e.g. email) and about the format of the notification (e.g. XML, plain text etc). The query part of a profile consists of an optional *boolean expression* and and optional *ranking part*. The boolean expression is an SQL-like attribute query, while the ranking part contains a *relevance threshold* and a term list. This term list may include phrases, proximity operators or weighted terms. When only the boolean part of the query filters incoming documents, they are matched according to boolean semantics. When only the ranking part constrains them, the documents are ranked using a process similar to information retrieval. Then, the documents which are similar to the query more than the given relevance threshold are returned to the user. Alternatively, the user can select a number n of documents to be returned. In that case, the n top-ranked documents are returned. When both parts of the query contain constraints, the query is used as a two-stage filter. Documents ranked according to the ranking part are only those that qualify matching with the boolean expression.

The architecture of Hermes consists of three components: The *Observer*, responsible for gathering the information from providers, the *Filter*, which performs matching and the *Notifier*, that sends the notifications to users. The Observer contains all the necessary modules that receive information from providers. There are also wrappers to support information extraction of data from non-cooperative providers. For the support of active providers, one or more receivers (which are protocol dependent) are enough. The notification from the provider is forwarded to the appropriate receiver (that implements the corresponding protocol to handle the notification). Then, from the receiver the notification is forwarded to a queue and from there to the appropriate wrapper. The wrapper transforms the notification in a format internally used by Hermes. After that the document is ready for matching. In order to support passive providers, the Observer uses a module called *Scheduler*. Scheduler generates an artificial provider notification. This notification triggers the respective wrapper to send a request to the passive provider. The answer of the provider is forwarded to the appropriate receiver and is treated as a notification for the rest of the process.

There were two alternatives to implement filtering. The first one used specialized middleware called *message-oriented middleware (MOM)*. The second one was based on relational databases. MOM is software that apart from filtering also supports transactions and buffering of messages. MOM can also serve as a connection point between different parts of the system, making them more independent from each other. This results to a more stable system, since failure of one module does not affect the rest of the system. But, there are some tradeoffs. MOM can evaluate matching of documents only against boolean queries. In addition, MOM cannot support a big number of profiles, as a notification needs too much time to reach an interested user. In other words, the system is unscalable using MOM (the results show that practically a maximum of 1024 subscribers can be supported). In the database solution, there are three tables: One for the profiles (having subscriber identifiers as a key), one for messages (with message id as a key) and a *queue* table in which notifications are stored before sending them to users (using both subscriber id and message id as a key). When a document arrives, new notifications should be generated and inserted into the queue table for the interested users. We can do that with the use of triggers , but this would mean that a number of triggers equal to the number of stored profiles should be initiated whenever a new document arrived, a solution that is clearly time consuming. On the other hand, using one trigger to examine all profiles and accordingly update the queue table does not offer any advantages. Instead, the authors suppose that users are tolerant to delays of one day, so what is done is batch processing of documents. For each profile stored into the profile table, the filtering algorithms match all newly arrived documents (documents arrived during the day) in the document table and fill up the queue table accordingly. This results to a filtering speedup, but it is not satisfying since the time to match 10000 documents with 10000 profiles with 0.1% selectivity was found to be about 3 hours.

To sum up, Hermes lacks scalability and processing speed. The results of the previous paragraph were extracted by testing the system with only boolean attribute-based profiles, while the rest of the supported language were not implemented. In conclusion, we could say that the idea of using relational databases or naive algorithms to perform matching gives unsatisfying results compared to other systems [36, 38].

2.4 The Distributed Information Alert System DIAS

The DIAS information dissemination system, originally presented in [21], is studied in this section. Like Hermes, DIAS is targeting in dissemination of digital library information. Furthermore, users may also publish documents. As we will see, DIAS has the most expressive data language of all systems discussed in this chapter.

The architecture of DIAS is based on a central peer-to-peer network of *middle agents*. An information provider publishes documents through a so called *resource agent*. A user of the system utilizes an *end-agent* (or *personal agent*) through which he can connect to the middle agent network and send his profiles or receive notifications about documents. The middle agents of the P2P network only communicate with each other and with resource agents as well as personal agents.

A resource agent has the mission to collect the information from an information provider and upload it to the rest of the network. Middle agents are the ones that store the profiles posted by personal agents and that perform matching of profiles with documents posted by resource agents. After the process of matching, middle agents send the notifications to the personal agents representing the users that submitted the matching profiles. Future work on this system will target in efficient distribution of documents and profiles, in order to achieve even smaller matching times and less network utilization.

The language currently supported in DIAS is a subset of $\mathcal{AWP}$ and $\mathcal{AWPS}$ [22, 32]. The models $\mathcal{AWP}$ and $\mathcal{AWPS}$ as well as efficient algorithms for them are presented later in this dissertation. For now, it is enough to know that each profile is a conjunction of atomic profiles of the form $\psi \wedge \sigma$, where ψ is a conjunction of atomic profiles under $\mathcal{AWP}$ (see Chapter 3), while σ is a conjunction of atomic profiles under $\mathcal{AWPS}$ (see Chapter 3).

Each profile in DIAS is replicated in all middle agents. In order for a middle agent A to propagate a newly received (from a personal agent) profile to all other middle agents, a *spanning tree* is used. Specifically, A sends the profile to all of its neighbors. Each middle agent M of the other middle agents of the network, runs the following algorithm: As soon as it receives the profile from a middle agent N, M examines if N is on the shortest path connecting M with A. If this is the case, M forwards the profile to all of its neighbors, except N. This algorithm requires that each middle agent M knows its (unique) neighbor that lies on the shortest path between M and any other middle agent in the network. To be able to keep track of this information, each middle agent maintains a *routing table*. The construction of such a table is a well known problem of data networks and there are several distributed algorithms that solve it. One of them is the asynchronous distributed version of Bellman-Ford's algorithm [14].

In order to reduce network utilization, each middle agent M propagates an $\mathcal{AWP}$ profile ψ to other middle agents by subscribing ψ to them. In order to reduce network utilization, ψ is propagated to a neighbor N of M only if M has not subscribed a more general profile to N. To achieve evaluation of whether ψ is more general from another profile or not, a *partially ordered set* (or *profile poset* [25]) is kept in each middle agent. This data structure is able to keep information for each $\mathcal{AWP}$ profile ψ about which $\mathcal{AWP}$ profiles are more general ψ. Also, for each profile ψ there is a set containing those middle agents that subscribed with ψ, as well as a set with those middle agents to which ψ was forwarded. These are necessary data in order to achieve reliable forwarding of subscriptions and notifications.

DIAS implements processes notifications in the following way: When a notification n that matches a profile ψ is published to a middle agent M, M routes the profile to the middle agent A, where ψ was initially subscribed. This routing is done using the *reverse path* that was created when ψ was propagated in the middle agent network. Filtering could be performed by the poset structure available at each node. But this solution would be very expensive, so specialized filtering algorithms are used to perform matching. As we will see in Chapter 6, there are scalable algorithms able to perform matching under the data models of $\mathcal{AWP}$ and $\mathcal{AWPS}$. Moreover, we will see that these algorithms can efficiently handle millions of profiles.

The DIAS system discussed in this section is still under development and is a part of project DIET[1].

2.5 The Multi-Modal Approach to VSM

In this section, we present the *Multi-Modal (MM)* approach to vector space user profiles that could help to increase the precision and recall of the respective data model under an information dissemination setting. The methods presented in this section were originally proposed in [7].

The data model considered in MM approach, is the *vector space model* [3]. In other systems and languages supporting the vector space model, such as SIFT [38] or $\mathcal{AWPS}$ [22, 32], a user profile is represented as a vector. In the MM approach, a user profile is represented as a set of *profile vectors.* This is because the MM approach supposes that a user's interests cannot be represented with a single profile. So, each of the profile vectors represents only a part of the user's interests, while altogether, they build up a profile that identifies him more precisely. One could argue that a user could submit more than one vector profiles to have the above functionality, so forming vector profile sets is not necessary. But, by using the MM profiles, we have a better view of a user's interest and we can use some interesting reformulation algorithms to increase precision and recall. These algorithms are based on *relevance feedback.* Also, the MM approach offers us a method to keep track of the changes in user's interests over time.

Filtering of a document d, with vector representation v_d, with a MM profile P is performed by matching each of the profile vectors p_i of P with v_d. There is a *similarity threshold*, in order to separate the matching profile vectors from the others. If there exists a p_i that matches, d, the entire profile P is considered to match d.

The user submitted a profile P (= set of profile vectors p_i) provides relevance feedback to the system by evaluating each document d (with a vector representation v_d) of the ones returned to him as relevant or irrelevant. The system represents his selection with a parameter f_d. For documents evaluated as relevant, $f_d = 1$, while for those identified as irrelevant, $f_d = -1$. After that, the profile vector p_{act} that was found to be the most similar to d of all vector profiles of P, is considered. If p_{act}'s similarity to v_d is greater than a threshold δ (with $0 \leq \delta \leq 1$), then p_i is adjusted in order to come closer (if d was judged to be relevant by the user) or to move away (if d was judged to be irrelevant by the

[1]http://www.dfki.de/diet

user) from v_d. The distance and direction of p_{act}'s move is relative to v_d and an *adaptability value* λ, with $0 \leq \lambda \leq 1$. In the case that there is no p_i the similarity of which with v_d is greater than δ, action is taken only if the document was judged as relevant. In that case, a new profile vector, equal to v_d, is created for P.

In order for all the above to be implemented, too much storage space as well as processing power is required, so that filtering can be completed in rational time. This is because we have profiles consisting of many vectors, which are often too big (consider the case where a new profile vector is created from a document). So, techniques that save storage space or speedup filtering could be proved very useful. In order to shrink filtering time and storage space needs, the authors of [7] suggest to reduce the number of profile vectors of a profile, by merging two of them into a new profile vector or by deleting the ones that are judged to not represent the user's needs any more.

Decision of whether a profile vector must be deleted is taken by utilizing negative feedbacks. For each profile vector, there is a parameter called *strength* of the profile vector. Each profile vector's strength is initialized to a default value upon establishment of the profile vector. Whenever a profile vector p_i is found to match a document d that is judged to be irrelevant by the user, the strength of p_i is reduced. When it becomes less than a threshold value, p_i is deleted.

In addition to profile vector deletion, there is also the option to merge profile vectors in order to reduce the profile size. This is done after filtering and after p_{act} has been adjusted according to v_d and relevance feedback. The similarities of both p_{act} and p_c are considered in order to decide if p_{act} and p_c will be merged into a new profile vector. If its similarity with p_{act} is greater than δ, then p_{act} and p_c are merged to form a new profile vector. The new profile vector depends on p_{act}, p_c and also on their strengths. We can imagine that the new profile vector the sum of p_{act} and p_c. Its own strength is equal to the sum of strengths of p_{act} and p_c. Of course, p_{act} and p_c are deleted after merging, as the new profile vector has taken their place. Note that the new profile vector's similarity with another profile vector of P can be greater than δ. In that case, merging of the two profile vectors should occur. This is left to be done in subsequent iterations, as in a different case there would be a great time overhead in filtering due to profile vector merging.

Evaluation of the multi-modal approach has proved that it is more accurate than a uni-modal one. Also, is was shown that the mergings, creations and deletions of profile vectors during filtering do represent the user interest changes over time. Moreover, the MM approach's adaptability was proven to be more efficient than other traditional approaches to relevance feedback. Finally, the MM approach has proved to successfully combine speed and effectiveness.

2.6 Conclusions

The concept of information dissemination, as well as systems that offer information dissemination functionality were presented in this chapter. Firstly, the conceptual differences between information push and information pull were presented. After that, we discussed SIFT, thee data model of which is based on both the boolean and the vector space model. Next, a system (named Hermes) that tries to solve the problem of information dissemination in the World Wide Web was shown. We presented one more system, named DIAS, that targets information dissemination. At the moment, DIAS's model is the boolean model. This can easily change, as support for models and data languages presented in the next chapter can be added with no significant effort. Lastly, we presented an alternate approach (Multi-Modal) to represent a user profile. This approach is based on the vector space model. It extends the basic representation of a user profile by adding various heuristics in order to improve precision and recall.

Chapter 3

Data Models and Query Languages

In this chapter, we will present the models $\mathcal{WP}$, $\mathcal{AWP}$ and $\mathcal{AWPS}$ defined in [15, 19, 22, 21, 20, 23, 32]. For the presentation of these models we rely on the publications [15, 19, 22, 21, 20, 23, 32] and present the definitions given there by the developers of these models verbatim.

We define formally the models $\mathcal{WP}$, $\mathcal{AWP}$ and $\mathcal{AWPS}$, and their corresponding languages for textual information dissemination in distributed agent systems such as the ones briefly discussed in Chapter 2. Data model $\mathcal{WP}$ is based on free text and its query language is based on the *boolean model with proximity operators*. The concepts of $\mathcal{WP}$ extend the traditional concept of proximity in IR [3, 9, 10] in a significant way and utilize it in a content language targeted at information dissemination applications. Data model $\mathcal{AWP}$ is based on *attributes* or *fields* with finite-length strings as values. Its query language is an extension of the query language of data model $\mathcal{WP}$. Our work on $\mathcal{AWP}$ complements recent proposals for querying textual information in distributed event-based systems [6, 5] by using linguistically motivated concepts such as *word* and not arbitrary strings. This makes $\mathcal{AWP}$ potentially very useful in some applications (e.g., alert systems for digital libraries or other commercial systems where similar models are supported already for retrieval). Finally, the model $\mathcal{AWPS}$ extends $\mathcal{AWP}$ by introducing a "similarity" operator in the style of modern IR, based on the vector space model [3]. The novelty of the work in this area is the move to query languages much more expressive than the one used in the information dissemination system SIFT [38] where documents and queries are represented by free text. The similarity concept of $\mathcal{AWPS}$ is an extension of the similarity concept pioneered by the system WHIRL [12] and recently also used in the XML query language ELIXIR [11]. We note that both WHIRL and ELIXIR target information retrieval and integration applications, and pay no attention to information dissemination and the concepts/functionality needed in such applications.

3.1 Text Values and Word Patterns

In this section we present our first data model and query language for textual information dissemination. The data model is based on free text which is captured formally by the concept of text value. Our query language is based on the Boolean model with proximity operators. Queries in this model are formalized using the concept of word pattern [10]. The two basic concepts of this section (text values and word patterns) are subsequently used in Section 3.3 to define the attribute-based data model and query language.

We assume the existence of a finite *alphabet* $\mathbf{\Sigma}$. A *word* is a finite non-empty sequence of letters from $\mathbf{\Sigma}$. We also assume the existence of an infinite set of words called the *vocabulary* and denoted by $\mathcal{V}$.

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

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

Example 3.1 *In all the examples of this chapter, our vocabulary will be the vocabulary of the English language and will be denoted by $\mathcal{E}$. The string*

$$Wavelet\ Image\ Coefficients$$

can be represented by a text value s of length 3 over vocabulary $\mathcal{E}$ with $s(1) = Wavelet$, $s(2) = Image$ etc. The text value "$Image\ Coefficients$" is included in s.

We now give the definition of word-pattern. The definition is given recursively in three stages.

Definition 3.2 *Let $\mathcal{V}$ be a vocabulary. A* proximity-free word pattern *over vocabulary $\mathcal{V}$ is an expression in any of the following forms:*

1. *w where w is a word in the vocabulary $\mathcal{V}$.*

2. *$\neg wp$ where wp is a proximity-free word pattern.*

3. *$wp_1 \wedge wp_2$ where wp_1, wp_2 are proximity-free word patterns.*

4. *$wp_1 \vee wp_2$ where wp_1, wp_2 are proximity-free word patterns.*

5. *(wp) where wp is a proximity-free word pattern.*

A proximity-free word pattern will be called positive *if it does not contain the negation operator.*

Example 3.2 *The following are proximity-free word patterns that might appear in queries of a user of a news dissemination system interested in articles on a specific field of Neural Networks:*

$$Network, \quad Recognition \wedge System, (Input \vee Recognition) \wedge System,$$

$$Middle \wedge Layer \wedge \neg Binary \wedge Logistic \wedge Units$$

We now introduce a new class of word patterns that allows us to capture the concepts of *order* and *distance* between words in a text document. We will assume the existence of a set of *(distance) intervals* $\mathcal{I}$ defined as follows:

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

The symbols $\in$ and $\subseteq$ will be used to denote membership and inclusion in an interval as usual.

The following definition uses intervals to impose lower and upper bounds on distances between word patterns.

Definition 3.3 *Let $\mathcal{V}$ be a vocabulary. A* proximity word pattern *over vocabulary $\mathcal{V}$ is an expression*

$$wp_1 \prec_{i_1} wp_2 \prec_{i_2} \cdots \prec_{i_{n-1}} wp_n$$

where $wp_1, wp_2, \ldots, wp_n$ are positive proximity-free word patterns over $\mathcal{V}$ and $i_1, i_2, \ldots, i_{n-1}$ are intervals from the set $\mathcal{I}$. The symbols $\prec_i$ where $i \in \mathcal{I}$ are called proximity operators*. The number of proximity-free word patterns in a proximity word pattern (i.e., n above) is called its* size.

Example 3.3 *The following are proximity word patterns:*

$$Recognition \prec_{[0,0]} System, \ Continuous \prec_{[0,0]} State \prec_{[0,0]} Space,$$

$$Middle \prec_{[0,0]} Layer \prec_{[0,3]} Binary \prec_{[0,0]} Logistic \prec_{[2,5]} Units,$$

$$Decision \prec_{[0,0]} Tree \prec_{[1,10]} Feature \prec_{[0,2]} (Selection \wedge Algorithm),$$

$$Input \prec_{[0,\infty)} Pattern$$

The proximity word pattern $wp_1 \prec_{[l,u]} wp_2$ stands for "word pattern wp_1 is *before* wp_2 and is separated by wp_2 by *at least l* and *at most u words*". In the above example $Layer \prec_{[0,3]} Binary$ denotes that the word "Layer" appears before word "Binary" and at a distance of at least 0 and at most 3 words. The word pattern $Recognition \prec_{[0,0]} System$ denotes that the word "Recognition" appears exactly before word "System" so this is a way to encode the string "Recognition

System". We can also have arbitrarily long sequences of proximity operators with similar meaning (see the examples above). Note that proximity-free subformulas in proximity word-patterns can be more complex than just simple words (but negation is *not* allowed; this restriction will be explained below). This makes proximity-word patterns a very expressive notation.

Definition 3.4 *Let $\mathcal{V}$ be a vocabulary. A* word pattern *over vocabulary $\mathcal{V}$ is an expression in any of the following categories:*

1. *a proximity-free word pattern over $\mathcal{V}$*
2. *a proximity word pattern over $\mathcal{V}$*
3. *$wp_1 \wedge wp_2$ where wp_1, wp_2 are word patterns.*
4. *$wp_1 \vee wp_2$ where wp_1, wp_2 are word patterns.*
5. *(wp) where wp is a word pattern.*

A word pattern will be called positive *if its proximity-free subformulas are positive.*

Example 3.4 *The following are word patterns of the most general kind we allow:*

$$Middle \wedge (Layer \prec_{[0,10]} (Binary \wedge Units)),$$

$$Feature \wedge (Decision \prec_{[0,0]} Tree) \wedge \neg Algorithm,$$
$$Feature \wedge (Decision \prec_{[0,0]} Tree) \wedge (Selection \prec_{[0,0]} Algorithm),$$

$$Estimation \wedge ((Continuous \prec_{[0,0]} State \prec_{[0,0]} Space) \vee$$
$$(Feature \prec_{[0,0]} Selection \prec_{[0,0]} Algorithm)),$$

$$(Middle \prec_{[0,0]} Layer) \wedge (Binary \prec_{[0,0]} Logistic \prec_{[0,0]} Units)$$

We have here completed the definition of the concept of word pattern. We now turn to defining their semantics.

3.2 Semantics

We now give meaning to the expressions that define word patterns. First, we define what it means for a text value to satisfy a proximity-free word pattern.

Definition 3.5 *Let $\mathcal{V}$ be a vocabulary, s a text value over $\mathcal{V}$ and wp a proximity-free word pattern over $\mathcal{V}$. The concept of s* satisfying *wp (denoted by $s \models_P wp$) is defined as follows:*

1. *If wp is a word of $\mathcal{V}$ then $s \models wp$ iff there exists $p \in \{1, \ldots, |s|\}$ and $s(p) = wp$.*

2. *If wp is of the form $\neg wp_1$ then $s \models wp$ iff $s \not\models wp_1$.*

3. *If wp is of the form $wp_1 \wedge wp_2$ then $s \models wp$ iff $s \models wp_1$ and $s \models wp_2$.*

4. *If wp is of the form $wp_1 \vee wp_2$ then $s \models wp$ iff $s \models wp_1$ or $s \models wp_2$.*

5. *If wp is of the form (wp_1) then $s \models wp$ iff $s \models wp_1$.*

The above definition mirrors the definition of satisfaction for Boolean logic [27]. This will allow us to draw on a lot of related results in the rest of this chapter.

Example 3.5 *Let s be the following text value:*

Currently most speech recognition systems are based on hidden Markov models

Then $s \models Regognition \wedge Systems$.

The following definition captures the notion of a set of positions in a text value containing exactly the words that contribute to the satisfaction of a proximity-free word pattern. This notion is then used to define satisfaction of proximity word patterns.

Definition 3.6 *Let $\mathcal{V}$ be a vocabulary, s a text value over $\mathcal{V}$, wp a proximity-free word pattern over $\mathcal{V}$, and P a subset of $\{1, \ldots, |s|\}$. The concept of s* satisfying *wp* with set of positions *P (denoted by $s \models_P wp$) is defined as follows:*

1. *If wp is a word of $\mathcal{V}$ then $s \models_P wp$ iff there exists $x \in \{1, \ldots, |s|\}$ such that $P = \{x\}$ and $s(x) = wp$.*

2. *If wp is of the form $wp_1 \wedge wp_2$ then $s \models_P wp$ iff there exist sets of positions $P_1, P_2 \subseteq \{1, \ldots, |s|\}$ such that $s \models_{P_1} wp_1$, $s \models_{P_2} wp_2$ and $P = P_1 \cup P_2$.*

3. *If wp is of the form $wp_1 \vee wp_2$ then $s \models_P wp$ iff $s \models_P wp_1$ or $s \models_P wp_2$.*

4. *If wp is of the form (wp_1) then $s \models_P wp$ iff $s \models_P wp_1$.*

Now we define what it means for a text value to satisfy a proximity word pattern.

Definition 3.7 *Let $\mathcal{V}$ be a vocabulary, s a text value over $\mathcal{V}$ and wp a proximity word pattern over $\mathcal{V}$ of the form*

$$wp_1 \prec_{i_1} wp_2 \prec_{i_2} \cdots \prec_{i_{n-1}} wp_n.$$

Then $s \models wp$ iff there exist sets $P_1, P_2, \ldots, P_n \subseteq \{1, \ldots, |s|\}$ such that $s \models_{P_j} wp_j$ and $min(P_j) - max(P_{j-1}) - 1 \in i_{j-1}$ for all $j = 2, \ldots, n$ (the operators max and min have the obvious meaning).

Example 3.6 *The text value*

Currently most speech recognition systems are based on hidden Markov models

satisfies the following word patterns:

$$Recognition \prec_{[0,6]} Markov \prec_{[0,0]} Models$$

$$Recognition \prec_{[0,0]} (Systems \vee Methods) \prec_{[0,6]} Markov,$$

$$Recognition \prec_{[0,10]} Markov \prec_{[0,0]} Models$$

The sets of positions required by the definition are for the first word pattern $\{4\}$, $\{10\}$ *and* $\{11\}$*, for the second,* $\{4\}$, $\{5\}$ *and* $\{10\}$*, and for the third one* $\{4\}$, $\{10\}$ *and* $\{11\}$.

If the structure of wp falls under the four cases of our most general definition (Definition 3.4), satisfaction is similarly defined in a recursive way as in Definition 3.5 (for Cases 1, 3 and 4) and Definition 3.7 (for Case 2).

Example 3.7 *The text value*

Currently most speech recognition systems are based on hidden Markov models

satisfies word pattern $Speech \wedge (Recognition \prec_{[0,0]} Systems \prec_{[0,7]} Models)$.

3.3 An Attribute-Based Data Model and Query Language

Now that we have studied text values and word patterns in great detail, we are ready to define our second data model and query language. This data model for text documents is based on *attributes* or *fields* with finite-length strings as values. Attributes are used to encode information such as author, title, date, body of text and so on. This simple data model is restrictive since it offers a rather flat view of a text document, but it has wide applicability as we will show below.

We start our formal development by defining the concepts of document schema and document. Throughout the rest of this chapter we assume the existence of a countably infinite set of attributes $\mathbf{U}$ called the *attribute universe.*

Definition 3.8 *A* document schema $\mathcal{D}$ *is a pair* $(\mathcal{A}, \mathcal{V})$ *where* $\mathcal{A}$ *is a subset of the attribute universe* $\mathbf{U}$ *and* $\mathcal{V}$ *is a vocabulary.*

Example 3.8 *An example of a document schema for a news dissemination application is* $\mathcal{D} = (\{SENDER,\ EMAIL,\ BODY\}, \mathcal{E})$.

Definition 3.9 *Let $\mathcal{D}$ be a document schema. A* document *d over schema $(\mathcal{A},\mathcal{V})$ is a set of attribute-value pairs (A, s) where $A \in \mathcal{A}$, s is a text value over $\mathcal{V}$, and there is at most one pair (A, s) for each attribute $A \in \mathcal{A}$.*

Example 3.9 *The following is a document over the schema of Example 3.8:*

$$\begin{aligned}\{ & (SENDER, \text{“}Manolis\ Koubarakis\text{”}),\\ & (EMAIL, \text{“}manolis@ced.tuc.gr\text{”})\\ & (BODY, \text{“}Currently\ most\ speech\ recognition\ systems\\ & are\ based\ on\ hidden\ Markov\ models\text{”})\ \}\end{aligned}$$

The syntax of our query language is given by the following recursive definition.

Definition 3.10 *Let $\mathcal{D} = (\mathcal{A},\mathcal{V})$ be a document schema. A* query *over $\mathcal{D}$ is a formula in any of the following forms:*

1. *$A \sqsupseteq wp$ where $A \in \mathcal{A}$ and wp is a* positive *word pattern over $\mathcal{V}$. The formula $A \sqsupseteq wp$ can be read as “A contains word pattern wp”.*
2. *$A = s$ where $A \in \mathcal{A}$ and s is a text value over $\mathcal{V}$.*
3. *$\neg\phi$ where ϕ is a query containing no proximity word patterns.*
4. *$\phi_1 \vee \phi_2$ where ϕ_1 and ϕ_2 are queries.*
5. *$\phi_1 \wedge \phi_2$ where ϕ_1 and ϕ_2 are queries.*

Example 3.10 *The following are queries over the schema of Example 3.8:*

$$SENDER \sqsupseteq (John \prec_{[0,2]} Smith),$$

$$(BODY \sqsupseteq (Markov \wedge (Speech \prec_{[0,5]} Recognition))) \wedge \neg SENDER = \text{“}John\ Smith\text{”}$$

3.3.1 Semantics

Let us now define the semantics of the above query language in our dissemination setting. We start by defining when a document satisfies a query.

Definition 3.11 *Let $\mathcal{D}$ be a document schema, d a document over $\mathcal{D}$ and ϕ a query over $\mathcal{D}$. The concept of document d* satisfying *query ϕ (denoted by $d \models \phi$) is defined as follows:*

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

3. *If ϕ is of the form $\neg\phi_1$ then $d \models \phi$ iff $d \not\models \phi_1$.*

4. *If ϕ is of the form $\phi_1 \wedge \phi_2$ then $d \models \phi$ iff $d \models \phi_1$ and $d \models \phi_2$.*

5. *If ϕ is of the form $\phi_1 \vee \phi_2$ then $d \models \phi$ iff $d \models \phi_1$ or $d \models \phi_2$.*

Example 3.11 *The first query of Example 3.10 is not satisfied by the document of Example 3.9 while the second one is satisfied.*

3.4 Extending $\mathcal{AWP}$ with Similarity

Let us now define our third data model $\mathcal{AWPS}$ and its query language. $\mathcal{AWPS}$ extends $\mathcal{AWP}$ with the concept of *similarity* between two text values (the letter $\mathcal{S}$ stands for similarity). The idea here is to have a "soft" alternative to the "hard" operator $\sqsupseteq$. This operator is very useful for queries such as "I am interested in documents sent by John Brown" which can be written in $\mathcal{AWP}$ as

$$SENDER \sqsupseteq (John \prec_{[0,0]} Brown)$$

but it might not be very useful for queries "I am interested in documents about the use of ideas from agent research in the area of information dissemination".

The desired functionality can be achieved by resorting to an important tool of modern IR: the *weight* of a word as defined in the Vector Space Model (VSM) [3, 26, 35]. In VSM, documents (text values in our terminology) are conceptually represented as vectors. If our vocabulary consists of n distinct words then a text value s is represented as an n-dimensional vector of the form $(\omega_1, \ldots, \omega_n)$ where ω_i is the weight of the i-th word (the weight assigned to a non-existent word is 0). With a good weighting scheme, the VSM representation of a document can be a surprisingly good model of its semantic content in the sense that "similar" documents have very close semantic content. This has been demonstrated by many successful IR systems recently (see for example, WHIRL [12])[1].

In VSM, the weight of a word is computed using the heuristic of assigning higher *weights* to words that are frequent in a document and *infrequent* in the collection of documents available. This heuristic is made concrete using the concepts of word frequency and the inverse document frequency defined below.

Definition 3.12 *Let w_i be a word in document d_j of a collection C. The* term frequency *of w_i in d_j (denoted by tf_{ij}) is equal to the number of occurrences of word w_i in d_j. The* document frequency *of word w_i in the collection C (denoted*

[1]Note that in the VSM model and systems adopting it (e.g., WHIRL [12]) word *stems*, produced by some stemming algorithm [28], are forming the vocabulary instead of words. Additionally, *stopwords* (e.g., "the") are eliminated from the vocabulary. These important details have no consequence for the theoretical results of this chapter, but it should be understood that our current implementation of the ideas of this section utilizes these standard techniques.

by df_i) is equal to the number of documents in C that contain w_i. The inverse document frequency *of w_i is then given by $idf_i = \frac{1}{df_i}$. Finally, the number $tf_{ij} \cdot idf_i$ will be called the* weight *of word w_i in document d_j and will be denoted by ω_{ij}.*

At this point we should stress that the concept of inverse document frequency assumes that there is a *collection* of documents which is used in the calculation. In our dissemination scenario we assume that for each attribute A there is a collection of text values C_A that is used for calculating the *idf* values to be used in similarity computations involving attribute A (the details are given below). C_A can be a collection of recently processed text values as suggested in [38].

We are now ready to define the main new concept in $\mathcal{AWPS}$, the similarity of two text values. The similarity of two text values s_q and s_d is defined as the cosine of the angle formed by their corresponding vectors[2]:

$$sim(s_q, s_d) = \frac{s_q \cdot s_d}{\|s_q\| \cdot \|s_d\|} = \frac{\sum_{i=1}^{N} w_{q_i} \cdot w_{d_i}}{\sqrt{\sum_{i=1}^{N} w_{q_i}^2 \cdot \sum_{i=1}^{N} w_{d_i}^2}} \tag{3.1}$$

By this definition, similarity values are real numbers in the interval $[0, 1]$.

Let us now proceed to give the syntax of the query language for $\mathcal{AWPS}$. Since $\mathcal{AWPS}$ extends $\mathcal{AWP}$, a query in the new model is given by Definition 3.3 with one more case for atomic queries:

- $A \sim_k s$ where $A \in \mathcal{A}$, s is a text value over $\mathcal{V}$ and k is a real number in the interval $[0, 1]$.

Example 3.12 *The following are some queries in $\mathcal{AWPS}$ using the schema of Example 3.9:*

$$BODY \sim_{0.6} \text{``}Use\ of\ Markov\ models\ in\ speech\ recognition,$$

$$(SENDER \sqsupseteq (John \prec_{[0,2]} Brown)) \wedge$$
$$(TITLE \sim_{0.9} \text{``}Applicationsi\ of\ Markov\ models\text{''}),$$
$$BODY \sim_{0.9} \text{``}Reliable\ optical\ character\ recognition\ software\text{''}$$

We now give the semantics of our query language, by defining when a document satisfies a query. Naturally, the definition of satisfaction in $\mathcal{AWPS}$ is as in Definition 3.11 with one additional case for the similarity operator:

- If ϕ is of the form $\mathcal{A} \sim_k s_q$ then $d \models \phi$ iff there exists a pair $(A,\ s_d) \in d$ and $sim(s_q, s_d) \geq k$.

[2]The IR literature gives us several very closely related ways to define the notions of weight and similarity [3, 26, 35]. All of these weighting schemes come by the name of $tf \cdot idf$ weighting schemes. Generally a weighting scheme is called $tf \cdot idf$ whenever it uses word frequency in a monotonically increasing way, and document frequency in a monotonically decreasing way.

The reader should notice that the number k in a similarity predicate $A \sim_k s$ gives a *relevance threshold* that candidate text values s should exceed in order to satisfy the predicate. This notion of relevance threshold was first proposed in an information dissemination setting by [17] and later on adopted by [38]. The reader is asked to contrast this situation with the typical information retrieval setting where a ranked list of documents is returned as an answer to a user query. This is not a relevant scenario in an information dissemination system because very few documents (or even a single one) enter the system at a time, and need to be forwarded to interested users.

A low similarity threshold in a predicate $A \sim_k s$ might result in many irrelevant documents satisfying a query, whereas a high similarity threshold would result in very few achieving satisfaction (or even no documents at all). In an implementation of our ideas, users can start with a certain relevance threshold and then update it using relevance feedback techniques to achieve a better satisfaction of their information needs. Recent techniques from adaptive IR can be utilized here [7].

Example 3.13 *The first query of Example 3.12 is likely to be satisfied by the document of Example 3.9 (of course, we cannot say for sure until the exact weights are calculated in the manner suggested above). The second query is not satisfied, since attribute $TITLE$ does not exist in the document. Moreover the third query is unlikely to be satisfied since the only common word between the query and Example 3.9 is the word "Recognition".*

3.5 Conclusions

In this chapter, previous work on data models and query languages for textual information dissemination has been presented. Specifically, the syntax and semantics of models $\mathcal{WP}$, $\mathcal{AWP}$ and $\mathcal{AWPS}$ have been defined. Implementation of basic functionality of a textual information dissemination system under these models is the subject of this dissertation.

The $\mathcal{WP}$ data model is a boolean model with proximity operators, based on free text. $\mathcal{AWP}$ is an extension of $\mathcal{WP}$, as it supports multi-attribute documents, where each attribute is considered to be free text. Finally, we add vector space support to $\mathcal{AWP}$ by introducing the model $\mathcal{AWPS}$. In the next chapter, we introduce some data structures and algorithms appropriate to support these data models and languages. Moreover, we study the complexity of these methods and we experimentally evaluate them.

Chapter 4

The HashTrie Method for Boolean Profiles

In this chapter, we present a method for the support of boolean atomic profiles and we experimentally evaluate and compare this method with other known algorithms.

4.1 Method Description

In this section, the HashTrie method, used for the manipulation of boolean profiles (or boolean parts of profiles), is presented. Specifically, we describe the data structures used for the profile storage and the procedures followed for the insertion and matching of the profiles. We also calculate time and space complexities of the algorithms used for these operations.

HashTrie tries to benefit from the commonalities between profiles. These commonalities are used to decrease the space used by the profiles and are also exploited during filtering. To achieve this, a *trie-like* data structure combined with a hash table is used for the storage of the profiles. The shape of the data structure used to store the profiles is the one presented in Figure 4.1.

Let us now explain the method employed to store profiles into a trie.

Definition 4.1 *Let p, q be profiles such that $q \subseteq p$. Then $r = p \setminus q$ will be called the* remainder of p with respect to q.

Definition 4.2 *Let P be a set of profiles, p a profile in P and $q \subseteq p$. Profile q will be called an* identifying subset of p in profile set P *if there is no other profile r in P such that $q \subseteq r$.*

Example 4.1 *Table 4.1 shows a sample profile set along with some of their possible identifying subsets. One can see that the identifying subsets used for two profiles are the same only if these profiles are identical (this holds for p_1 and p_2).*

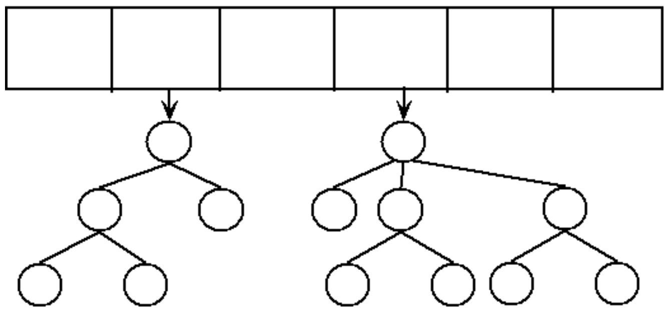


Figure 4.1: The shape of the data structure that stores the profiles under the HashTrie method

Profile	Possible Identifying Subset
p_0: Databases	{Databases}
p_1: Relational $\wedge$ Databases	{Databases, Relational}
p_2: Databases $\wedge$ Relational	{Databases, Relational}
p_3: Neural $\wedge$ Networks $\wedge$ Relational $\wedge$ Databases	{Databases, Neural, Relational}
p_4: Knowledge $\wedge$ Artificial $\wedge$ Relational $\wedge$ Intelligence $\wedge$ Databases	{Artificial, Databases, Intelligence, Knowledge, Relational}
p_5: Artificial $\wedge$ Relational $\wedge$ Intelligence $\wedge$ Databases	{Artificial, Databases, Intelligence, Relational}

Table 4.1: Sample of profiles and some possible identifying subsets

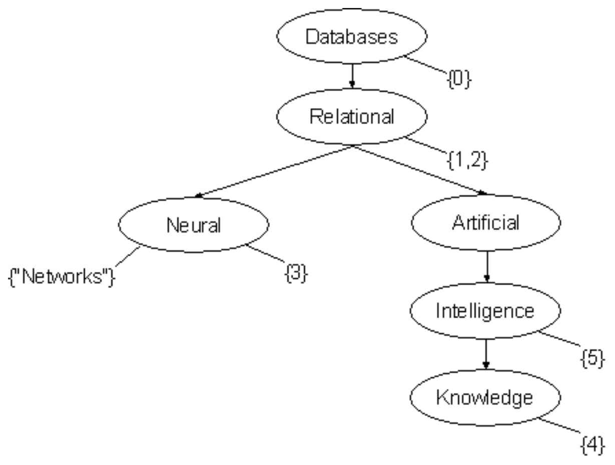


Figure 4.2: A possible trie created from a profile set

As profiles that have common words arrive, HashTrie organizes some of their identifying subsets (one identifying subset for each profile) into a profile trie. The identifying subset with the minimum number of words is selected for each profile. The root of the profile trie (at depth 1) corresponds to a one-word identifying subset. A node n at depth i corresponds to a set I consisting of i words. The set I is a subset of some identifying subsets. Node n represents all identifying subsets identical to I. A node n is implemented as a structure consisting of the following fields:

- $Word$: a string representing one of the words of I.
- $Profiles$: A table containing the profile identifiers for which I is the identifying subset.
- *Remainder* : a table containing the words that form the remainder of the identified profiles.
- *Children list* : a linked list of pairs (w,ptr), where w is a word such that $I \cup \{w\}$ is the set corresponding to a child of n and ptr is a pointer to that child.

Example 4.2 *The profiles* p_0, ..., p_5 *of Table 4.1 can be stored in a trie like the one in Figure 4.2. We can see the profile identifiers stored at the nodes of the trie. We can also see the remainder "Networks" stored at the node with the word "Neural".*

One can easily notice that the selected identifying subset I_p of each profile p can be figured out by simply traversing the trie from the root to the node n

containing p (or backwards) and noting down the word associated with each node. The complete set of words contained in p can be retrieved by taking the union of the remainder of n with I_p, i.e. $I_p \cup remainder[n]$. Only the leaves of a trie can have a non-empty remainder. If a non-leaf node had a non-empty remainder, this would mean that all the profiles in its sub-tries should include the words in its remainder. This is unacceptable. The calculation of the word set identified by a node should not include remainders of the intermediate nodes. The purpose of the remainder is to enable us to better exploit commonalities between profiles. The insertion process described in the rest of this secction will reveal the manner in which the remainder helps.

It is important to understand that using the above structure (tries with nodes containing common words) only profiles that have one or more common words can be stored in a single trie. But, obviously, this does not hold for all the profiles that users submit. To overcome this, we use more than one tries, each containing profiles that have one or more common words. So, for the set of all the profiles, we actually use a forest. A hash table H is used to index the roots of the tries, speeding up searching. A profile p can only be inserted in tries with roots containing words of p. Any such trie will be called a *candidate trie*.

Our intention is to cluster profiles as compactly as possible so that we save storage space but also achieve very fast filtering performance. Whenever a new profile p arrives in the system, we select a trie node n representing the word set J which has the maximum number of common words with p. J may be represented by a node which is located in any of the candidate tries. For example, the node that represents the set $Artificial \wedge Intelligence$ may be located in the trie that has either the word *Artificial* or *Intelligence* as root. Moreover, in each candidate trie, there may be more than one solutions (more than one positions that represent J). Also, in each node of each candidate trie, there may exist more than one sub-tries able to include the best solution. All these force us to perform almost exhaustive search of each of the candidate tries. However we are still able to avoid searching of certain sub-tries. These are the sub-tries the roots of which contain words that does not exist in the document.

From the above, it is clear that the profile insertion procedure consists of two parts:

1. Find the best node in which profile p can be inserted (or else the node n that represents a word set J that has the most possible common words with p).

2. Insert p into n (or create a new node under n to insert p).

Searching of the node that represents J is performed using a special algorithm called ChooseBestTrieNode. ChooseBestTrieNode traverses one of the candidate tries in a depth-first search order. This way, it examines all the nodes of the trie that possibly contain the best solution. By the end of ChooseBestTrieNode's execution, we know the best node for a single trie. This means that we have a *"trie-wide"* solution. What we want is a *"forest-wide"* solution, that is the node n that represents J. All we have to do is to let ChooseBestTrieNode update a global variable, in which n will be stored. This global variable will be updated only when ChooseBestTrieNode finds a better solution than the one already stored in the global variable. Alternatively, we could compare all the trie-wide solutions and pick up the best of them in order to find J. After the execution of any (but not both) of these steps, we can insert p into n.

Insertion of p is done in a way such that its commonality with J is exploited. The insertion procedure follows:

1. Run ChooseBestTrieNode in each of the candidate tries and find the set J which has the maximum number of common words with p.
2. If no set J was found, create a new trie (containing only p) and exit algorithm.
3. Let n be the node representing the set J, $J' = p \setminus J$ and $r = \mathit{remainder}[n]$. We will try to insert p under n.
4. If $J' = r$, store p into n' and exit algorithm.
5. Let C be the set of words contained in both J' and r.
6. If $C = \emptyset$, let $n' = n$, $r' = r$ and $J'' = J'$ and go to step 8.
7. Create a node for each of the words of C. Set them to be each other's child (at most one child per node). Set the one that is nobody's child to be n's child and let n' be the one that has no child. Let $r' = r \setminus C$ and $J'' = J' \setminus C$.
8. If $r' \neq \emptyset$, choose a word of r' to create a new node and set that node to be child of n'. Move the rest of the remaining words as well as the profiles of n to the new node.
9. If $J'' \neq \emptyset$, choose a word of J'' to create a new node and set that node to be child of n'. Store p as well as the remaining words of J' into the new node and exit algorithm.
10. Store p into n' and exit algorithm.

There is a similar method (called *Tree* method) [37] to insert profiles in the tries. The Tree method does not try to insert p in the best possible position (as defined here) of all the tries (and does not utilize any algorithm similar to ChooseBestTrieNode). Instead, it sorts the words of the profile in alphabetical order and only considers the common *prefixes* between profiles. Using the

Tree method, there is a single candidate trie for each profile. Moreover, only common prefixes (and not common subsets) are considered. Also, the remainder of a profile is called *postfix* in the Tree method.

To better understand insertion, consider the profiles p_0 to p_5 of Table 4.1. Assume that there is a trie hash table H with only one trie T_2, shown in Figure 4.8. T_2 contains a profile $p_9 = Artificial \wedge Databases$. Let us consider that each of $p_0, \ldots, p_5$ arrives in the system (p_0 arrives first, p_1 arrives second etc). We will examine the creation and modification of T_2 and other tries in H, after each profile's arrival. For simplicity, in the following example we assume that initially there is only p_9 in the system. To better understand the advantages of HashTrie against the Tree method, the latter is also studied in parallel with HashTrie. So, at each step, we will present the actions taken by HashTrie and the Tree method in order to insert the respective profile.

When p_0 arrives, a new trie T_1 (shown in Figure 4.3) with the word *Databases* as root is created by HashTrie, containing the identifier of p_0. T_2 is not a candidate trie for p_0, as p_0 does not contain the word *Artificial*. p_0 does not contain *Life* either, but this is not important. If *Artificial* existed in p_0, T_2 would be a candidate trie.

The Tree method sorts p_0 alphabetically ($p_0 = (Databases)$) and searches the trie with *Databases* as root for common prefixes. Since there is no such trie at the moment, Tree also creates T_1.

Next, p_1 arrives. Again, T_2 is not a candidate trie (as *Artificial* is not contained in p_1) under HashTrie, so ChooseBestTrieNode examines only T_1. p_1 is found to contain the word *Databases*, so a new node with the word *Relational* is created under the node containing p_0 in T_1. The new node contains the identifier of p_1. The snapshot of T_1 after the insertion of p_1 is shown in Figure 4.4.

Under the Tree method, p_1 is alphabetically sorted ($p_1 = (Databases, Relational)$). So, for the Tree method, T_1 is the unique candidate trie (as *Databases* is its root). Since there are no other nodes in T_1, the Tree method also transforms T_1 to the trie presented in Figure 4.4.

The next profile is p_2. Once more, under HashTrie T_1 is the only candidate trie in H (and the only examined by ChooseBestTrieNode), as p_2 does not contain *Artificial*. We can see that there is already a node in T_1 that represents a set containing all the words of p_2 (the node that contains p_1). This node is detected by ChooseBestTrieNode. So, as we can see in Figure 4.5, p_2 is stored under this node too, with no other modifications in T_1.

Tree sorts p_2 alphabetically ($p_2 = (Databases, Relational)$). Once more, T_1 is the unique candidate trie. Searching for common prefixes, Tree also finds the node containing p_1 and inserts p_2 there. So, Tree also changes T_1 to the one shown in Figure 4.5.

After p_2, p_3 arrives. Again T_1 is the only candidate trie under HashTrie, as p_3 does not contain *Artificial*. ChooseBestTrieNode finds that the node holding p_1 and p_2 represents a set with more p_3's words than any other set represented by a node of a trie of H. So, p_3 is inserted under that node. We can see how T_1 is transformed in Figure 4.6. A new node, with the word *Neural*, is constructed under the one containing p_1 and p_2. This node stores the identifier of p_3, as well as the remainder of p_3. *Neural* (rather than *Networks*) is selected to represent the new node. This is because whenever a new node is to be created, the "leftmost" free word (that is not contained in the word set J) of the profile is selected to represent the new node. The use of this rule makes it possible to apply various heuristics by simply sorting the input profiles. For example, we could sort profiles in descending profile frequency order (implementing the *rank* heuristic, see Section 4.2.5), making frequent words appear in nodes with small depths and infrequent ones to appear more deeply in tries. On the other hand, we could sort profiles in ascending profile frequency order (implementing the *irank* heuristic, see Section 4.2.5), making infrequent words appear in nodes with small depths and frequent ones to appear more deeply in tries.

Until the arrival of p_3, the Tree method inserted profiles in such a way that the resulting organization was the same as the organization resulted after HashTrie. But this does not hold with p_3, as Tree sorts alphabetically its words ($p_3 = (Databases, Neural, Networks, Relational)$). Then, Tree searches for common prefixes in T_1 (as its root contains the alphabetically first word of p_3, *Databases*). It turns out that (*Databases*) is the only common prefix found in T_1. So, p_3 is inserted in T_1, under the root node. T_1 is transformed to the one shown in Figure 4.9. We can see that although p_3 has one more common word with p_1 and p_2 (word *Relational*), this commonality is not exploited. *Relational* exists twice in T_1. This is worse clustering than the one achieved by HashTrie.

Next, p_4 arrives. This time, T_1 (as presented in Figure 4.6) and T_2 (as presented in Figure 4.8) are both candidate tries (as p_4 contains both *Databases* and *Artificial*) for HashTrie, so ChooseBestTrieNode examines both of them. But, in T_2 there is a node representing a set with at most one common word with p_4 (*Artificial*). On the other hand, in T_1 ChooseBestTrieNode finds a node (the one containing p_1 and p_2) that represents a set with two common words with p_4 (*Databases* and *Relational*). So, the node containing p_1 and p_2 in T_1 is the node under which p_4 will be inserted by HashTrie. A snapshot of T_1 after this insertion is shown in Figure 4.7. Same as previous, a new node with the word *Artificial* is created, containing the identifier of p_4 and the words *Intelligence* and *Knowledge* as the remainder of p_4.

Tree on the other hand sorts p_4 ($p_4 = (Artificial, Databases, Intelligence, Knowledge, Relational)$) and only examines T_2 for common prefixes (as T_2's root is *Artificial*, the first alphabetically word of p_4). *Artificial* is the only common prefix found, so p_4 is inserted under the node containing it and T_2 is altered to the one shown in Figure 4.10.

Finally, p_5 arrives. Under the HashTrie method, ChooseBestTrieNode examines both T_1 (as shown in Figure 4.7) and T_2 (as shown in Figure 4.8) and finds that the node (contained in T_1) where p_4 is stored represents a set

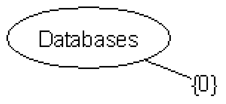


Figure 4.3: T_1 after the insertion of p_0

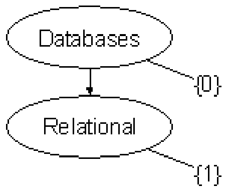


Figure 4.4: T_1 after the insertion of p_1

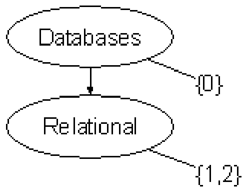


Figure 4.5: T_1 after the insertion of p_2

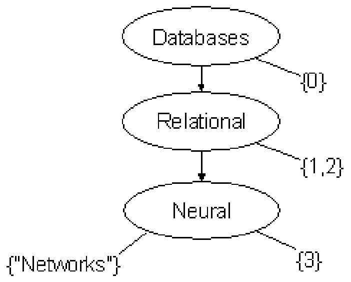


Figure 4.6: T_1 after the insertion of p_3 using HashTrie

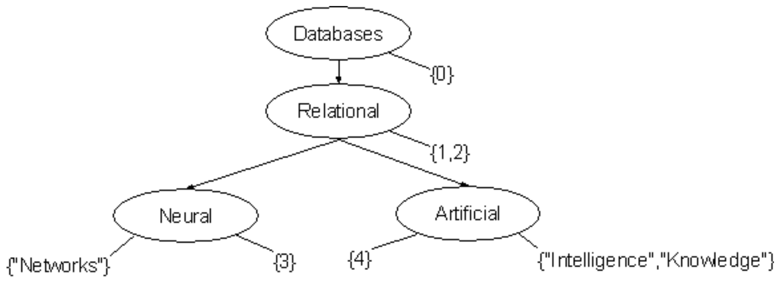


Figure 4.7: T_1 after the insertion of p_4 using HashTrie

that contains more common words with p_5 than any other set represented by a node in any of the tries of H. So, p_5 is inserted under the node containing p_4 and T_1 is transformed into the trie of Figure 4.2. We can see how the concept of remainder is used to exploit commonalities between p_4 and p_5. *Intelligence*, originally contained in the remainder of p_4, was found to be contained in p_5. One could argue that the result would be the same if, instead of p_4's remainder, two other nodes, containing *Intelligence* and *Knowledge* (like the ones of Figure 4.2) were created during p_4's insertion. But, before the arrival of p_5, we were not sure whether p_5 or a profile (for example) $p_6 = Artificial \wedge Relational \wedge Knowledge \wedge$ *Databases* would appear. With the use of remainder we are able to not create certain sub-tries until the time we know how to create and organize them in such a way so that we exploit commonalities as better as possible.

On the other hand, Tree sorts p_5 alphabetically ($p_5 = (Artificial$, *Databases*, $Intelligence$, *Relational*)) and searches T_2 (as presented in Figure 4.10) for common prefixes (because T_2's root word is the fist alphabetically word of p_5). The node that contains p_4 is found to represent the prefix with the most common words with p_5. So, p_5 is inserted under that node and T_2 changes to the one shown in Figure 4.11.

As demonstrated by the example, HashTrie organizes the profiles in two tries: T_1, as presented in Figure 4.2 and T_2 as presented in Figure 4.8. On the other hand, Tree organizes the profiles in T_1 as shown in Figure 4.9 and in T_2 as shown in Figure 4.11. One can easily see that HashTrie's organization is better, as in the tries created by Tree there are many words repeated in more than once in nodes or postfixes. Also, in the tries created by Tree, there is a total of 9 nodes and a sum of 3 words in postfixes, while in the tries created by HashTrie, there is a total of 7 nodes and a sum of 2 words in remainders. The fact that HashTrie created less nodes and stored less words in remainders is another intuitive verification that HashTrie performed better clustering than Tree.

The algorithm used for the insertion of the profiles is named HASHTRIEINSERT and is presented in Figure 4.12 and Figure 4.13. It inserts a profile p in a node n of a trie T, which is chosen as the best node among all nodes of all candidate tries.

Lines 7 to 21 search all candidate tries for possible positions in them where the profile could be inserted. Variable *tree_x* is a pointer to that node and *tree_rest* is the set of words that do not belong to the set identified by *tree_x*. Variable *tree_depth* indicates the depth of *tree_x*. The algorithm CHOOSEBESTTRIENODE that chooses the best node in a single trie is utilized here. This algorithm is presented and explained later. For the time being, it is enough to know that CHOOSEBESTTRIENODE updates *tree_x* (if necessary, i.e. if it finds a better node), *tree_depth* and *tree_rest* and that it returns an indication whether updated *tree_x* identifies p (which means that *tree_x* represents an identifying subset that contains all of the words of p or that *tree_x* contains a profile identical to p). If this holds, then p is saved in *tree_x* with no other modification to

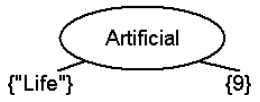


Figure 4.8: T_2 before the insertion of any profile

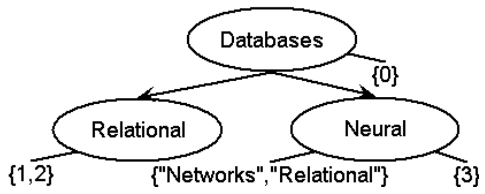


Figure 4.9: T_1 after the insertion of p_3 using Tree

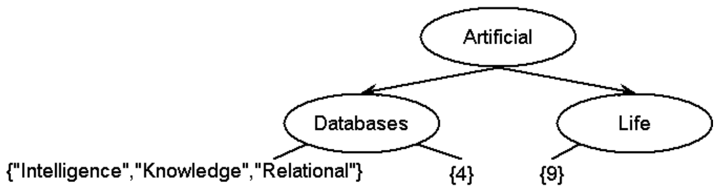


Figure 4.10: T_2 after the insertion of p_4 using Tree

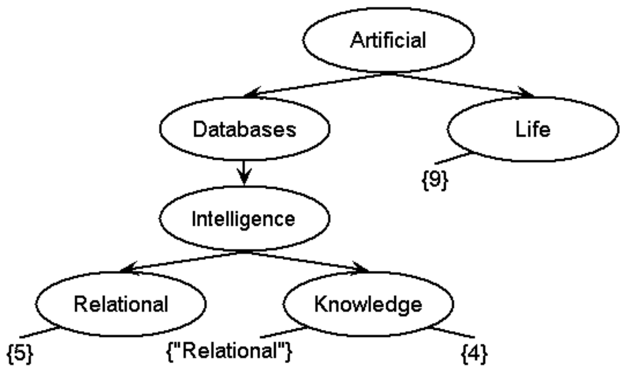


Figure 4.11: T_2 after the insertion of p_5 using Tree

```
algorithm HashTrieInsert
input: a profile $p$ represented as a sorted sequence of words,
       a profile hash table $H$
output: -

$words = elements[p]$
$tree\_depth \leftarrow 0$
$tree\_x \leftarrow null$
$tree\_rest \leftarrow \emptyset$
$non\_existent\_word \leftarrow null$
$same\_found \leftarrow False$
for each word $w$ in $words$ do
    if $same\_found = False$ then
        $rest \leftarrow words - \{w\}$
        let $T$ be the hash table entry with the trie with root $w$
        if $T \neq null$ then
            $same\_found =$ ChooseBestTrieNode($rest$,$T$)
            if $same\_found = True$ then
                add $identifier[p]$ to $profiles[tree\_x]$
            end if
        else if $non\_existent\_word = nil$ then
            $non\_existent\_word \leftarrow w$
        end if
        $rest \leftarrow rest + \{w\}$
    end if
end for
if $same\_found = False$ then
    if $tree\_depth > 0$ or $non\_existent\_word = null$ then
        $common \leftarrow$ the common part of $remainder[tree\_x]$ and $tree\_rest$
        $rem\_rest \leftarrow tree\_rest - common$
        $rem\_remainder \leftarrow remainder[tree\_x] - common$
        $remainder[tree\_x] \leftarrow \emptyset$
        $profiles \leftarrow profiles[tree\_x]$
        $profiles[tree\_x] \leftarrow \emptyset$
```

Figure 4.12: The algorithm HashTrie for the insertion of a profile

```
            **while** $common \neq \emptyset$ **do**
                create a new tree node $y$ and let $ptr\_y$ to point to $y$
                insert the pair ($head[common]$,$ptr\_y$) in $children[tree\_x]$
                $remainder[y] \leftarrow \emptyset$
                $profiles[y] \leftarrow \emptyset$
                $children[y] \leftarrow \emptyset$
                $common \leftarrow tail[common]$
                $tree\_x \leftarrow y$
            **end while**
            **if** $rem\_rest \neq \emptyset$ **then**
                create a new tree node $z$ and let $ptr\_z$ to point to $z$
                insert the pair ($head[rem\_rest]$,$ptr\_z$) in $children[tree\_x]$
                $remainder[z] \leftarrow tail[rem\_rest]$
                $profiles[z] \leftarrow \{identifier[p]\}$
                $children[z] \leftarrow \emptyset$
            **else**
                $profiles[tree\_x] \leftarrow \{identifier[p]\}$
            **end if**
            **if** $rem\_remainder \neq \emptyset$ **then**
                create a new tree node $y$ and let $ptr\_y$ to point to $y$
                insert the pair ($head[rem\_remainder]$,$ptr\_y$) to $children[tree\_x]$
                $remainder[y] \leftarrow tail[rem\_remainder]$
                $profiles[y] \leftarrow profiles$
                $children[y] \leftarrow \emptyset$
            **else**
                $profiles[tree\_x] \leftarrow profiles$
            **end if**
        **else**
            create a new node $x$ of $T$
            $remainder[x] \leftarrow words - \{non\_existent\_word\}$
            $children[x] \leftarrow \emptyset$
            $profiles[x] \leftarrow \{identifier[p]\}$
            insert $x$ in the root hash table of $T$ with $non\_existent\_word$ as its key
        **end if**
    **end if**

**end algorithm**
```

Figure 4.13: The algorithm HashTrie for the insertion of a profile (continued)

```
algorithm ChooseBestTrieNode
input: a set $rest$ representing the remaining words in $p$,
       a profile trie $T$
output: $same\_found$ (indication that the same profile was found in T)

    $same\_found \leftarrow False$
    $depth \leftarrow 1$
    let $x$ be the root node of $T$
    if $tree\_depth = 0$ then
        $tree\_depth \leftarrow 1$
        $tree\_x \leftarrow x$
        $tree\_rest \leftarrow rest$
    end if
    if $rest = \emptyset$ then
        $tree\_x \leftarrow x$
        $tree\_rest \leftarrow \emptyset$
        if $remainder[root] = \emptyset$ then
            $same\_found \leftarrow True$
        end if
    end if
    while $rest \neq \emptyset$ and $same\_found = False$ do
        $child\_found = False$
        if $x$ is a leaf then
            let $n$ be the number of words common in $remainder[x]$ and $rest$
            if $n = |rest| = |remainder[x]|$ then
                $tree\_x \leftarrow x$
                $same\_found \leftarrow True$
            else if $n + depth > tree\_depth$ then
                $tree\_x \leftarrow x$
                $tree\_depth \leftarrow n + depth$
                $tree\_rest \leftarrow rest$
            end if
```

Figure 4.14: The algorithm that finds the best position in a trie

```
        else
          for each $w \in rest$ and $w \notin tested$ do
            if $same\_found = False$ and $child\_found = False$ then
              $tested \leftarrow tested + \{w\}$
              if a pair ($w$,$ptr$) exists in $children[x]$ then
                let $y$ be the node of $T$ pointed to by $ptr$
                $rest \leftarrow rest - \{w\}$
                if $rest = \emptyset$ then
                  $tree\_x \leftarrow y$
                  $same\_found \leftarrow True$
                end if
                push $tested$ to $stack$
                $tested \leftarrow \emptyset$
                increase $depth$ by 1
                $x \leftarrow y$
                if $depth > tree\_depth$ then
                  $tree\_x \leftarrow x$
                  $tree\_depth \leftarrow depth$
                  $tree\_rest \leftarrow rest$
                end if
                $child\_found \leftarrow True$
              end if
            end if
          end for
        end if
        if $same\_found = False$ and $child\_found = False$ and $depth > 1$ then
          pop $tested$ from $stack$
          $x \leftarrow father[x]$
          $rest \leftarrow rest + \{w\}$
          decrease $depth$ by 1
        end if
    end while

end algorithm
```

Figure 4.15: The algorithm that finds the best position in a trie (continued)

any of the tries. Lines 23 to 56 are executed only if *tree_x* represents a set I that has some common words with p (but p and / or I also have some non-common words) or if there is no room for new tries (all the words in p already index a trie). In that case, lines 24 to 29 find the common words between $remainder[n]$ and *tree_rest* and lines 30 to 38 make a new sub-trie under *tree_x* with these words. By doing this, we create a sequence of nodes (children of one another). The last of them, n', identifies a set I' with all the common words between *tree_rest* and I. If there are any words left in *tree_rest* that are not part of I', lines 39 to 44 make a child to n' with one of the remaining words of *tree_rest* as identifier and insert p into this child. If this is not the case, then p is inserted into n' (lines 45 - 46). Lines 48 to 56 do the same for the profiles that were stored in *tree_x*. Note that it is not possible for both p and these profiles to be saved into n'. That would mean that p is identical to them, something that would have been detected by ChooseBestTrieNode and then, p along with the other profiles would be saved in *tree_x*. Lines 58 to 62 are executed in the case that no candidate trie or no suitable node in any of the candidate tries is found. What is done is to create a new trie that only contains p. The word of p that indexes the new trie is chosen in lines 16 - 17.

The algorithm ChooseBestTrieNode is given in Figure 4.14 and Figure 4.15. Given a trie T and a set *rest* of words (which is equal to the set of the words contained in $p - \{root[T]\}$), ChooseBestTrieNode finds the best (in the greatest possible depth) node n in T where the profile can be inserted. This means that such a position n is chosen so that for the set I identified by n holds $I \subset s \cup \{root[T]\}$ and I contains the maximum possible number of words. To achieve this, the algorithm traverses T using depth first search. The output of the algorithm is an indication that a node n was found that identifies a set $I = s \cup \{root[T]\} = words[p]$. ChooseBestTrieNode updates three global variables: *tree_x*, *tree_depth* and *tree_rest*. The first points to the best node n, found by the successive application of ChooseBestTrieNode for each candidate trie, (either if it identifies a set $I = words[p]$ or not), while the second identifies the depth of *tree_x*. The third is a set with the words in *rest* that do not exist in I. If such a node n is found such that its depth is greater than *tree_depth*, then *tree_x* is set to point to n, *tree_depth* is set to the depth of n and *tree_rest* is set to the set I identified by n. *tree_x* is also set if n is such that $I = words[p]$. In such a case, *tree_rest* $= \emptyset$ and we are sure that p will be inserted into n. Lines 1 to 15 initialize the variables. If no other candidate trie hash been found yet, lines 4 to 8 initialize *tree_x*, *tree_depth* and *tree_rest*. Lines 9 to 15 are executed in the case *rest* is empty. In that case, the root of T is the node we are seeking. Lines 16 to 59 consist the main loop of the algorithm. The traversal of T is done in this loop and node x at depth represented by the variable *depth* is examined in each iteration. Lines 18 to 27 examine the case x is a leaf. If this holds, we search for common words between $remainder[x]$ and *rest*. In the case that $remainder[x]$ and *rest* contain exactly the same words, we know that

p is going to be inserted in x, so we update the variables accordingly. If this is not true, then we examine the case the sum of the number of common words and current depth is bigger than *tree_depth*. In that case, we update the global variables, because, in the case p will be inserted here, it will be inserted in depth equal to this sum (understanding the way HashTrieInsert works will help to understand why this happens). If x is not a leaf, lines 29 to 51 are executed. Every child of x is searched for a word included in *rest*. When one is found, this child becomes node x and the procedure continues by searching its children or remainder. Lines 54 to 57 are executed in the case we examined all of x's children or remaining words. These lines make x to return one level up to the trie, so that we can search the rest of its father's children. The main loop (lines 16 - 59) is terminated when all of the possible positions in T where p could be inserted are searched. When this happens, if a better node than *tree_x* was found to insert p, then the global variables have been updated to values concerning that node.

The algorithm that matches a document d with a set of profiles stored in a trie hash table is named HashTrieMatch and is presented in Figure 4.16. It traverses each trie in a breadth first search way. This algorithm is almost the same as the one used in [23] for matching. The HashTrie Method utilizes two data structures for the matching of a document d: The occurrence table (OT or $OT(d)$) and distinct word list (DWL of $DWL(d)$) of d. They both contain all the distinct words met in d. While OT is implemented as a hash table with words as keys (so, search of a word is fast), DWL is implemented as a list (so, iteration through d's words is fast).

HashTrieMatch uses a queue Q to temporarily store nodes to be searched. Lines 1 to 7 create the occurrence table and distinct word list of d and initialize Q with the roots of the tries that may contain matching profiles. Lines 8 to 21 search the tries. Lines 10 to 15 search the current node for children that may contain matching profiles and add these children to Q. Lines 16 to 20 examine if the words of the word set identified by the current node exist in the occurrence table of d. If this is true, the profiles contained in the current node are added to the success list.

Next, we calculate the complexities of the data structures and algorithms used in the HashTrie method. The parameters used in the complexity calculations of HashTrie and subsequent algorithms are presented in Table 4.2.

In the presentation of complexity bounds in the rest of this work, if A is a set then $|A|$ will denote its cardinality. For the HashTrie complexity computations, we use the following notation: Let P be a set of profiles and I_P the set of identifying subsets in P. Let K be the number of nodes of all tries in the profile hash table. Thus, if we store the profiles of P in a trie hash table H, the number of nodes of all tries in H is equal to K and the number of leaves of all tries in H is at most $|I_P|$. To understand this, consider the case where all the profiles are stored each one in a leaf of a trie (one or more profiles per leaf). Then, each profile's (unique) identifying subset corresponds to exactly one leaf and each leaf

algorithm HashTrieMatch
input: a document d, a profile Hash Table H
output: success_list (list of matching profile identifiers)

```
1    create the Occurrence Table and Distinct Word List of $d$
2    $Q \leftarrow \emptyset$
3    for each word $w$ in $DWL(d)$ do
4        if there exists a trie $T$ in $H$ with root node $x$ that contains $w$ then
5            enqueue($Q$,$x$)
6        end if
7    end for
8    while $Q \neq \emptyset$ do
9        $x \leftarrow$ dequeue($Q$)
10       for each pair $(u, ptr\_y)$ in $children[x]$ do
11           if word $u$ exists in $OT(d)$ then
12               let $y$ be the node of $T$ pointed to by $ptr\_y$
13               enqueue($Q$,$y$)
14           end if
15       end for
16       if $profiles[x] \neq \emptyset$ then
17           if $remainder[x] = \emptyset$
                 or all words of $remainder[x]$ exist in $OT(d)$ then
18               $success\_list \leftarrow success\_list + profiles[x]$
19           end if
20       end if
21   end while

end algorithm
```

Figure 4.16: The algorithm used for matching a document with a trie hash table

Symbol	Parameter
N	Number of profiles
S	Maximum profile size
D	Document size
L	Maximum number of letters (characters) in a word
V_d	Document vocabulary
V_p	Profile vocabulary

Table 4.2: Parameters used in complexity calculations

corresponds to exactly one identifying subset (the one of the profiles it stores). So, the number of the leaves is equal to $|I_P|$. In the (more common) case where one or more profiles are stored in a non-leaf node (which means that their identifying subsets are subsets of other identifying subsets), then the number of leaves is smaller than $|I_P|$. Also, note that generally $K \geq |I_P|$, as a node n may exist without containing any profiles (its identifying word is a word common in two or more profiles that have one or more different words). In that case, n represents no identifying subset. On the other hand, each identifying subset is represented by a node (since the profile with that subset is stored in a node). Finally, let M be the maximum number of children in a node of a trie of H. During the time complexity calculations, we assume that the insertion or lookup of a trie in H takes $O(1)$ time. This is realistic, as H is implemented using Open Addressing with Double Hashing [13]. Moreover, its load factor α is less than 0.1 and its size is a prime number. These also hold for all other hash table instances described in this section as well. Hash table insertion and search speedup is achieved with these techniques.

4.1.1 Space Complexity

The space needed for the storage of profile identifiers is $O(N)$. The space required by all nodes is $O(K)$ and the total number of word-keys at each node is $O(K)$. In order to calculate the number of words that the trie-like structures need to store the profiles, we have to add the number of words in the nodes to the number of words in the remainder list of each leaf. So the space needed for storing words in the trie-like structures would normally be $O((K-|I_P|)\cdot L+|I_P|\cdot S\cdot L) = O(L\cdot(K-|I_P|+|I_P|\cdot S))$. In order to reduce this size, we consider the fact that many words in the nodes are repeated, so the space needed for each word is $O(L\cdot R)$, where R is the number of repetitions of the word in the nodes. To avoid this, we use another data structure called *word_pool* in which we store all the words met in the inserted profiles. The words in the nodes (word-keys and words in remainders) are actually pointers to the respective words stored in *word_pool*. The space needed for *word_pool* is $O(L\cdot|V_p|)$. So the space needed for storing the words is $O(K-|I_P|+|I_P|\cdot S+L\cdot|V_p|) = O(K+S\cdot|I_P|+|V_p|\cdot L)$. Thus, in total the space required by HashTrie Method is:

$$O(N+K+K+S\cdot|I_P|+|V_p|\cdot L) =$$

$$O(L\cdot|V_p|+S\cdot|I_P|+2\cdot K+N)$$

4.1.2 Update Complexity

The algorithm ChooseBestTrieNode in Figures 4.14 and 4.15 traverses the trie T in a DFS way and finds the best node for the profile to be inserted. In the worst case, ChooseBestTrieNode examines all the nodes of all the tries, so its time complexity is $O(K)$.

The algorithm HashTrieInsert is listed in Figures 4.12 and 4.13. The loop of lines 7 to 21 takes $O(K)$ time. Line 24 takes $O(S^2)$, so do lines 25 and 26. The loop of lines 30 to 38 takes $O(S)$ time. Thus, the update complexity of HashTrieInsert is $O(K + S^2)$.

4.1.3 Filtering Complexity

The algorithm HashTrieMatch is listed in Figure 4.16. The loop of lines 3 to 7 take $O(min(D, |V_d|))$ time. The loop of lines 8 to 21 do a breadth-first search of each trie in H and examine $O(K)$ nodes. Line 17 takes $O(S)$ time. Thus, the total time for filtering of one document is $O(min(D, |V_d|) + K \cdot S)$. The time consumed for the construction of the occurrence table and distinct word list of the document must be added. This is $O(D)$, so the total time consumed for document matching is $O(min(D, |V_d|) + K \cdot S + D)$.

4.2 Experimental Evaluation

In this section, we proceed with the experimental evaluation of the HashTrie method and we compare it with the Tree method [37] as well as with a Brute Force algorithm.

4.2.1 The Neural Networks Corpus

The testset used for the evaluation of the algorithms, is based on a document corpus (referenced as *NN corpus*). The advantage of this method is that the profiles and documents used for evaluation are more realistic than test data created using randomly chosen words (as, considering a specialized corpus, someone would submit a profile containing words of the corpus area's terminology).

The *NN corpus* was initially created and processed by the group of Evaggelos Milios at Dalhousie University [1]. The documents were created from a set of research papers about Neural Networks. These papers were downloaded from *ResearchIndex* [1, 24], that is a digital library aiming to the dissemination of scientific information. The documents were downloaded as postscript files and converted into simple text. Any non-text information such as images and equations were wiped out, resulting in documents that contain simple text words. After this, a part-of-speech (POS) tagger [4] was applied to the documents, adding an indication about whether a word is a verb or a noun etc. This information were used in order to find the candidate multi-word terms for profile generation. The C-Value and NC-Value method proposed in [18] were applied in order to retrieve

[1] `http://www.cs.dal.ca/~eem`

Description	Value
Number of documents	10,426
Document vocabulary size	641,242
Maximum document size (words)	110,452
Minimum word size (letters)	1
Maximum word size (letters)	35

Table 4.3: Some characteristics of the NN corpus

Attribute	number of documents	% fraction of documents
TITLE	6523	63%
AUTHORS	6046	58%
ABSTRACT	9162	88%
BODY	8990	86%

Table 4.4: Number of documents containing each attribute

that terms. A multi-word term sets were the result of this processing. Some statistical information about the *NN corpus* are presented in Table 4.3.

The *NN corpus* was further processed by Theodoros Koutris [23] and Christos Tryfonopoulos [32]. A set containing the authors of the documents and another one containing the words in the abstracts were created during this procedure. The frequency that each author appears in the author set is proportional to the number of citations the author receives. The POS tags were removed from the documents, while four possible attributes were left: Author, Title, Abstract and Body.

The overall result of the *NN corpus* processing was documents containing free text organized in attributes, as well as datasets proper for profile generation. Statistical information on how many documents contain a specific attribute can be found in Table 4.4. Statistical information about how many documents contain a specific number of attributes can be found in Table 4.5.

4.2.2 Unit Sets Creation

All the profiles used to evaluate the algorithms for the boolean model are generated from the combination of three different unit sets created from the selection of words and multi-word terms that appear in the NN corpus documents. In this section, we describe these unit sets as well as the procedure followed for their creation. There is one unit set created from the modification of the multi-word terms of the corpus. There is also a unit set containing the last names of authors of the corpus documents. Finally, there is a unit set that contains

Number of attributes	number of documents	% fraction of documents
1	479	4.59%
2	3495	33.52%
3	1641	15.74%
4	4680	44.89%

Table 4.5: Percentage of documents containing one or more attributes

selected nouns from the abstracts of the documents.

In order to create the unit set with the multi-word terms, the original list of multi-word terms was sorted by C-value / NC-value. Then, the terms containing more than 5 words were cleared out, as they were noise created by the C-value / NC-value method. The discriminating power of terms with high C-value / NC-value is very low, so terms C-value / NC-value greater than an upper bound that we defined, were also rejected. Also, terms with C-value / NC-value lower than a bound are considered to be noise created by the conversion of the initial postscript files to text, so they were rejected too. The set of multi-word terms created with this method will be referred as MS.

The second unit set contains nouns taken from the abstracts of the documents and will be referred as NS. The indications created by the POS tagger were used to identify the nouns. Very frequent nouns would have small discrimination power and would not be used in profiles, while infrequent ones were noise. So, after the extraction, the nouns were sorted by frequency, an upper and lower threshold were set and nouns with frequencies greater than the upper threshold as well as the ones with frequencies below the lower threshold were removed. As a result, we created a set of words very likely to be used for searching in an information dissemination setting. The list of the initial nouns was chosen to contain nouns only from abstracts because a paper's abstract is an overview of the whole paper, a place where "keywords" that define the paper's subject are mostly met.

The last unit set contains the surnames of the authors of the documents. It will be referred as AS. There are 8832 distinct authors. Something we were aware of is that the average user would request papers published by specific authors more often than papers published by others. This is because there are scientists more or less active in a region or scientists whose work is considered to be more important than others'. So, these authors gain more visibility and consequently their work is more oftenly searched for. Thus, a collection of all the authors' names would not be enough by itself. What we do is to enter an author's name more than once in the author set.

In order to determine how many times an author a should appear in the author set, we apply the following method: Let N_i be the number of papers that contain a citation to an author i's paper. We define the probability:

$$P(a) = \frac{N_a}{\sum_{i \in Vauthor} N_i}$$

$P(a)$ is a metric of the popularity of author a and consequently, a good metric for the frequency of the appearance of a in a profile. So, an author a is repeated n times in the author set, where n is proportional to $P(a)$. This way, popularity information about each author is also contained in the author set. One thing we noticed during the author unit set creation is that $P(a)$ follows the Zipf distribution. This becomes logical since we realize that in most specialized scientific areas there are a few popular scientists and many that receive less popularity.

In this section we described the different unit sets used for boolean atomic profile generation as well as the process of their creation. More details can be found in [23, 32].

4.2.3 Boolean Atomic Profiles Generation

In this section we describe a method to generate profiles appropriate to evaluate algorithms and data structures that support boolean atomic profiles. The method presented here is similar to the one used in [32] for the evaluation of algorithms that implement the same functionality under $\mathcal{AWP}$.

A boolean atomic profile is a conjunction of words $w_1 \wedge w_2 \wedge \ldots \wedge w_n$, where w_i is a word. We form such a profile by concatenating one or more different units from the unit sets described in Section 4.2.2. The first step for the creation of an atomic profile is to decide how many units will take part in it. Let the number of these units be S. S is an integer randomly chosen from the interval $[1, S_{max}]$, where S_{max} is defined separately for each attribute in the attribute set.

In order to decide which unit sets will offer units to take part in the creation of an atomic profile, we use a selection probability for each of the unit sets. These selection probabilities for each of the sets of Section 4.2.2 are shown in Table 4.6. Using these probabilities, we decide whether a unit set will take part in the creation of the atomic profile. We are able to change the frequency that each unit set appears in profiles by properly adjusting the corresponding probability. We could make the units of a specific set appear in all the atomic profiles by setting the respective probability equal to 1. Similarly, we could prevent these units from appearing in any of the atomic profiles by setting the probability equal to 0. If a unit set is decided to take part, then one of its units is randomly chosen to be inserted in the atomic profile, using a uniform distribution. To select the units, a loop executes S iterations. In each iteration i, a unit of the atomic profile is created. The possibility to be selected by MS, NS or AS is shown in Table 4.6.

There are two possible ways of forming a profile: In the first of them (with which 75% of the profiles are constructed), a multi-word term from MS is selected

Participating unit sets	Selection probability	S_{max}	percentage of profiles
MS	0.8	2	25%
NS	0.2		
MS	0.8	3	50%
NS	0.2		
AS	1.0	2	25%

Table 4.6: Boolean atomic profile generation concepts

with a probability of 0.8 and a noun from NS with a probability of 0.2. The rest 25% of the profiles is constructed using one or two authors from AS. The concepts of boolean atomic profile generation are summarized in table 4.6.

4.2.4 Experiment Settings

Now that the profile generation techniques have been explained, it is time to present the results of simulations and to experimentally evaluate our algorithms. Several of their performance characteristics are revealed through this process.

All the tested algorithms were implemented in C++. The simulations were carried out on a Pentium 4 at 1.7GHz running Linux. The system had 1GB main memory. At the time of each simulation, no other user programs were running, while the process of each experiment was running with highest priority to avoid multitasking with operating system services.

Testing of the various algorithms is done according to the following procedure: Firstly, the profile set appropriate to test each algorithm is generated according to the procedure described in the respective section. After the generation of the profiles, we select 100 documents from the corpus in order to test filtering performance. Moreover, the results of matching are averaged, so testing with the whole document set is not necessary. The average number of words in each document was 971, while the average size of a document was 31KB.

To perform matching, the document is loaded into the main memory and its occurrence table is constructed. Matching time calculations include no overhead due to hard disk I/O. Also, the time consumed for the creation of the occurrence table and for its destruction after the matching process, is not added into matching time calculation, but they are calculated separately. So, match time calculations only include the time consumed by the various matching methods, excluding time needed for any memory allocation or deallocation. Moreover, concerning the various matching time calculations, the simulations only measured the time needed for matching, occurrence table creation and destruction. The time needed for document download as well as the time needed to send the document to interested users (whose profiles are found to match the document), are not taken into account.

Due to many unstable factors, errors may occur in time calculations concerning matching. In order to prevent this, the matching process for each document is repeated eight times and the measured times are averaged. These averages are considered to be the time consumed for matching and occurrence table creation and destruction.

Performance of profile database update is also evaluated. Specifically, we compare the time needed for the insertion of a number of profiles for various database sizes, between the various algorithms. One could argue that the problems that appear in matching time evaluation may also appear in these calculations. But, insertion time calculates the time needed for the insertion of a great number of profiles, so what is really computed is a statistical value.

Another counted size, is the memory allocated by each algorithm for various profile database sizes. This size is compared with respective measurements taken for Brute Force algorithm, showing up the memory size overhead required for each method. It gives us an idea about the price of matching speed.

4.2.5 Simulation Results

In this section, we present the data that came out of the analysis of experimental data for HashTrie as well as some other algorithms. These data were the result of simulations ran with the settings described in Section 4.2.4. The efficiency of HashTrie was measured and compared to that of the Tree method [37] and to that of a Brute Force method. The profiles for these simulations were generated with the method described in Section 4.2.3.

The memory space needed for each of the algorithms is shown in Figure 4.17. We can see that Tree and HashTrie, not only need more space than Brute Force, but also, the space needed by HashTrie and Tree, increases more rapidly with the profile database size than the space needed by Brute Force. Moreover, Tree is the worst algorithm of the three considering memory space, as it allocates more memory than the others.

Speed is the advantage of HashTrie and Tree. The average match time for each of the algorithms is presented in Figure 4.18. Filtering time for HashTrie and Tree is much less than filtering time of the Brute Force algorithm. Also, the time needed by Brute Force increases more rapidly with profile database size. Between HashTrie and Tree, HashTrie is clearly a winner.

A metric that is also evaluated through matching time, is the throughput of the system. Throughput represents the computational capability of our algorithms. It is of great importance, as a big throughput minimizes the possibility of system overload, while maximizing quality of service. In Figure 4.19, the processing capability of each algorithm is presented. The entry *Algorithm-xM* means "performance of Algorithm with a database size of x millions of profiles". We can clearly see once more that HashTrie is the best of the three algorithms. What is most impressing is that even with a database size of three times the database

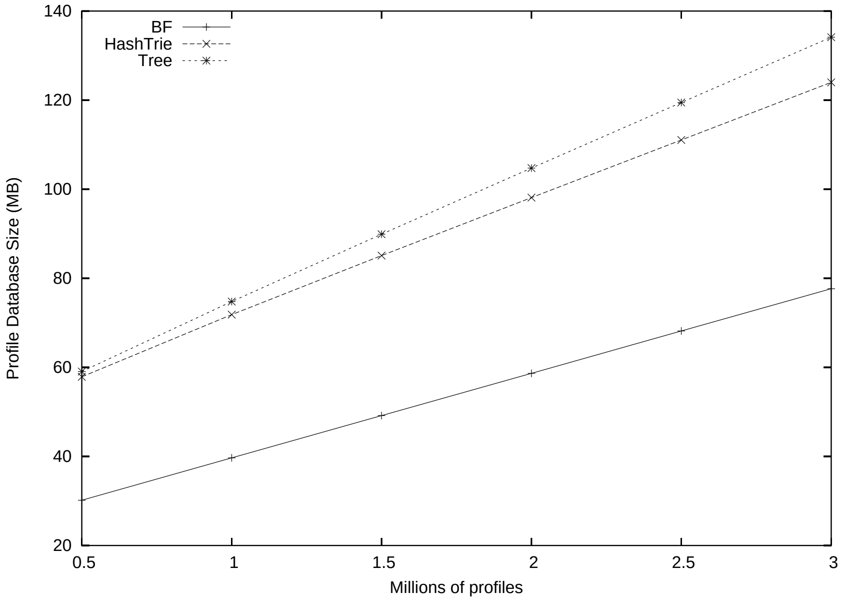


Figure 4.17: Effect of database size in allocated memory space for the Boolean Model

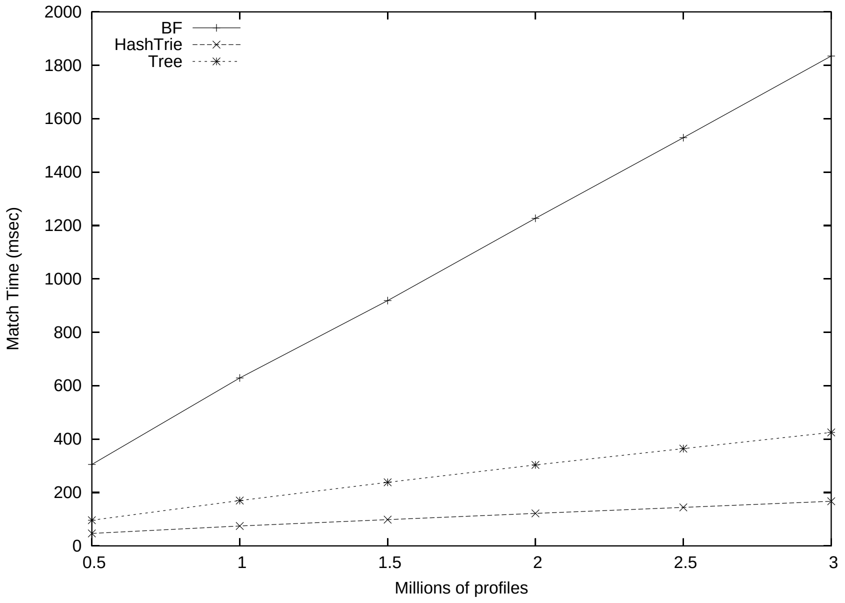


Figure 4.18: Effect of database size in filtering time for the Boolean Model

size of Tree, HashTrie is still faster than Tree.

The Tree algorithm uses alphabetical rules for the selection of the word which will be used to create a new trie or node during insertion. This means that during profile insertion, whenever a new node is needed to be constructed or a new trie is needed to be indexed in a profile hash table, the alphabetically smallest word of the profile (from the words in *tree_rest* - see Section 4.1) is selected. So, moving from the root of a trie towards its roots, we meet the words of the profiles in alphabetical order. HashTrie picks the "leftmost" word of *tree_rest* to create a new node. For example, if we wish to insert the profile $Neural \wedge Networks \wedge Corpus$ in a node containing *Networks*, then *Neural* is selected (as *tree_rest* contains *Neural, Corpus*) as the key of the new node. There were two heuristics used for the selection of the word which will be used to create a new trie or node during insertion in HashTrie. These heuristics utilize ranking information [37]. Using the first of them (called *rank* heuristic), the word in *tree_rest* that is most popular among the currently stored profiles is used to index the new node or trie. Using the second one (called *irank* heuristic), the least popular one is used. In other words, using the *rank* heuristic, we expect popular words to be inserted near the roots of tries (which we expect to result in wide and short tries), while with *irank*, we expect that they will be inserted far from the roots (which will result in tall and thin tries). In the former case, we expect that the profile database size will be minimized, as there will be better clustering of profiles, as more common words are expected to be found among profiles. In the latter case though, we expect that the filtering time will be reduced, as less tries are expected to be examined (many profiles will be discarded without needing to examine in depth the tries in which they are stored). Ranking information is extracted from the inserted profiles. For each word, its popularity is equal to the number of inserted profiles containing it. This means that ranking information is updated in every insertion.

We experimented with these heuristics. The results for the profile database size are shown in Figure 4.20. We can see that all versions of HashTrie need about the same memory, which is considerably less than the memory needed by Tree. The heuristics did not have any serious impact on the profile database size.

The results for matching time are shown in Figure 4.21. The results this time correspond to what we expected. We see a significant speedup using the *irank* heuristic. On the other hand, using the *rank* heuristic slows down the system. In any case though, HashTrie is fastest than Tree.

Having seen the results of heuristic evaluation, we decided to apply the *irank* heuristic (which was shown to speedup the system with almost no cost) to the Tree algorithm. This means that the resulting algorithm will still search for common prefixes among the profiles, but now, instead of sorting the profiles alphabetically to determine common prefixes, we sort them by inverse ranking frequency. The results of this algorithm's comparison with HashTrie with the *irank* heuristic are following.

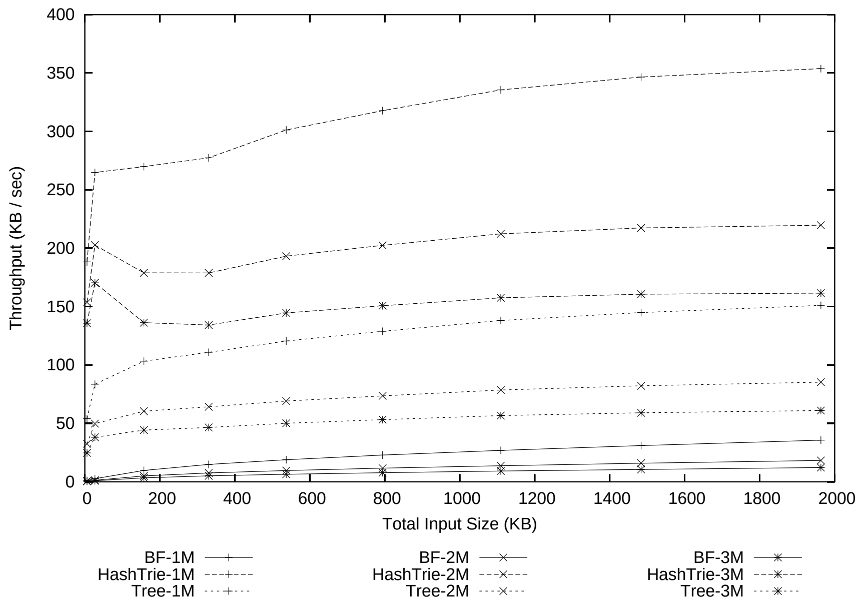


Figure 4.19: Throughput of algorithms for the Boolean Model

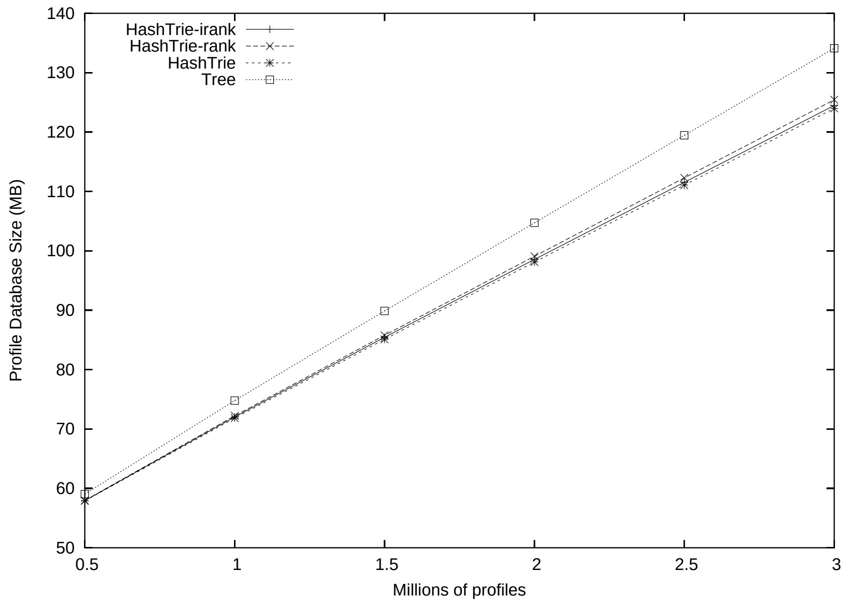


Figure 4.20: Allocated memory with the use of heuristics for the Boolean Model

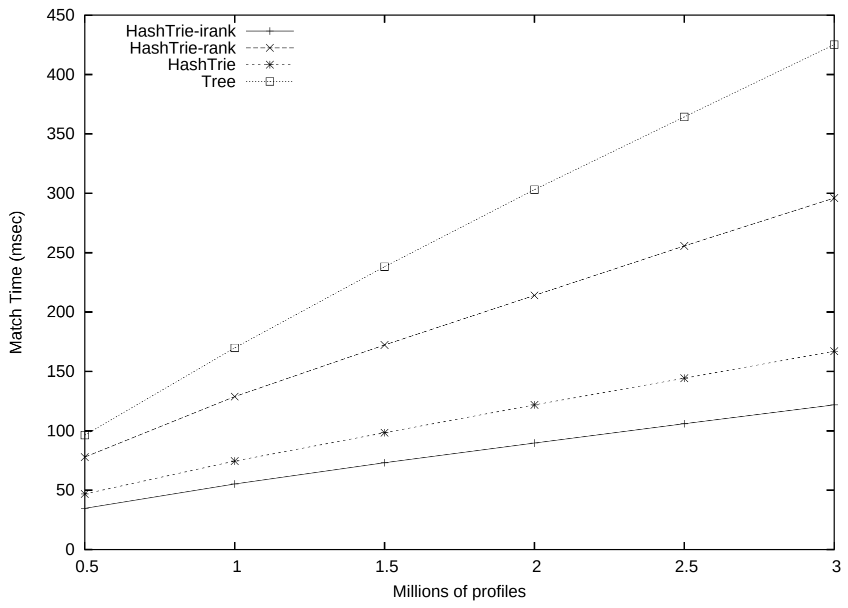


Figure 4.21: Filtering time with the use of heuristics for the Boolean Model

In Figure 4.22, we can see the results of the comparison between the space needed by the two algorithms. We see that Tree with *irank* needs about the same space with the "simple" Tree, which is more that the space needed by HashTrie. But, the time consumed by Tree (with *irank*) matching is less than the time needed by the respective version of HashTrie, as we can see in Figure 4.23. A reason for this could be that HashTrie changes word ordering in the profiles in order to achieve better clustering. Tree does not do this, it simply looks for common prefixes using the existing ordering. So, word ordering in the profile has a greater impact in Tree than in HashTrie. That's why *irank* performs even better with Tree.

4.3 Conclusions

Methods appropriate to support data models and languages under the boolean model are presented in this chapter. The HashTrie method is proposed and its space and time complexities are calculated. Moreover, the performance of this method is experimentally evaluated and compared to the performance of other methods for the boolean model, such as the Tree method. A Brute Force algorithm was also implemented for comparison purposes. Finally, various heuristics based on ranking were tested. It was shown that bad use of ranking may result in undesired slowdown of the system. On the other hand, if we properly exploit ranking, we can gain many benefits. Use of ranking information can also change

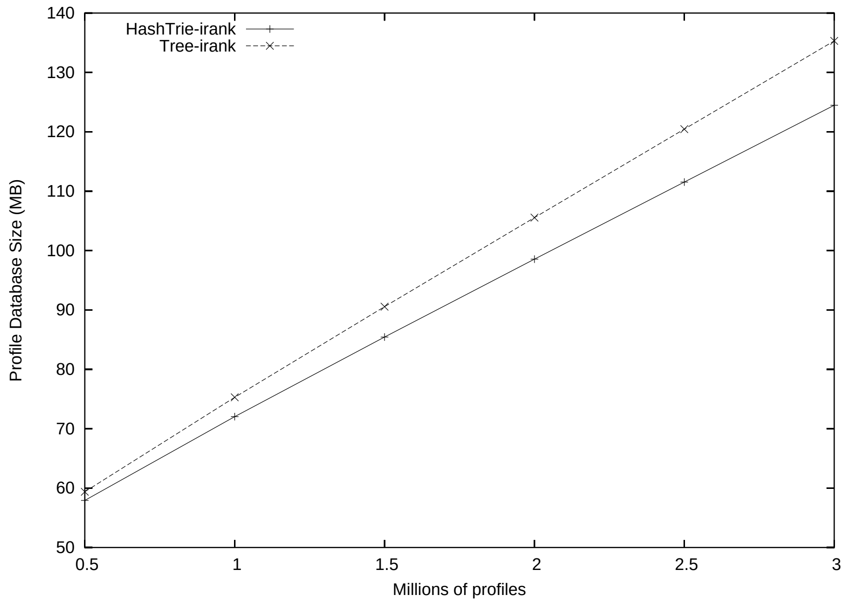


Figure 4.22: Allocated memory with the use of *irank* heuristic for the Boolean Model

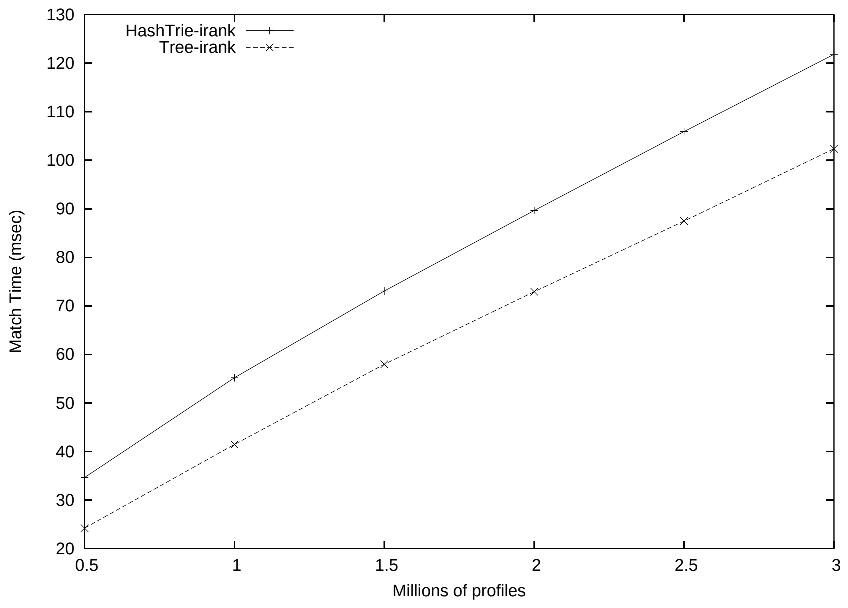


Figure 4.23: Filtering time with the use of *irank* heuristic for the Boolean Model

the effectiveness ordering of two methods.

Chapter 5

Methods for $\mathcal{AWP}$ Support

In this chapter, we propose a method to support profiles under $\mathcal{AWP}$. We analytically present the data structures and algorithms of the method and we conclude with its evaluation and experimental comparison with other algorithms.

5.1 Proximity Storage and Evaluation

In this section we present the Proximity method used for proximity evaluation. This method is used in conjunction with HashTrie for the support of boolean profiles with proximity operators.

Under the Proximity method, proximity formulas contained in a profile are stored in an array. Each slot of this array is actually a structure representing a proximity formula with n words. This structure consists of the following fields:

- *words* : An array that contains the words taking part in the proximity formula, in the order they appear in the formula. It size is equal to n.

- *min_distance* : An array each slot i of which represents the minimum required distance between $words[i]$ and $words[i+1]$. Its size is equal to $n-1$.

- *max_distance* : An array each slot i of which represents the maximum required distance between $words[i]$ and $words[i+1]$. Its size is equal to $n-1$.

Example 5.1 *The proximity formula*

$$Artificial \prec_{[0,0]} Intelligence \prec_{[3,4]} Neural \prec_{[0,2]} Network$$

is represented with the following structure:

$words:$	$Artificial$		$Intelligence$		$Neural$		$Network$
$min\_distance:$		0		3		0	
$max\_distance:$		0		4		2	

To fully support proximity profiles, we use a proximity table PT. PT's size is equal to the number of profiles N. Each of its slots contains the array of proximity formulas of the respective profile. If a profile p has no proximities, the slot of PT corresponding to p is empty.

Example 5.2 *Let us assume three profiles p_1, p_2 and p_3 with the proximity formulas:*

$$p_1 : Artificial \prec_{[0,0]} Intelligence$$

$$p_2 : Speech \wedge Recognition \wedge System$$

$$p_3 : Markov \prec_{[0,2]} Chain \prec_{[4,8]} Monte \prec_{[0,3]} Carlo$$

Notice that p_1 has one proximity formula, p_2 has no proximity formulas, while p_i has two proximity formulas. We are inserting these profiles into a proximity table PT. The proximity table will look like the one in Figure 5.1.

For the evaluation of a proximity formula with a document, an occurrence table (OT) like the one used in HashTrie is used. Each record in OT is indexed using the word and it also contains a list of the positions in the document where the word is found. These positions are expressed in terms of word indexes (word in position 0 is the first word in the document, word in position 1 is the second etc). The positions are stored in the list in ascending order.

Example 5.3 *Some of the records of an OT of a document is shown in Figure 5.2. We can see that the word "Artificial" appears in positions 3 and 15 etc.*

The algorithm used for matching a document with a proximity formula is named EvaluateProximityFormula and is given in Figure 5.3. For each word w_i in the occurrence table of the document, the algorithms chooses w_{i+1}'s positions in the document that satisfy the proximity formula's constraint between w_i and w_{i+1}. Only these positions qualify to be checked for satisfaction with the positions of w_{i+2}. If there are qualifying positions of the last word in the proximity formula, the algorithm returns True (which means that the proximity formula is satisfied). The algorithm returns False as soon as a w_i with no qualifying positions is found. $positions[w_i]$ are taken from the occurrence table of a document, with a hash table search. We assume that all the words in the proximity formula exist in the occurrence table. By the time we study the unification of the Proximity

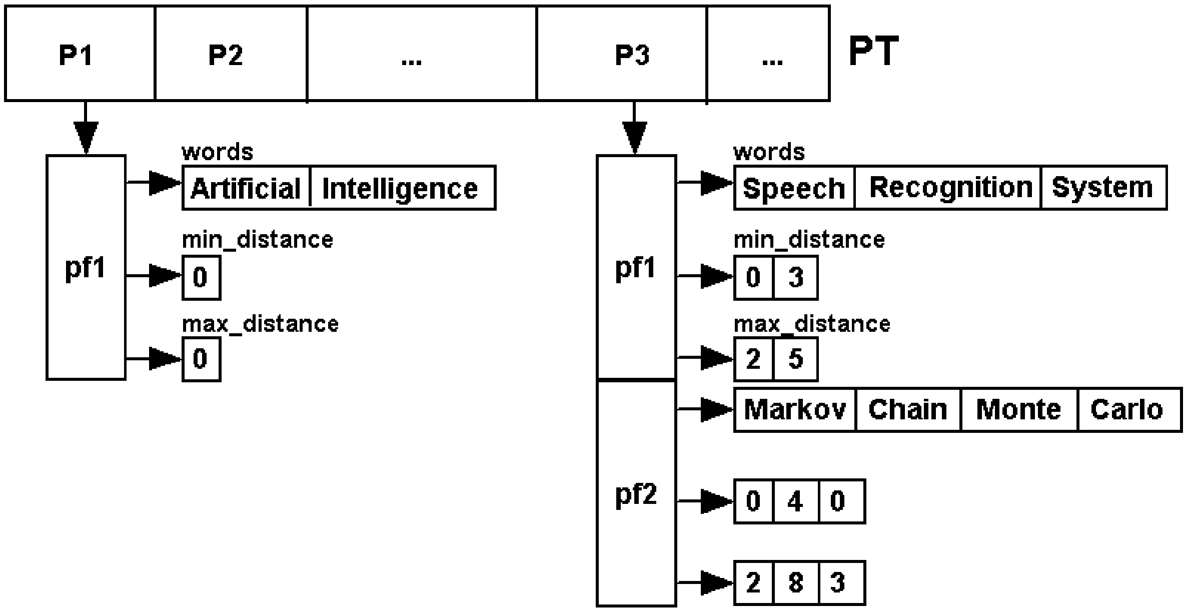


Figure 5.1: A proximity table *PT* with some profiles in it

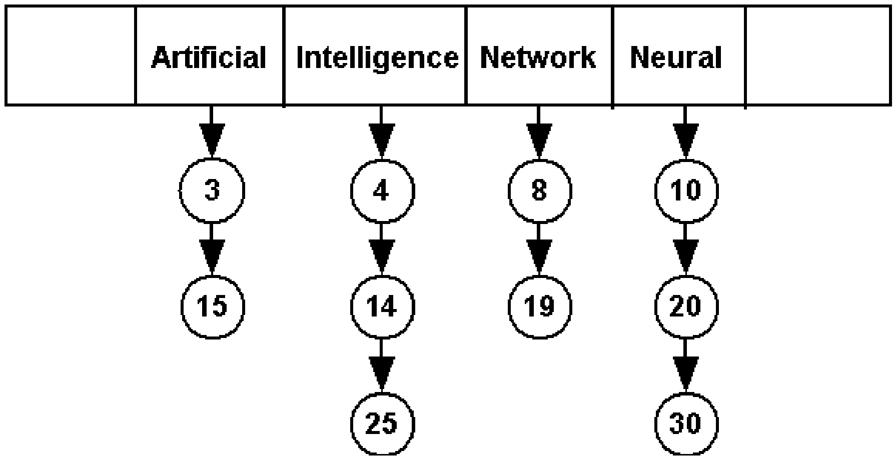


Figure 5.2: Some of the records of an OT used for proximity evaluation

algorithm EVALUATEPROXIMITYFORMULA
input: a proximity formula pf, a document occurrence table OT
output: *matches* (indication whether the document represented by OT matches pf)

```
1    $matches \leftarrow False$
2    $temp\_positions \leftarrow positions[first\ word\ in\ words]$
3    for each word $w_i$ in $words[pf]$ do
4        if $temp\_positions$ is not empty then
5            if $w_i$ is the last of $words[pf]$ then
6                $matches \leftarrow True$
7            else
8                $candidate\_positions\_list \leftarrow \emptyset$
9                fetch the first position from $temp\_positions$ into $p_i$
10               fetch the first position from $positions[w_{i+1}]$ into $p_{i+1}$
11               while there exist words in $temp\_positions$ and $positions[w_{i+1}]$ do
12                   $diff \leftarrow p_{i+1} - p_i - 1$
13                   while $diff \geq min\_distance[w_i]$ and $diff \leq max\_distance[w_i]$
                         and there exist positions in $positions[w_{i+1}]$ do
14                       add $p_{i+1}$ to $candidate\_positions\_list$
15                       fetch next position from $positions[w_{i+1}]$ into $p_{i+1}$
16                       $diff \leftarrow p_{i+1} - p_i - 1$
17                   end while
18                   while $diff < min\_distance[w_i]$
                         and there exist positions in $positions[w_{i+1}]$ do
19                       fetch next position from $positions[w_{i+1}]$ into $p_{i+1}$
20                       $diff \leftarrow p_{i+1} - p_i - 1$
21                   end while
22                   if $diff > max\_distance[w_i]$
                       and there exist positions in $positions[w_i]$ then
23                     fetch next position from $temp\_positions$ into $p_i$
24                   end if
25               end while
26               $temp\_positions \leftarrow candidate\_positions\_list$
27           end if
28       end if
29   end for

end algorithm
```

Figure 5.3: The algorithm used for evaluation of a proximity formula

method with the HashTrie method, we will understand why this is a reasonable assumption.

The EVALUATEPROXIMITYFORMULA algorithm takes a proximity formula pf and an occurrence table OT as input. Lines 1 and 2 initialize the output variable and store the positions of the first word of pf in *temp_positions*. *temp_positions* is the place where the qualifying profiles of w_i are stored at each iteration. All of the positions of the first word are assumed to be qualifying (since there is no constraint considering a word preceding the first word). The loop of lines 3 to 29 is executed for every word w_i in pf. Line 4 checks whether or not we have reached to a point where no qualifying positions exist. If we have, the algorithm returns immediately. If we haven't, then line 5 checks if w_i is the last word of pf. If it is, then pf is satisfied by OT, so the output variable is set appropriately in line 6. Lines 7 to 27 examine the symmetric case: They determine the qualifying positions of w_{i+1}. Line 8 initializes the list of the candidate positions of w_{i+1}. Line 9 initializes p_i, the currently examined position of w_i. The same is done for p_{i+1}, the currently examined position of w_{i+1} in line 10. The loop of lines 11 to 25 find the qualifying positions of w_{i+1} and insert them into the candidate positions list. Line 12 calculates the distance between p_i and p_{i+1}. Lines 13 to 17 are executed while positions of w_{i+1} that satisfy p_i are found. These positions are inserted into the candidate positions list without any further examination. p_{i+1} advances to the next position of w_{i+1} after each iteration. The loop of lines 18 to 21 is executed while p_i and p_{i+1} are too close to each other. While this happens, p_{i+1} advances to the next position of w_{i+1} in the document, until a satisfying position is found. Lines 22 to 24 examine the case where p_i and p_{i+1} are too far away from each other. In that case, p_i advances to the next position of *temp_positions*. Careful study of the algorithm will make clear that this is done until a satisfying position p_i is found. Finally, line 26 initializes *temp_positions* to be used in the next iteration.

As mentioned earlier, a profile p contains zero, one or more proximity formulas. Given that all of p's words exist in a document d with occurrence table OT, to be able to decide whether d matches p, we have to determine whether all of the proximity formulas of p are satisfied by d. What we do is to apply the EVALUATEPROXIMITYFORMULA algorithm to each of the proximity formulas of p. If at least one of them is not satisfied, then d does not match with p. The rather simple algorithm used to perform this calculation is named EVALUATEPROXIMITY and is presented in Figure 5.4.

5.1.1 Space Complexity

For the complexity calculations of the Proximity method, let us assume that each profile has an average of f word formulas and that each word formula contains an average of m words. So, each proximity formula needs $O(L \cdot f)$ space, while a profile takes $O(L \cdot f \cdot m) = O(L \cdot S)$ space. Consequently, the proximities of all the profiles in PT would normally need $O(N \cdot S \cdot L)$ space. In an effort to decrease the space complexity, we once more use a *word_pool* data structure in which we store all the words met in the stored proximity formulas. Each of the contents of the *words* array of each proximity formula is actually a pointer to the respective word stored in *word_pool*. As a result, the space used to store PT becomes $O(N \cdot S)$. The space needed for *word_pool* is $O(L \cdot |V_p|)$, so the total space needed for the storage of the profiles' proximity information is $O(N \cdot S + L \cdot |V_p|)$.

5.1.2 Update Complexity

A proximity formula needs $O(f)$ time to be saved, so all the proximity information of a profile needs $O(m \cdot f) = O(S)$ time to be saved.

5.1.3 Filtering Complexity

The algorithm EVALUATEPROXIMITYFORMULA is listed in Figure 5.3. The loops of lines 13 to 17 and 18 to 21 are mutual exclusive: Execution of one or more iterations in the first, mean that the evaluation expression of the second is False and vice-versa. So, the time complexity of these combined loops is $O(R)$, where R is the average number of repetitions of a word in a document. The loop of lines 11 to 25 executes $O(R)$ iterations, so its time complexity is $O(R^2)$. The loop of lines 3 to 29 executes $O(f)$ iterations, so the time complexity of that loop and of the algorithm if $O(f \cdot R^2)$. Thus, The time complexity of the evaluation of proximities in a single profile is $O(m \cdot f \cdot R^2) = O(S \cdot R^2)$. Evaluation of the proximities in all the profiles is done in $O(N \cdot S \cdot R^2)$ time.

5.2 Combining HashTrie with Proximity

In this section we study how the HashTrie method described in Chapter 4 can be combined with the Proximity method described in section 5.1 in order to support boolean profiles with proximity operators. The model supported by this combination is a subset of $\mathcal{WP}$ described in Chapter 3.

Example 5.4 *A profile supported by the method of this section is:*

$$Artificial \wedge Intelligence \wedge Markov \prec_{[0,2]} Chain \prec_{[4,8]} Monte \prec_{[0,3]} Carlo$$

The semantics of such a profile are that all words in profile must be found in a document in order for the document to be able to match the profile. Additionally, the proximity formula(s) of the profile must be satisfied, or else the document does not match the profile.

In order to support boolean queries with proximity operators, we use the trie hash table H used in the HashTrie method and the proximity table PT used in the Proximity method. The shape of such data structures is shown in Figure 5.5. Profile insertion and matching rely on the property that the existence of all the words contained in a profile p is required in a document d in order for d to be able to match p. Proximity is an additional constraint that must be satisfied for p to qualify as a matching profile.

Insertion of a profile p is done in two stages. Firstly, we create a distinct word list containing the words of p, without considering any proximity operators. Using this distinct word list, we insert p into H. Secondly, we identify the distinct proximity formulas of p and we create an array (each slot of which contains a structure like the one described in section 5.1) with them. We insert this array into the PT slot that corresponds to p. This means that no indexing exists for proximities. Instead, the words that take part in proximities are indexed in the tries used by HashTrie.

One obvious algorithm for matching is to use the algorithm presented in Figure 5.6. As we can see, this algorithm calls HashTrieMatch to get the candidate matching profiles and then constructs an occurrence table to perform proximity matching. But HashTrieMatch has already constructed, used and destructed an occurrence table of the document (as shown in Figure 4.16). In order to avoid multiple occurrence table constructions, we utilize a new algorithm to match a document with boolean profiles that may also contain proximities. The algorithm, named HashTrieProximityMatch is presented in Figure 5.7. It is slightly different from the HashTrieMatch algorithm listed in Figure 4.16. What HashTrieProximityMatch does is to determine the candidate profiles to be inserted into the success list, by the use of the HashTrie data structures. Next, it evaluates the proximity constraints of the candidate profiles and inserts into the success list these profiles that all of their proximities are satisfied.

HashTrieProximityMatch uses a queue Q to temporarily store nodes to be searched. Lines 1 to 8 create the occurrence table $OT(d)$ (with word position information) and distinct word list $DWL(d)$ of d and initialize Q with the roots of the tries that may contain matching profiles. Lines 9 to 26 search the tries. Lines 11 to 16 search the current node for children that may contain matching profiles and add these children to Q. Line 18 examines if the words of the word set identified by the current node exist in d. If this is true, the proximities of the profiles contained in the current node are examined in lines 19 to 23. The profiles the proximities of which are satisfied by d are added into the success list.

5.2.1 Space Complexity

For the complexity calculations, we use the same notation used in the complexity calculations of the various HashTrie algorithms. The space required for the storage of the profiles is the space required for the HashTrie hash table H and

```
algorithm EvaluateProximity
input: a profile $p$, a document occurrence table $OT$
output: $matches$ (indication whether the document represented by $OT$ matches $p$)

1    for each proximity formula $pf_i$ contained in $p$ do
2        if EvaluateProximityFormula($pf_i$,$OT$) = $False$ then
3            return $False$
4        end if
5    end for
6    return $True$

end algorithm
```

Figure 5.4: The algorithm used for the evaluation of a profile's proximities

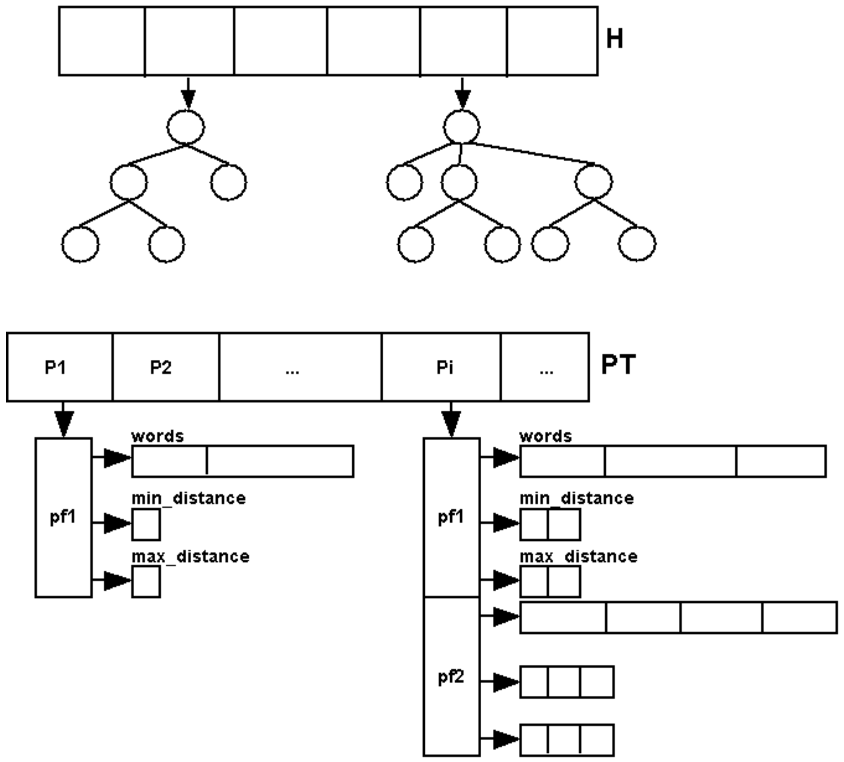


Figure 5.5: Shape of data structures able to store boolean atomic profiles with proximities

for the Proximity table PT. We also utilize the *word_pool* data structure in the same way as it is used in the standalone HashTrie and Proximity methods. Note that we only need one *word_pool*, since the vocabulary of the profiles is common, no matter whether the profiles contain proximity operators or not. So, according to Sections 4.1.1 and 5.1.1, the space needed for the storage of boolean profiles with proximity operators is:

$$O(L \cdot |V_p| + S \cdot |I_P| + 2 \cdot K + N + N \cdot S) =$$

$$O(L \cdot |V_p| + S \cdot (|I_P| + N) + 2 \cdot K)$$

5.2.2 Update Complexity

According to Section 4.1.2 and Section 5.1.2 and since the insertion of a profile consists of an execution of HashTrieInsert followed by the insertion of the proximity information of the profile, the update complexity is:

$$O(K + S^2 + S) = O(K + S^2)$$

K is the variable defined in Section 4.1.

5.2.3 Filtering Complexity

The algorithm HashTrieProximityMatch is listed in Figure 5.7. The loop of lines 4 to 8 takes $O(min(D, |V_d|))$ time. The loop of lines 9 to 26 do a breadth-first search of each trie in H and examine $O(K)$ nodes. Line 18 takes $O(S)$ time. According to section 5.1.3, the loop of lines 19 to 23 takes $O(S^2 \cdot R^2)$ time. Thus, the total time for filtering of one document is $O(min(D, |V_d|) + K \cdot S^2 \cdot R^2)$. The time consumed for the construction of the occurrence table and distinct word list of the document must be added. This is $O(D)$, so the total time consumed for document matching is $O(min(D, |V_d|) + K \cdot S^2 \cdot R^2 + D)$.

5.3 Adding Attribute Support

In this section, we extend the data structures and data algorithms of Section 5.2 in order to support attribute based profiles and documents under the model $\mathcal{AWP}$.

Support for attribute based profiles means that a profile may consist of one or a conjunction of more atomic profiles. Each atomic profile refers to an attribute and can be a boolean profile with proximities. The document model supported by this method consists of attributes containing free text. Each document has to be consisted of one or more parts. These parts are the attributes of the document.

Example 5.5 *An example of a profile p_1 supported by the method of this section is:*

$$\begin{aligned}(AUTHOR \sqsupseteq\ & Manolis \prec_{[0,0]} Koubarakis) \wedge \\ (BODY \sqsupseteq\ & Artificial \wedge Intelligence \wedge Markov \prec_{[0,2]} Chain \prec_{[4,8]} Monte \prec_{[0,3]} Carlo)\end{aligned}$$

This profile consists of the atomic profile:

$$p_{1_1} \sqsupseteq Manolis \prec_{[0,0]} Koubarakis$$

that refers to attribute $AUTHOR$ and

$$p_{1_2} \sqsupseteq Artificial \wedge Intelligence \wedge Markov \prec_{[0,2]} Chain \prec_{[4,8]} Monte \prec_{[0,3]} Carlo$$

that refers to attribute $BODY$. The semantics of p_1 are that the constraints concerning all of the profile's attributes must be satisfied by a document d in order for the document to match the profile (so, d must contain text in both AUTHOR and BODY attributes). In other words, p_{1_1} and p_{1_2} must be satisfied by the the text of the corresponding attributes of d in order for d to match p_1.

All the other methods we have so far discussed, consider a document containing free text only. In Section 5.2, we presented a method to support a language with boolean expressions and proximities. In order to support attributes, we employ the algorithms and data structures used in that method.

The data structures of the HashTrie method with proximities are a trie hash table H and a proximity table PT. Let us consider a data structure AT that contains both H and PT. Each AT is able to store an atomic profile that refers to an attribute of a profile. The main data structure used is a table ATT of AT. Its size is equal to the total number of attributes of profiles currently inserted in the system. ATT can grow to support more attributes that belong to the attribute universe of our system. Each attribute is encoded in a number. Each slot of ATT contains all the necessary data structures to support $\mathcal{AWP}$ atomic profiles referring to the corresponding attribute. A view of how an ATT table looks like is shown in Figure 5.8. There is also a table *total*, with size equal to the number N of the profiles currently inserted in the system. For each profile, there is exactly one record in *total*. Each record of *total* contains the number of atomic profiles that in conjunction form a specific profile. During filtering, a table *count* is used along with *total* to find the matching profiles. While each slot of *total* contains the number of attributes of a profile, the respective slot of *count* contains the number of attributes of that profile that match with the respective attributes of the document.

Example 5.6 *Assuming that no other profile exists in the system, ATT with only the profile of Example 5.5 stored looks like the table of Figure 5.9, which also presents the* total *table.*

Insertion is performed in the same way as in the HashTrie method with proximities. For each atomic profile p_i of the inserted attribute profile p, a distinct word list containing p_i's words is created, without considering any proximity operators. Using this distinct word list, we insert p_i into the HashTrie hash table H of the proper slot AT of ATT. Secondly, we identify the distinct proximity formulas of p_i and we create an array (each slot of which contains a structure like the one described in section 5.1) with them. We insert this array into AT's PT slot that corresponds to p_i.

In the attribute enabled matching process, we use a slightly different version of HashTrieProximityMatch (presented in Figure 5.7), called AwpMatch. Its only difference from HashTrieProximityMatch is that this algorithm does not operate on any *success_list*. Instead (as it is already said) there is a table named *count*, the contents of which are modified. Document matching is done in a similar way as profile insertion. Firstly, we identify which attributes exist in the incoming document d. Then, for each attribute i of d, we probe the data structures in the respective slots of ATT using AwpMatch. When AwpMatch is called, its input document is the text contained in the respective attribute i of d, not the whole document. *count* is a table containing integers, with size is equal to the number of profiles N. There is exactly one slot for each profile. Before AwpMatch is called, *count*'s contents are all set to zero. Whenever a profile p's attribute i is found to match the respective attribute of d, p's entry in *count* is increased by one. At the end of the matching process, the contents of *count* are compared with the 0 contents of *total*. The matching profiles are the ones the entries of which in *count* is equal to the respective entry in *total* (the ones that have all their attributes matched). This is a very fast method to find the matching profiles and helps us avoid set intersection (among the *success_list*s that each run of HashTrieProximityMatch would produce). This method was inspired by the Count algorithm proposed in [37]. The algorithm used for matching is presented in Figure 5.10.

5.3.1 Space Complexity

For the complexity calculations of the attribute enabled method, let us assume that each profile contains A attributes. In this method, the idea of *word_pool* of size $O(L \cdot |V_p|)$ is also employed. According to section 5.2.1, the space complexity of each slot of ATT is $O(S \cdot (|I_P| + N) + 2 \cdot K)$, so the space employed for ATT is $O(N \cdot S \cdot (|I_P| + N) + 2 \cdot N \cdot K)$. Also, the spaced used by *total* is $O(N)$. Thus, the space needed by the attribute enabled method is:

$$O(L \cdot |V_p| + N \cdot S \cdot (|I_P| + N) + 2 \cdot N \cdot K + N) =$$

$$O(L \cdot |V_p| + N \cdot S \cdot |I_P| + 2 \cdot N^2 \cdot K + N^2 \cdot S) =$$

$$O(L \cdot |V_p| + N \cdot S \cdot |I_P| + 2 \cdot N^2 \cdot K)$$

where K and I are as defined in Section 4.1.

5.3.2 Update Complexity

According to Section 5.2.2, the insertion of a boolean atomic profile with proximities needs $O(K + S^2)$ time. So, the update complexity is:

$$O(A \cdot (K + S^2))$$

K is the variable defined in Section 4.1.

5.3.3 Filtering Complexity

According to Section 5.2.3, the time needed to match a single document attribute is $O(T + min(T, |V_d|) + K \cdot S^2 \cdot R^2)$, where T is the average size of the text contained in an attribute and R is the average number of repetitions of a word in a document. Therefore, in AwpMatch, the loop of lines 3 to 6 takes $O(A \cdot (T + min(T, |V_d|) + K \cdot S^2 \cdot R^2))$ time. The loop of lines 7 to 10 consumes $O(N)$ time. So, the filtering complexity is:

$$O(A \cdot (T + min(T, |V_d|) + K \cdot S^2 \cdot R^2) + N) =$$

$$O(D + min(D, A \cdot |V_d|) + A \cdot K \cdot R^2 + N)$$

5.4 Experimental Evaluation

In this section, we evaluate the method previously presented and we compare it with a Brute Force algorithm, as well as with algorithm SingleWordIndex presented in [32].

5.4.1 Unit Sets Creation

All the profiles used to evaluate algorithms for $\mathcal{AWP}$ are generated from the combination of four different unit sets created from the selection of words and multi-word terms that appear in the NN corpus documents. In this section, we describe these unit sets as well as the procedure followed for their creation. There are two unit sets that contain proximity formulas. There is also the set NS of nouns taken from document abstracts and the set of author surnames AS, the creation of which is described in Section 4.2.2.

In order to create the unit sets that contain proximity formulas, the set of multi-word terms MS (the creation of which is described in Section 4.2.2) is used. For the explanation of the creation of each of these unit sets, let us assume a multi-word term $w_1 w_2 \dots w_n$.

The first unit set contains proximity formulas of the form $w_1 \prec_{[0,0]} w_2 \prec_{[0,0]} \ldots \prec_{[0,0]} w_n$. It will be referred as PF_0. The number of words that take part in such a formula is 2 to 5 (since multi-word terms contain 2 to 5 words). It is obvious that the proximity formulas in this set represent search strings of the form "w_1 w_2 $\ldots w_n$", which is actually one of the possible ways to support string search.

Example 5.7 *The 3-word term* Wavelet Image Coefficients *creates the proximity formula* $Wavelet \prec_{[0,0]} Image \prec_{[0,0]} Coefficients$. *This is equal to the search string* "Wavelet Image Coefficients".

The second unit set contains proximity formulas of the form $w_1 \prec_{[0,k]} w_j$, where k is an integer with $1 \leq k \leq 10$. This unit set will be referred as PF_k. It is created using 3,4 and 5-word terms. For each multi-word term used, the first and last of its words (w_1 and w_j respectively) are used to create the proximity formula $w_1 \prec_{[0,k]} w_j$. k is a number randomly chosen uniformly at random among all integers between 1 and 10.

Example 5.8 *The proximity formula* $Conjugate \prec_{[0,4]}]Method$ *can be created using the multi-word term* Conjugate Gradient Method *and setting (randomly selecting)* $k = 4$.

Note that we do not consider the creation of proximity formulas of the form $w_1 \prec_{[l,k]}]$, with $1 \leq l \leq k$ or $w_1 \prec_{[m,n]} w_2 \prec_{[o,p]} \ldots \prec_{[q,r]} w_n$, with $1 \leq m, n, o, p, q, r$. This is not necessary for the type of the experiments we would like to perform, as these experiments are not affected by different values of l.

Finally, as mentioned before, the set NS that contains nouns from document abstracts as well as AS that contains author surnames are used. These are the same sets described in Section 4.2.2.

In this section we described the different unit sets used for $\mathcal{AWP}$ atomic profile generation as well as the process of their creation. More details can be found in [23, 32].

5.4.2 $\mathcal{AWP}$ Profiles Generation

First of all, we generate some realistic profiles appropriate to evaluate algorithms and data structures that support $\mathcal{AWP}$. The method presented here is the same as the one used in [32] for the evaluation of algorithms that implement the same functionality under an information dissemination setting.

As mentioned earlier, a profile under $\mathcal{AWP}$ is a conjunction of atomic profiles. All of the atomic profiles must be satisfied in order for the whole profile to match a document. In general, an atomic profile is a conjunction of one or more words or proximity formulas. An atomic profiles refers to an attribute and is of the form $A \sqsupseteq wp$, where A is an attribute and wp is a conjunction of one or more words and proximity formulas with only words as operands.

Our first challenge is the creation of realistic atomic profiles. To achieve this, we keep in mind that each atomic profile is nothing more than a conjunction of different units, such as the ones contained in the unit sets described in Section 4.2.2. So, we create atomic profiles by combining units from these unit sets. The first step for the creation of an atomic profile is to decide how many units will take part in it. Let the number of these units be S. S is an integer randomly chosen from the interval $[1, S_{max}]$. S_{max} is different for each attribute. It is equal to 2 for atomic profiles referring to the Author or Title attribute, while it is equal to 3 for profiles referring to Abstract and body attributes. This is logical, as the abstract and the body of a document are usually larger than the authors or title part.

In order to decide which unit sets will offer units to take part in the creation of an atomic profile, we use a selection probability for each of the unit sets. These selection probabilities for each of the sets of Section 4.2.2 are shown in Table 5.1. Using these probabilities, we decide whether a unit set will take part in the creation of the atomic profile. We are able to change the frequency that each unit set appears in a specific attribute of profiles by properly adjusting the corresponding probability. We could make the units of a specific set appear in all the atomic profiles of a specific attribute by setting the respective probability equal to 1. Similarly, we could prevent these units from appearing in any of these atomic profiles by setting the probability equal to 0. If a unit set is decided to take part, then one of its units is randomly chosen to be inserted in the atomic profile, using a uniform distribution. To select the units, a loop executes S iterations. In each iteration i, a unit of the atomic profile is created. The possibility to be selected by PF_0, PF_k, NS or AS is shown in Table 5.1.

Example 5.9 *An atomic profile referring to the Body attribute may be:*

$$BODY \sqsupseteq Wavelet \prec_{[0,0]} Image \prec_{[0,0]} Coefficients \wedge Interpolation$$

We can discriminate two units in this atomic profile. The first is $Wavelet \prec_{[0,0]} Image \prec_{[0,0]} Coefficients$ and comes from PF_0. The second is $Interpolation$ and comes from NS. Other examples of atomic profiles are:

$$ABSTRACT \sqsupseteq Origin \wedge Unary \prec_{[0,3]} Specifiers \wedge Linear \prec_{[0,4]} Method$$

$$TITLE \sqsupseteq Conjugate \prec_{[0,3]} Method$$

$$BODY \sqsupseteq Monte \prec_{[0,4]} Simulation \wedge Optimal \prec_{[0,0]} Linear \prec_{[0,0]} Threshold$$

The atomic profiles that refer to the Author attribute worth to be specially considered. As we can see from table 5.1, all types of units except AS can participate in the creation of an atomic profile that refers to the Title, Abstract

```
algorithm HashTrieProximityMatchNaive
input: a profile trie hash table $H$, a proximity table $PT$, a document $d$
output: $success\_list$ (list of matching profile identifiers)

1    $candidate\_profiles \leftarrow$ HashTrieMatch($d$,$H$)
2    construct occurrence table with positions $OT$ of $d$
3    $success\_list \leftarrow \emptyset$
4    for each profile $p$ in $candidate\_profiles$ do
5        if EvaluateProximity($PT[p]$,$OT$) $= True$ then
6            add $p$ into $success\_list$
7        end if
8    end for
9    return $success\_list$

end algorithm
```

Figure 5.6: A naive algorithm to support matching of documents with boolean profiles with proximities

Attribute	Participating unit sets	Selection probability
Title	PF_0	0.4
	PF_k	0.4
	NS	0.2
Abstract	PF_0	0.4
	PF_k	0.4
	NS	0.2
Body	PF_0	0.4
	PF_k	0.4
	NS	0.2
Author	AS	1.0

Table 5.1: Participation of different unit sets in the creation of atomic formulas for each attribute

```
algorithm HashTrieProximityMatch
input: a document $d$, a profile Hash Table $H$, a profile proximity table $PT$
output: $success\_list$ (list of matching profile identifiers)

1    create proximity occurrence table with position information $OT(d)$ of $d$
2    create Distinct Word List $DWL(d)$ of $d$
3    $Q \leftarrow \emptyset$
4    for each word $w$ in $DWL(d)$ do
5        if there exists a trie $T$ in $H$ with root node $x$ that contains $w$ then
6            enqueue($Q$,$x$)
7        end if
8    end for
9    while $Q \neq \emptyset$ do
10       $x \leftarrow$ dequeue($Q$)
11       for each pair $(u, ptr\_y)$ in $children[x]$ do
12           if word $u$ exists in $OT(d)$ then
13               let $y$ be the node of $T$ pointed to by $ptr\_y$
14               enqueue($Q$,$y$)
15           end if
16       end for
17       if $profiles[x] \neq \emptyset$ then
18           if $remainder[x] = \emptyset$
                 or all words of $remainder[x]$ exist in $OT(d)$ then
19               for each profile $p_i$ in $profiles[x]$ do
20                   if EvaluateProximity$(PT[p_i], OT(d)) = True$ then
21                       add $p_i$ to $success\_list$
22                   end if
23               end for
24           end if
25       end if
26   end while
27   return $success\_list$

end algorithm
```

Figure 5.7: The algorithm used to match a document with a trie hash table and a proximity table

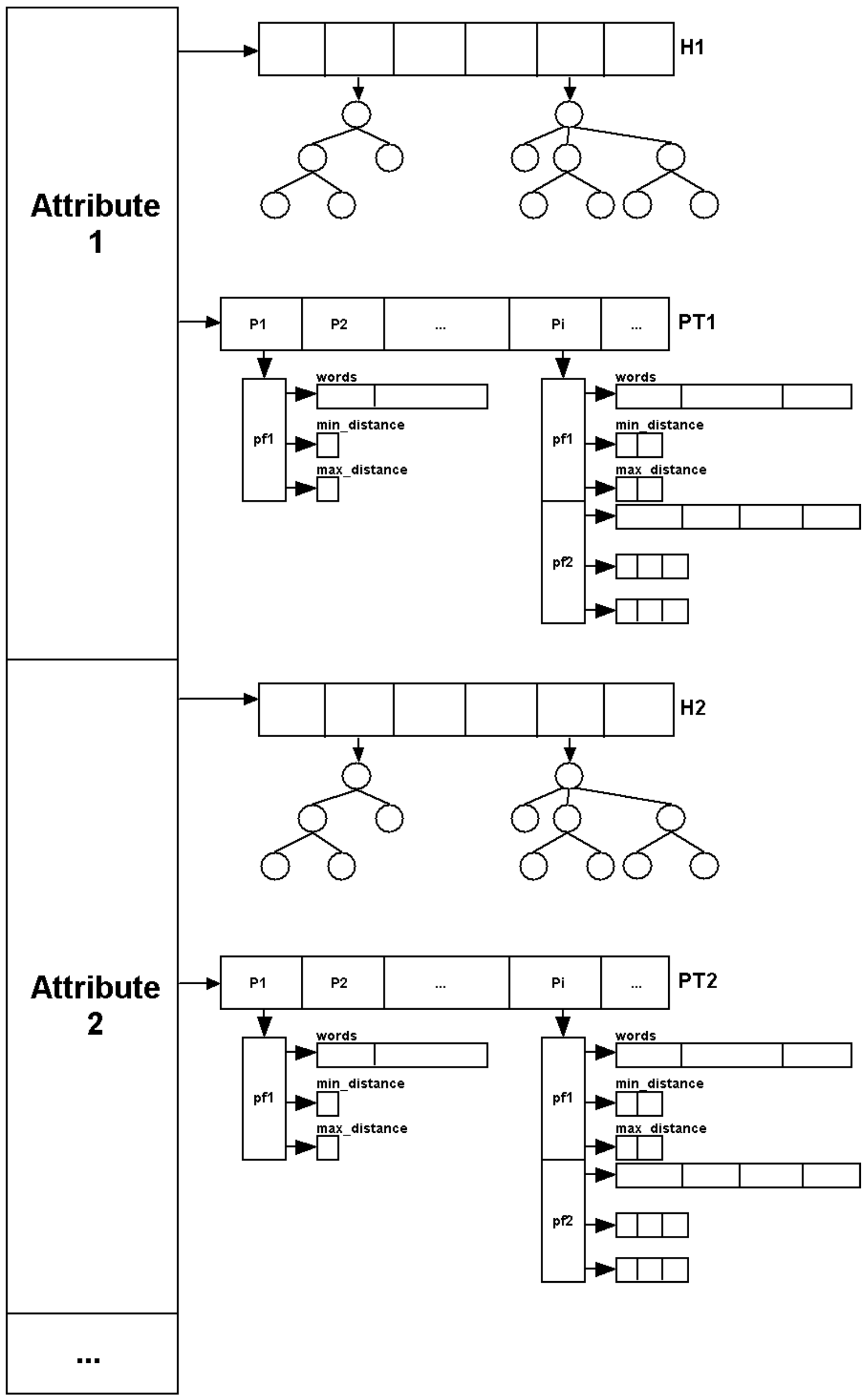


Figure 5.8: The shape of a table able to store $\mathcal{AWP}$ profiles

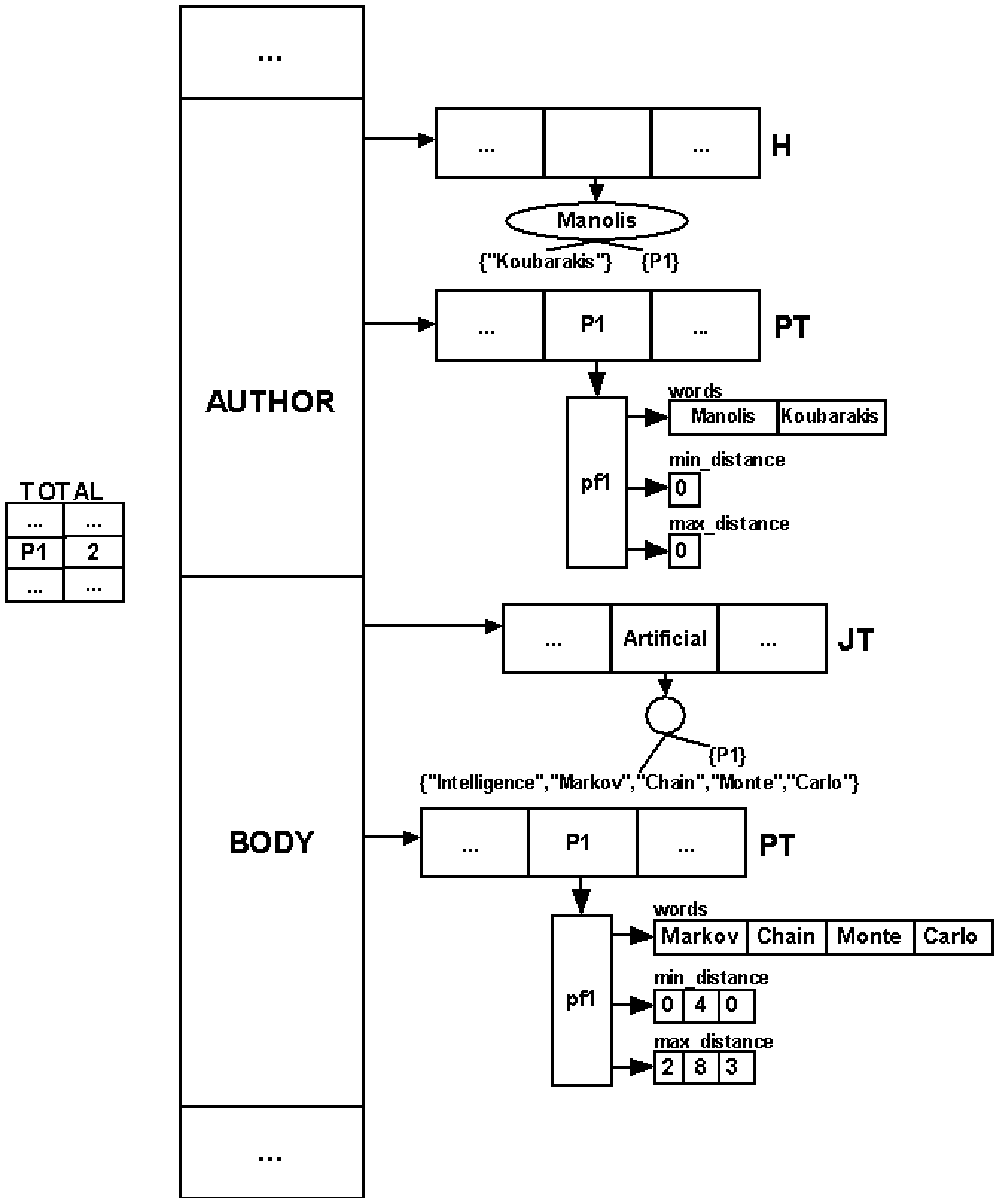


Figure 5.9: An example attribute table with an $\mathcal{AWP}$ profile inserted

```
algorithm AwpMatch
input: a document $d$, an attribute table $ATT$, a total table
output: success_list (list of matching profile identifiers)

1     $success\_list \leftarrow \emptyset$
2     set all entries of $count$ equal to 0
3     for each attribute $i$ of $d$ do
4         let $t$ be the text of attribute $i$
5         HashTrieProximityMatch($t$,$H[ATT[i],PT[ATT[i]]]$)
6     end for
7     for each entry $p$ in $count$ do
8         if $count[p] = total[p]$ then
9             add $p$ into $success\_list$
10        end if
11    end for
12    return $success\_list$

end algorithm
```

Figure 5.10: The algorithm used for matching under $\mathcal{AWP}$

or Body attribute. On the other hand, only AS can take part in the creation of an atomic profile for the Author attribute. This is logical if we consider that in the Author attribute only author surnames are expected to be entered. Recall that for the author attribute $S_{max} = 2$, so an atomic profile for this attribute consists of one or the conjunction of two author surnames. Additionally, for the Author attribute the selection of only one author is favored against the selection of two authors. Specifically, the probability of using one author is 0.8 and that of selecting two authors is 0.2. The same does not hold for the other attributes, where S_{max} is drawn from a uniform distribution. The use of proximity operators is not expected to be exploited by a user of the system, as proximity operators have no meaning in such a small and specialized attribute. Proximity operators would be useful if the corpus provided the first names of the authors. Then, we could create proximity formulas consisting of the name and the surname of an author (for example, $George \prec_{[0,0]} Smith$). But, the first names of the authors are not provided. Also, the use of more than two authors is quite rare when searching for a paper.

Example 5.10 *Two possible atomic profiles for the Author attribute could be* $AUTHOR \sqsupseteq Riedel$ *and* $AUTHOR \sqsupseteq Rice \wedge Barton$.

Having defined a relatively realistic method to generate atomic profiles for $\mathcal{AWP}$, the next step is to select which of the attributes will be entered into a profile. To make this decision, we once more use the idea of selection probabilities. Each attribute has a specific probability of inserting an atomic profile referring to it in a given profile. For each attribute, this probability is equal to 0.2. This means that each generated profile is composed by few atomic profiles. Measuring the resulting profiles, we saw that there were about 1.5 atomic profiles per profile.

5.4.3 Simulation Results

In this section, we present the data that came out of the analysis of experimental data for HashTrie as well as some other algorithms for $\mathcal{AWP}$. These data were the result of simulations were carried out with the settings of Section 4.2.4. The efficiency of HashTrie with proximities was measured and compared to that of the SingleWordIndex method [32] and to that of a Brute Force method. The profiles for these simulations were generated with the method described in Section 5.4.2.

The memory space needed for HashTrie and Brute Force is shown in Figure 5.11. We can see that HashTrie needs more space than Brute Force. The memory requirements for the two algorithms increase at about the same rate for different profile database sizes.

The average match time for each of the algorithms is presented in Figure 5.12. HashTrie and SingleWordIndex are shown to be much faster than Brute Force. Also, the time needed by Brute Force is more sensitive to the profile database size. Between HashTrie and SingleWordIndex, HashTrie is clearly faster.

In Figure 5.13, the processing capability of each algorithm (throughput) is presented. The entry *Algorithm-xM* means "performance of Algorithm with a database size of x millions of profiles". We can clearly see once more that HashTrie is the best of the three algorithms. What is most impressing is that even with a database size of three times the database size of SingleWordIndex, HashTrie is still faster. The same holds for SingleWordIndex and the Brute Force algorithm.

Motivated by the results of Section 4.2.5, we applied and tested the *irank* heuristic with HashTrie and compared it with the original HashTrie method with proximities. Moreover, we evaluated a method that combines Tree with proximities in a similar way with HashTrie and we included the results of the simulations of this algorithm with the rest.

The results for the profile database size are shown in Figure 5.14. We can see that both versions of HashTrie need about the same memory, which is a bit less than the memory needed by Tree with *irank*. The heuristics did not have any serious impact on the profile database size.

The results for matching time are shown in Figure 5.15. The results this time correspond to what we expected. We see a significant speedup when using the *irank* heuristic. Moreover, Tree with *irank* is faster than HashTrie with *irank*. The speedup achieved with Tree is about twice the speedup achieved with HashTrie when using *irank*. A possible reason for this can be found in Section 4.2.5.

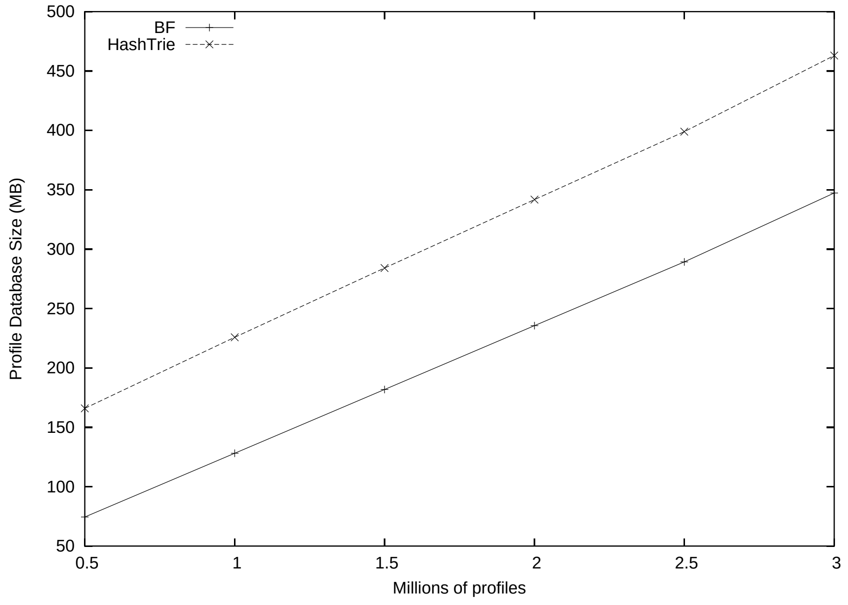


Figure 5.11: Effect of database size in allocated memory space for $\mathcal{AWP}$

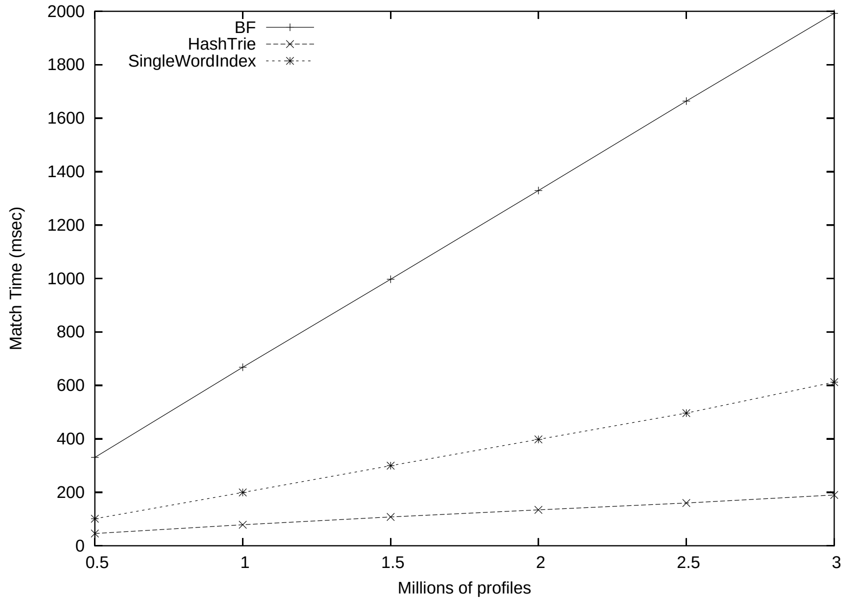


Figure 5.12: Effect of database size in filtering time for $\mathcal{AWP}$

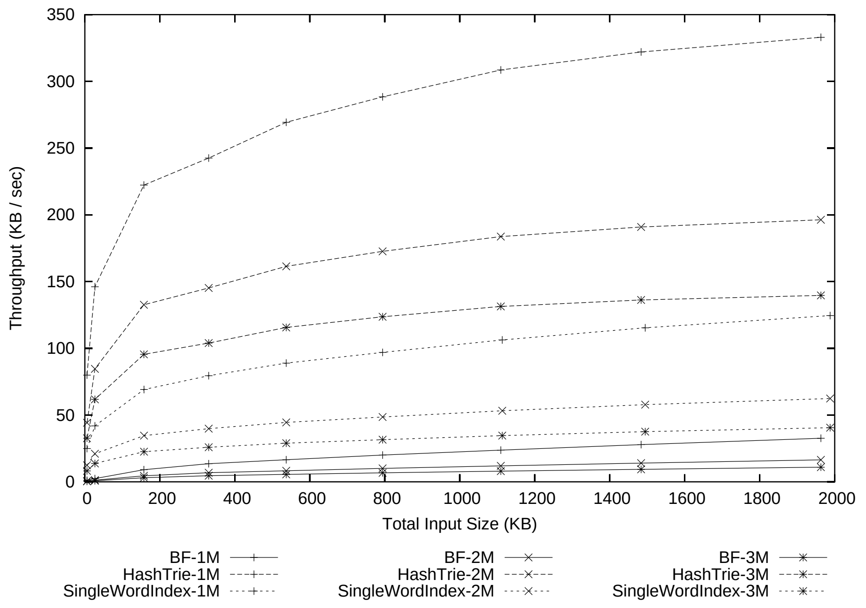


Figure 5.13: Throughput of algorithms for $\mathcal{AWP}$

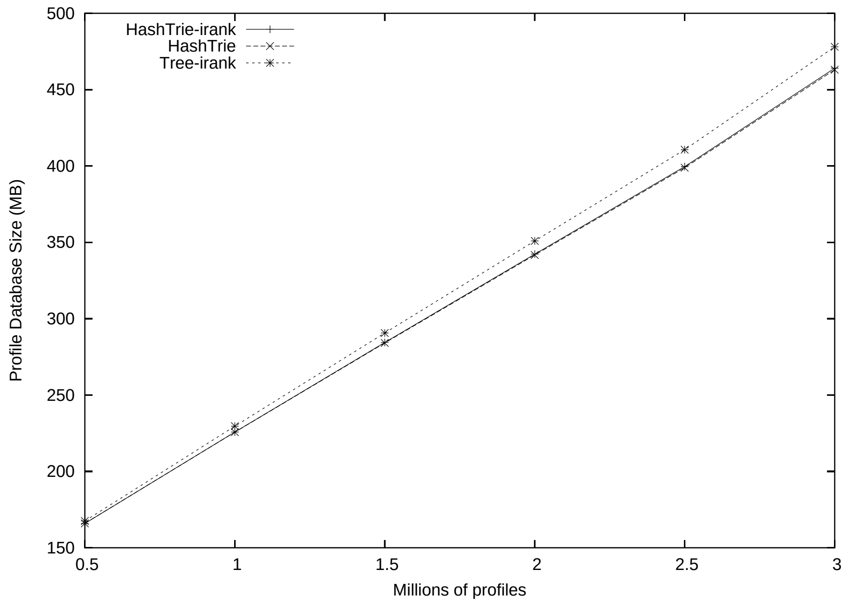


Figure 5.14: Allocated memory with the use of *irank* for $\mathcal{AWP}$

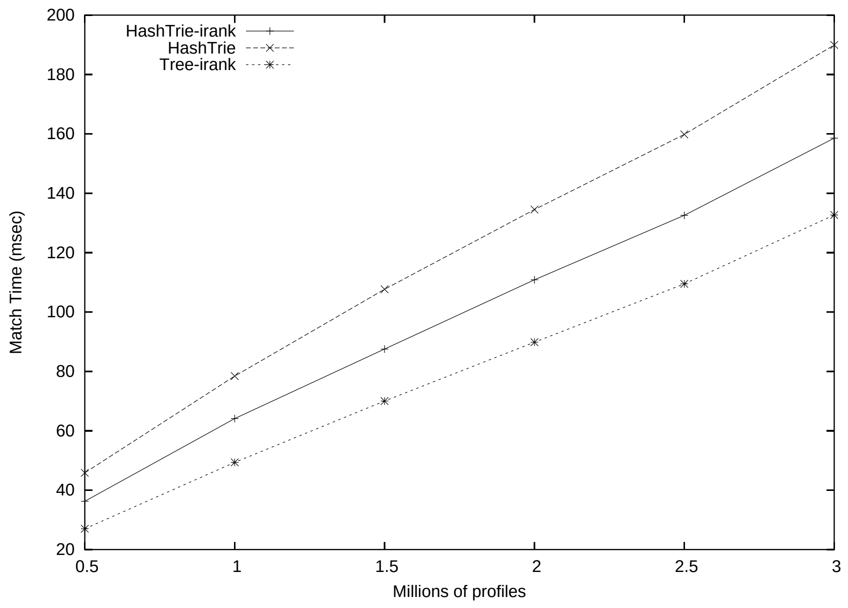


Figure 5.15: Filtering time with the use of *irank* for $\mathcal{AWP}$

Note that this time, Tree with *irank* does not seem to have as big memory space overhead as in the boolean model. This is because the proximity formulas allocate much space, decreasing the importance of such differences in memory space.

5.5 Conclusions

In this section, we extended the HashTrie method with proximities proposed in Chapter 4 in order to support $\mathcal{AWP}$ profiles. The space and time complexity of this method were calculated and experimental evaluation took place. HashTrie with proximities was compared with SingleWordIndex (an algorithm proposed in [32]) as well as a Brute Force algorithm. Our algorithms outperformed both SingleWordIndex and Brute Force. Motivated by the results of the evaluation that took place for HashTrie under the boolean model, we applied ranking heuristics and once more concluded that ranking is an excellent way of improving our algorithms.

Chapter 6

Methods for $\mathcal{AWPS}$ Support

In this chapter, we present some methods able to support vector space atomic profiles. Moreover, we propose a method that with the use the data structures and algorithms of Chapter 5, is able to offer support for $\mathcal{AWPS}$. Finally, we experimentally evaluate the method and compare it with other algorithms.

For convenience, in this chapter we use the following notation to represent strings of vector space atomic profiles:

$$\langle (w_1, g_1), (w_2, g_2), \ldots, (w_n, g_n) \rangle$$

The above expression assumes that the string contained in a vector space atomic profile p contains n distinct words (w_1 to w_n). Also, the symbol g_i in the above expression represents the weight of the word w_i in p. Similar notation is used for documents matched with atomic profiles under the vector space model. For example, a document d represented as:

$$d = \langle (wd_1, gd_1), (wd_2, gd_2), \ldots, (wd_m, gd_m) \rangle$$

is a document with m distinct words (wd_1 to wd_m), where gd_i is the weight of wd_i in d.

Also, the similarity of a document with a profile is given by Equation 3.1, repeated here for convenience:

$$sim(s_q, s_d) = \frac{s_q \cdot s_d}{\|s_q\| \cdot \|s_d\|} = \frac{\sum_{i=1}^{N} w_{q_i} \cdot w_{d_i}}{\sqrt{\sum_{i=1}^{N} w_{q_i}^2 \cdot \sum_{i=1}^{N} w_{d_i}^2}}$$

However, we assume that the weights of words in a vector space atomic profile or in a document are normalized by the magnitude of the profile or document. Therefore, the vector representation p of a vector space atomic profile is equal to $s_q/\|s_q\|$. Similarly, the vector representation d of a document is equal to $s_d/\|s_d\|$. Due to these normalizations, we can calculate the similarity of the document with the profile using the formula:

$$sim(p, d) = p \cdot d = \sum_{i=1}^{N} g_i \cdot gd_i \tag{6.1}$$

where g_i and gd_i are considered to be the (normalized) weights of word i in the profile and in the document respectively.

6.1 The Query Indexing Method

In this section we present a method to handle vector space atomic profiles. Its name is Query Indexing (QI) and it was originally proposed in [38].

QI uses an inverted index (word directory). For each word w appearing in a set A of vector space atomic profiles, we construct an inverted list of *postings*. There is exactly one posting for each of the words of A in that list. Each posting contains the identifier of the corresponding profile and the weight of w in this profile. The list is saved in the word directory along with w. This means that the word directory contains entries that include a word w and the posting list of w. So, a profile that contains k words appears in k postings, one for each word. The word directory is implemented as a hash table indexing words. During the document matching process, candidate profiles for matching are only those that are included in the posting lists of the terms that appear in the document.

Except for the word directory, an array to store the profile thresholds is also used. The size of this *threshold* array is equal to the number of profiles in the system. Each slot i of this table contains a real number representing the threshold of the respective profile.

Example 6.1 *Let us assume the following vector space atomic profiles:*

$$\begin{aligned} p_1 \sim_{0.25} & \langle (Artificial, 0.20), (Intelligence, 0.14), \\ & (Natural, 0.17), (Language, 0.40), (Recognition, 0.62)\rangle \\ p_2 \sim_{0.20} & \langle (Artificial, 0.95), (Intelligence, 0.30)\rangle \\ p_3 \sim_{0.20} & \langle (Natural, 0.14), (Language, 0.25), (Recognition, 0.21), (System, 0.18)\rangle \end{aligned}$$

The thresholds of p_1, p_2 and p_3 are $\theta_1 = 0.25$, $\theta_2 = 0.20$ and $\theta_3 = 0.20$ respectively. The word directory as well as the threshold table employed for the QI method to represent the profiles are shown in Figure 6.1.

To perform matching of a document (or text string) d, QI uses an extra profile table, named *score*. Each slot i of this table is a real number and corresponds to the same profile as slot i of the *threshold* table. QI also uses an occurrence table $OT(d)$ and a distinct word list $DWL(d)$ of the document. Except for the words

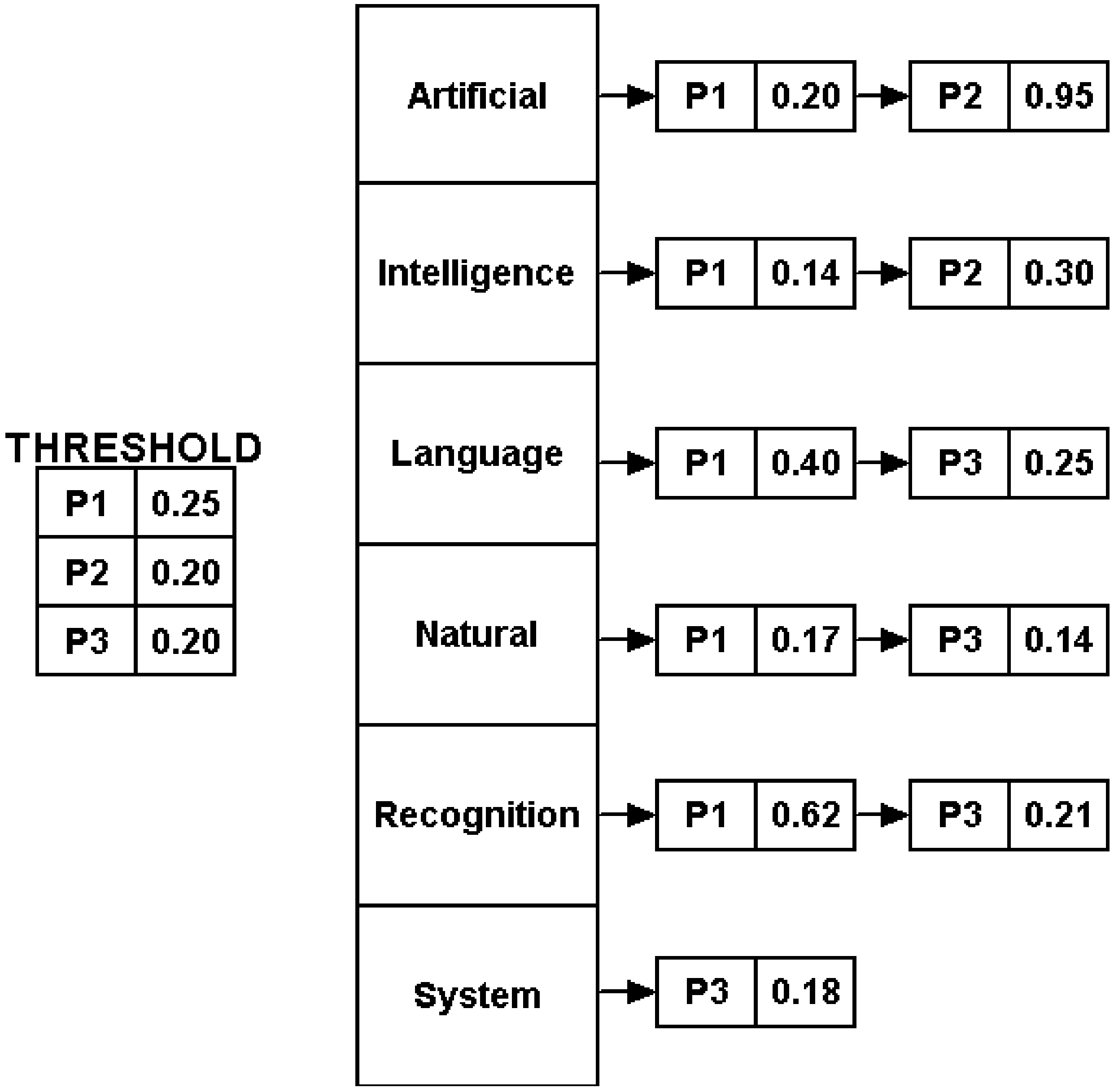
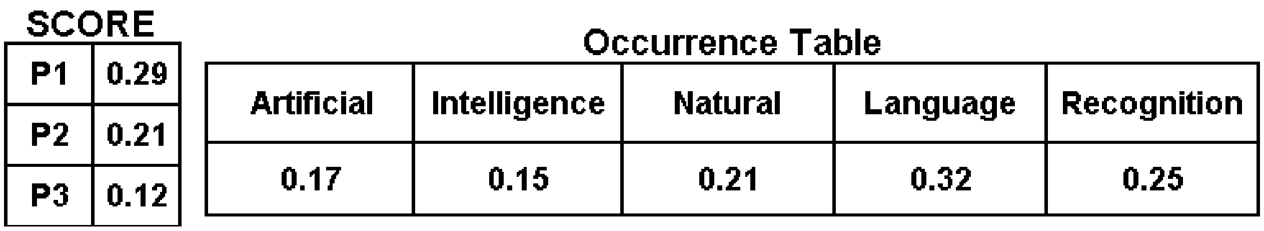


Figure 6.1: A QI word directory and a threshold table



SCORE

P1	0.29
P2	0.21
P3	0.12

Occurrence Table

Artificial	Intelligence	Natural	Language	Recognition
0.17	0.15	0.21	0.32	0.25

Figure 6.2: A sample OT and a score table used for document matching under the QI and SQI method

of d, $OT(d)$ also contains the weight of each word in d. For each distinct word w with weight g in $DWL(d)$, QI uses the word directory to find the postings of the profiles that contain w. For each posting *post* representing profile p, QI adds $weight(post) \cdot g$ to the slot representing p in the *score* table. By the end of this process, we can determine which profiles match d by comparing each entry of the *score* table with the respective entry of the *threshold* table. The algorithm that performs matching is presented in Figure 6.3.

Example 6.2 *Figure 6.2 shows the occurrence table of a document d and the score table after matching with the profiles stored in the data structures of Figure 6.1. Only p_1 and p_2 match the document, as $score[p_3] < threshold[p_3]$.*

6.1.1 Space Complexity

The number of slots of the word directory WD is $O(|V_p|)$. Each profile appears in $O(S)$ postings, so the total number of postings is $O(N \cdot S)$. The size of the threshold table TT is $O(N)$. So, the space complexity of QI is:

$$O(L \cdot |V_p| + N \cdot S + N) = O(L \cdot |V_p| + N \cdot S)$$

6.1.2 Update Complexity

For each word of an inserted profile, there is an insertion in a posting list of WD that takes $O(1)$ time. Thus, each profile needs $O(S)$ time to be inserted.

6.1.3 Filtering Complexity

For the algorithm of Figure 6.3, the above hold: The creation of $OT(d)$ and $DWL(d)$ take $O(D)$ time. The loop of lines 5 to 12 examines $O(N \cdot S)$ postings, so it needs $O(N \cdot S)$ time. The loop of lines 13 to 18 needs $O(N)$ time. So, the filtering of a document using the QI method takes $O(D+N \cdot S+N) = O(D+N \cdot S)$ time.

```
algorithm QiMatch
input: a document $d$, a QI word directory $WD$, a threshold table $TT$
output: $success\_list$ (list of matching profile identifiers)

1   create the occurrence table with weights $OT(d)$ of $d$
2   create the distinct word list $DWL(d)$ of $d$
3   $success\_list \leftarrow \emptyset$
4   set all slots of $score$ table equal to 0
5   for each word $w$ with weight $g$ in $DWL(d)$ do
6       if there exists an entry $e$ indexed by $w$ in $WD$ then
7           for each posting $post$ in $e$ do
8               let $p$ be the profile represented by $post$
9               add $weight(post) \cdot g$ to $score[p]$
10          end for
11      end if
12  end for
13  for each profile $p$ in $TT$ do
14      let $thres$ be the threshold stored in $TT$ for $p$
15      if $score[p] \geq thres$ then
16          add $p$ to $success\_list$
17      end if
18  end for
19  return $success\_list$

end algorithm
```

Figure 6.3: The algorithm used for matching under the QI method

6.2 The Selective Query Indexing Method

The *Selective Query Indexing (SQI)* method used to store and match vector space atomic profiles is presented in this section. This method is an evolution of the QI method presented in Section 6.1. In QI method, a profile is indexed by all its terms. In contrary, SQI indexes only the "significant" words of a profile. Typically, these are the less frequent words in documents. Document matching may become less expensive this way. Like QI, this method was presented in [38].

To motivate SQI, recall that for every document or vector space atomic profile with a vector representation d or p respectively, $0 \leq \|d\| \leq 1$ and $0 \leq \|p\| \leq 1$. Consequently, for the similarity of d with a profile p, according to the Cauchy-Schwarz Inequality [29],it holds:

$$sim(p, d) = p \cdot d \leq |p \cdot d| \leq \|p\| \cdot \|d\| \leq \|p\|$$

This means that the magnitude of a vector space atomic profile is an upper bound of its similarity with any document. This also holds for any subvector of the profile: The similarity of this subvector with any document is less than or equal to the magnitude of the subvector. If this magnitude is less than the atomic profile's threshold θ, then the words contained in the subvector are "unable" to match any document if the document does not contain at least one of the other words of the profile. The words of this subvector are then called *insignificant* words of the atomic profile at a threshold θ. The words of the profile that are not contained in any such subvector are called *significant* words of the profile at a threshold θ.

For example, consider the word "Natural" in profile p_3 of Example 6.1. Suppose that a document that does not contain words "Language", "Recognition" or "System" arrives. The maximum score p_3 could have against this document is 0.14 (if "Natural"'s weight in the document is 1, the highest possible), which is less than the threshold specified for p_3. So, at a threshold of 0.25, the word "Natural" is insignificant, because it alone cannot "produce enough similarity" for any document to be relevant. Thus, we can leave "Natural" unindexed. No document that does not contain any of the other words of p_3 will match it anyway. But, the information about "Natural" and its weight in p_3 must not be discarded, as a document that contains at least one of the other words of p_3 may match it with the help of "Natural".

Similarly, consider the subvector $\langle (Intelligence, 0.14), (Natural, 0.17) \rangle$ in p_1. Suppose a document arrives that contains none of the other words of p_1. In that case, an upper bound to the similarity between this document and p_1 is the the magnitude of this subvector, that is $\sqrt{0.14^2 + 0.17^2} = 0.22$ (in the case "Intelligence" and "Natural" have the maximum possible weight, 1, in the document). With a threshold of 0.25, this subvector is insignificant. So, as we did above, we can leave "Intelligence" and "Natural" unindexed. Again, we must keep the information about these words and their weights in p_1, in case a document with any of the other words in p_1 appears.

A profile like p_1 may have more than one insignificant subvectors. For example, p_1 has among others: $\langle(Intelligence, 0.14), (Natural, 0.17)\rangle$ and $\langle(Artificial, 0.20), (Intelligence, 0.14)\rangle$. So, we have many choices in which subvector to choose. Keeping in mind that what we want to do is decrease the number of postings in order to save lookup work during matching, it is rational to choose the words with the lowest *idf*s in the profile as insignificant terms (they are the words most likely to "offer less similarity" than any of the other words of the profile). Thus, to find the best possible insignificant subvector, we first sort the words by *idf*. Then, we include in the subvector as many low *idf* words as possible, without having the magnitude of the subvector (calculated using the weights of the words) exceed the threshold.

Example 6.3 *Consider the atomic profile p_1 of Example 6.1. In this and all subsequent example, we assume that for any word i in a vector space profile or document, $tf_i = 1$, so i's weight is equal to its idf. If we sort the words of p_1 according to their idfs, then we have:*

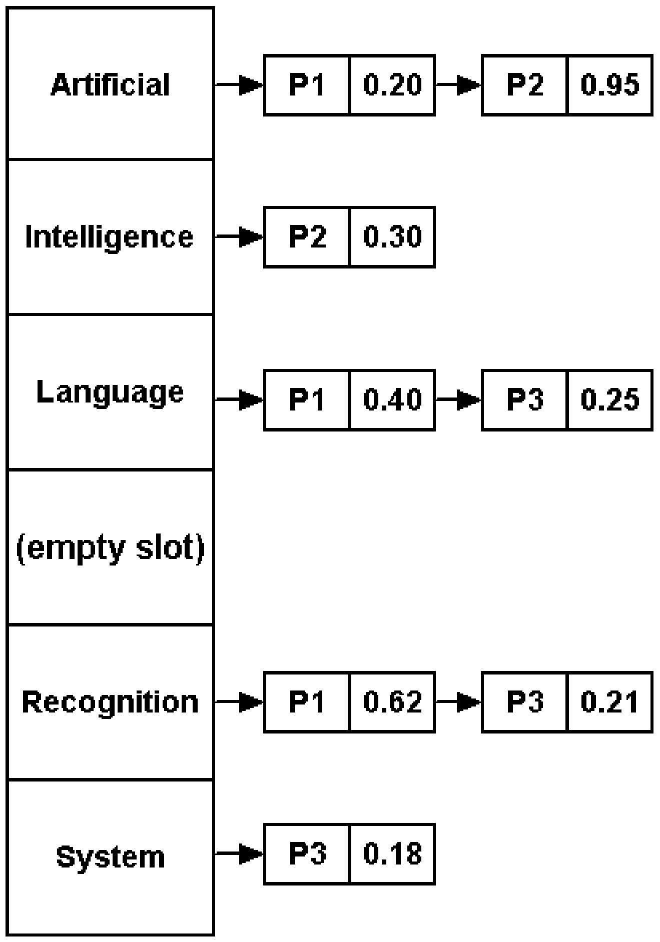
$$p_1 \sim_{0.25} \langle(Intelligence, 0.14), (Natural, 0.17), (Artificial, 0.20), \\ (Language, 0.40), (Recognition, 0.62)\rangle$$

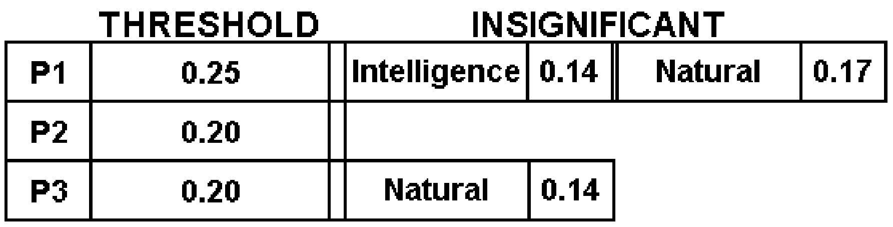
One can easily calculate that the subvector $\langle(Intelligence, 0.14), (Natural, 0.17)\rangle$ is the subvector with the largest possible number of insignificant words (and that contains no significant words).

SQI uses an inverted index (word directory) WD and a threshold table TT just as QI does. But there are some differences: The "insignificant" words of an atomic profile are stored in an array, along with their weights in the profile. There should be a table where the array with the insignificant words of each atomic profile could be stored. This table is unified with TT. So, except for a profile's threshold, each slot of TT also stores the array with the insignificant words of the profile along with their weights in that atomic profile.

Example 6.4 *Under the SQI method, the atomic profiles of Example 6.1 are stored in the data structures of Figure 6.4. Notice that atomic profile p_2 has no insignificant words and therefore, the array in the respective position of TT is empty.*

SQI performs matching in a similar way as QI. There is a *score* table, an occurrence table $OT(d)$ and a distinct word list $DWL(d)$ for the document d. As QI, for each distinct word w with weight g in $DWL(d)$, SQI uses the word directory to find the postings of the profiles that contain w. For each posting *post* representing a profile p, QI adds $weight(post) \cdot g$ to $score[p]$ (the slot representing p





	THRESHOLD	INSIGNIFICANT			
P1	0.25	Intelligence	0.14	Natural	0.17
P2	0.20				
P3	0.20	Natural	0.14		

Figure 6.4: A SQI word directory and a threshold and insignificant words table

in the *score* table). Moreover, if $score[p]$ was equal to 0 before before the addition of the product, for each word wi contained in the insignificant array of $TT[p]$, SQI adds the product of the weight of wi in p with the one in d to $score[p]$. This way, insignificant terms are also taken into account during matching. Note that insignificant terms are considered the first time a word of p is probed in WD and only then. The algorithm used for matching is presented in Figure 6.5.

6.2.1 Space Complexity

For the complexity calculations of the SQI method, let us assume that each profile has k significant and l insignificant words. The number of slots of the word directory WD is $O(|V_p|)$. Each profile appears in $O(k)$ postings, so the total number of postings is $O(N \cdot k)$. Each profile has $O(l)$ insignificant words. The number of slots of the threshold table TT is $O(N)$, so the size of TT would normally be $O(N \cdot L \cdot l)$. But, inspired of the idea of *word_pool* of Section 4.1.1, we also use a *word_pool* in the SQI method. The *idf* of each word is saved in the *word_pool* along with the word itself. So, the size of TT is $O(N \cdot l)$ and the space complexity of the SQI method is:

$$O((L+1) \cdot |V_p| + N \cdot k + N \cdot l) =$$
$$O(L \cdot |V_p| + N \cdot S)$$

6.2.2 Update Complexity

For each of the significant words of an inserted profile, there is an insertion in a posting list of WD that takes $O(1)$ time. For each of the insignificant words of the profile, there is an insertion in an array that also needs $O(1)$ time. Thus, each profile needs $O(k + l) = O(S)$ time to be inserted.

6.2.3 Filtering Complexity

For the algorithm of Figure 6.5, the above hold: The creation of $OT(d)$ and $DWL(d)$ take $O(D)$ time. The loop of lines 10 to 14 needs $O(l)$ time. The total number of iterations of the loop of lines 7 to 17 (given that it is included in the loop of lines 5 to 19) is $O(N \cdot k)$. The loop of lines 20 to 25 needs $O(N)$ time. So, the filtering complexity of a document under the SQI method is:

$$O(D + N \cdot k \cdot l + N) = O(D + N \cdot k \cdot l) < O(D + N \cdot S^2)$$

6.3 Joining HashTrie with Proximity and SQI

In this section we introduce a method consisting of data structures and algorithms appropriate to support both boolean atomic profiles with proximities

```
algorithm SqiMatch
input: a document $d$, a QI word directory $WD$, a threshold table $TT$
output: $success\_list$ (list of matching profile identifiers)

1     create the occurrence table with weights $OT(d)$ of $d$
2     create the distinct word list $DWL(d)$ of $d$
3     $success\_list \leftarrow \emptyset$
4     set all slots of $score$ table equal to 0
5     for each word $w$ with weight $g$ in $DWL(d)$ do
6         if there exists an entry $e$ indexed by $w$ in $WD$ then
7             for each posting $post$ in $e$ do
8                 let $p$ be the profile represented by $post$
9                 if $score[p] = 0$ then
10                    for each word $w_i$ contained in $insignificant[TT[p]]$ do
11                        let $g_i$ be the weight of $w_i$ in $d$
12                        let $weight(w_i)$ be the weight of $w_i$ in $p$
13                        add $weight(w_i) \cdot g_i$ to $score[p]$
14                    end for
15                end if
16                add $weight(post) \cdot g$ to $score[p]$
17            end for
18        end if
19    end for
20    for each profile $p$ in $TT$ do
21        let $thres$ be the threshold stored in $TT$ for $p$
22        if $score[p] \geq thres$ then
23            add $p$ to $success\_list$
24        end if
25    end for
26    return $success\_list$

end algorithm
```

Figure 6.5: The algorithm used for matching under the SQI method.

and vector space atomic profiles. The method is based on the combination of the HashTrie method with proximities presented in Section 5.2 and the SQI method presented in Section 6.2. The type of the supported atomic profiles is either boolean with proximities or vector space.

This *Joint* method indexes words in a hash table JH. This hash table plays the role of hash table H in the HashTrie method, while at the same time serves as a word directory (same as WD) for the SQI method. This means that each slot of JH serves as a placeholder for a trie, a posting list or both a trie and a posting list. The shape of such a hash table is shown in Figure 6.6. As we can see, a slot may contain a trie, a posting list or both.

Apart from JH, the Joint method also uses a proximity table PT that stores the proximity data for each boolean atomic profile with proximities. The proximity table is the same as the one used in Section 5.2. A threshold table TT, same as the one of Section 6.2, is also used to store the thresholds and insignificant words of vector space profiles.

Insertion of boolean atomic profiles with proximities is done in exactly the same way as in Section 5.2. The data structures present to serve vector space atomic profiles are ignored. Only PT and the tries stored in JH are considered. Similarly, during the insertion of a vector space profile, the data structures that serve boolean atomic profiles with proximities are ignored and only TT and the postings of JH are examined and the procedure presented in Section 6.2 is followed. So, insertion of a boolean profile with proximity is done by using HashTrieInsert and by creating the proximity formula arrays, while insertion of a vector space profile is done by using SQIInsert. The algorithm that implements the insertion is presented in Figure 6.7.

Matching of a document (or part of a document) d is performed with the use of an occurrence table $OT(d)$ that is a combination of the ones used in the HashTrie method with proximity and in the SQI method. More specifically, the occurrence table is implemented as a hash table indexing document words. Each slot of $OT(d)$ indexing a word w contains the weight of w in d and a list showing the positions in d where w is met. An example of such a data structure is shown in Figure 6.8. Note that a word w_1 that appears in more positions in d than another word w_2 does not necessarily have a grater weight than w_2, as w_2 may has a much greater idf. During the matching process, a distinct word list $DWL(d)$ of the words of d is also used.

The matching procedure is quite simple. The document could be processed first by using the HashTrieProximityMatch in order to find matching profiles among the set of boolean profiles with proximity in the system. Then with the help of SqiMatch algorithm, we could also find the matching vector space profiles. But this method would require that $OT(d)$ and $DWL(d)$ would be constructed twice and that $DWL(d)$ would be also traversed twice. This is the reason that we create a new algorithm called JointMatch to perform matching of a document. This algorithm traverses $DWL(d)$, performing matching with

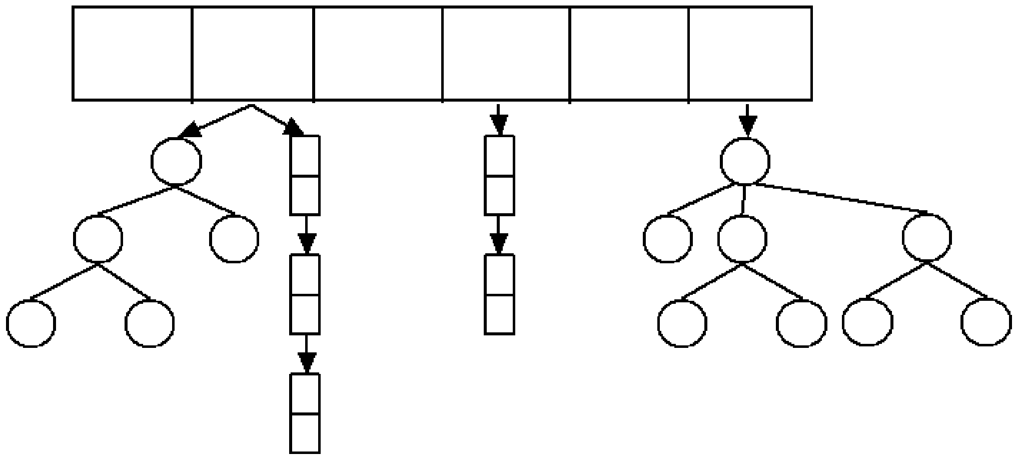


Figure 6.6: The shape of a hash table for boolean or vector space atomic profiles

algorithm JOINTINSERT
input: a profile p, a joint hash table JT, a threshold table TT, a proximity table PT
output: -

```
1    if $p$ is a boolean profile with proximities then
2        HASHTRIEINSERT($p$,$JT$)
3        identify and save all proximity formulas of $p$ into $PT$
4    else
5        insert the significant words of $p$ into the proper posting lists of $JT$
6        insert the insignificant words of $p$ into the proper position of $TT$
7        insert the threshold of $p$ into the proper position of $TT$
8    end if

end algorithm
```

Figure 6.7: The algorithm used for profile insertion under the Joint method

vector space atomic profiles. At the same time, the queue Q (used in the same way as in HashTrie) is filled. After DWL's traversal, the matching vector space profiles are entered into the *success_list*, by performing the same score evaluation as in SQI. Finally, the matching boolean atomic profiles with proximities are added to the *success_list*, just as in the HashTrie method with proximities. This algorithm is given in Figure 6.9 and Figure 6.10.

6.3.1 Space Complexity

For the Joint method complexity calculations, let us suppose that a number B of the atomic profiles are boolean atomic profiles with proximities. Then a number $(N - B)$ of the profiles are vector space profiles. The use of a *word_pool* is also employed in the Joint method as in all other methods. As in SQI, the *idf* of each word is saved in the *word_pool*. The size of *word_pool* is $O((L+1) \cdot |V_p|) = O(L \cdot |V_p|)$. According to Section 4.1.1, the space employed for the tries used for boolean profile support, is $O(S \cdot |I_P| + 2 \cdot K + B)$. According to Section 5.1.1, the space needed for the storage of proximities is $O(B \cdot S)$. According to Section 6.2.1, the size employed for the SQI postings and threshold table is $O((N - B) \cdot S)$. So, the total space needed for the data structures of the Joint methods, is:

$$O(L \cdot |V_p| + S \cdot |I_P| + 2 \cdot K + B + B \cdot S + (N - B) \cdot S) =$$

$$O(L \cdot |V_p| + S \cdot |I_P| + 2 \cdot K + N \cdot S)$$

6.3.2 Update Complexity

As shown in Section 5.2.2, the insertion of a boolean atomic profile with proximities takes $O(K + S^2)$ time, where K is the variable defined in Section 4.1. Similarly, based on Section 6.2.2, we have that the insertion of a vector space atomic profile needs $O(S)$ time.

6.3.3 Filtering Complexity

As we did in the complexity calculations of the SQI method, we assume that each vector space atomic profile has k significant and l insignificant words. Also, let R be the average number of repetitions of a word in a document. For the JointMatch algorithm of Figure 6.9 and Figure 6.10, the above hold: The creation of $OT(d)$ and $DWL(d)$ require $O(D)$ time. The loop of lines 11 to 15 needs $O(l)$ time. The total number of iterations of the loop of lines 8 to 18 (given that it is included in the loop of lines 6 to 23) is $O(N \cdot k)$. The number of executed iterations of loop in lines 6 to 23 is $O(min(D, |V_d|))$. The loop of lines 24 to 29 requires $O(N)$ time. The loop of lines 30 to 47 do a breadth - first search of each trie in H and examine $O(K)$ nodes (where K is defined as in Section 4.1). Line

39 takes $O(S)$ time. According to section 5.1.3, the loop of lines 40 to 44 takes $O(S^2 \cdot R^2)$ time. Thus, the time consumed for matching of a document is:

$$O(D + N \cdot k \cdot l + min(D, |V_d|) + N + K \cdot S^2 \cdot R^2) <$$
$$O(D + min(D, |V_d|) + N \cdot S^2 + K \cdot R^2 \cdot S^2)$$

6.4 An Attribute Based Method for $\mathcal{AWPS}$

In this section, we extend the data structures and algorithms presented in Section 6.3, in order to support attribute based profiles and documents under the $calAWPS$ model.

Support for attribute based profiles means that a profile may consist of one or more atomic profiles. Each atomic profile refers to an attribute and can be a boolean atomic profile with proximities or a vector space atomic profile. The document model supported by this method consists of attributes containing free text. The various parts of the document (body, abstract, title, author, etc) may be defined as attributes.

Example 6.5 *An example of a profile p supported by the method of this section is:*

$$\begin{aligned}
AUTHOR \sqsupseteq \quad & Manolis \prec_{[0,0]} Koubarakis \\
TITLE \sim_{0.25} \quad & \langle (Artificial, 0.20), (Intelligence, 0.14), \\
& (Natural, 0.17), (Language, 0.40), (Recognition, 0.62) \rangle \\
ABSTRACT \sim_{0.20} \quad & \langle (Natural, 0.14), (Language, 0.25), (Recognition, 0.21), (System, 0.18) \rangle \\
BODY \sqsupseteq \quad & Artificial \land Intelligence \land Markov \prec_{[0,2]} Chain \prec_{[4,8]} Monte \prec_{[0,3]} Carlo
\end{aligned}$$

This profile consists of the atomic profiles:

$$p_1 \sqsupseteq Manolis \prec_{[0,0]} Koubarakis$$

that refers to attribute AUTHOR,

$$\begin{aligned}
p_2 \sim_{0.25} & \langle (Artificial, 0.20), (Intelligence, 0.14), \\
& (Natural, 0.17), (Language, 0.40), (Recognition, 0.62) \rangle
\end{aligned}$$

that refers to attribute TITLE,

$$p_3 \sim_{0.20} \langle (Natural, 0.14), (Language, 0.25), (Recognition, 0.21), (System, 0.18) \rangle$$

that refers to attribute ABSTRACT and

$$p_4 \sqsupseteq Artificial \land Intelligence \land Markov \prec_{[0,2]} Chain \prec_{[4,8]} Monte \prec_{[0,3]} Carlo$$

that refers to attribute $BODY$. The semantics of p are that the constraints concerning all of the profile's attributes must be satisfied by a document d in order for the document to match the profile (so, d must contain text in all four attributes). In other words, p_1, p_2, p_3 and p_4 must be satisfied by the the text of the corresponding attributes of d in order for d to match p.

All the other methods we have so far discussed, consider a document containing free text only. In Section 6.3, we presented a method that supports a language with boolean expressions, proximities and vector space queries. In order to support attributes, we employ the algorithms and data structures used in that method.

The data structures of the Joint method are a joint hash table JH, a proximity table PT and a threshold table TT. Let us consider a data structure ATS that contains each of JH, PT and TT. Each ATS is able to store an atomic profile that refers to an attribute of a profile. The main data structure used is a table $ATTS$ of ATS. Its size is equal to the total number of attributes of profiles currently inserted in the system. $ATTS$ can grow to support more attributes. Each slot of $ATTS$ contains all the necessary data structures to support atomic profiles referring to the corresponding attribute. A view of how an $ATTS$ table looks like is shown in Figure 6.11. There is also a table *total*, with size equal to the number N of the profiles currently inserted in the system. For each profile, there is exactly one record in *total*. Each record of *total* contains the number of atomic profiles that consist the respective profile.

Example 6.6 *Assuming that no other profile exists in the system, ATT with only the profile of Example 6.5 stored looks like the table of Figure 6.12 and Figure 6.13. Figure 6.12 also presents the* total *table.*

Insertion is performed in the same way as in the Joint method. For each attribute i of the inserted attribute profile p, algorithm JointInsert (presented in Figure 6.7) is applied to the data structures of the proper slot of $ATTS$, using the atomic profile p_i that represents the respective attribute. So, assuming that p has A attributes, in order to insert p in the system, algorithm JointInsert is executed A times.

In the attribute enabled matching process, we use a slightly different version of JointMatch (presented in Figure 6.9 and Figure 6.10), called AwpsMatch: Its only difference from JointMatch is that this algorithm does not operate on any *success_list*.Instead, there is a table named *count*, the contents of which are modified. Document matching is done in a similar way as profile insertion. Firstly, we identify which attributes exist in the incoming document d. Then, for each attribute i of d, we probe the data structures in the respective slots of $ATTS$ using AwpsMatch. When AttributeJointMatch is called, its input document is the text contained in the respective attribute i of d, not the whole document.

count is a table containing integers. Its size is equal to the number of profiles N. There is exactly one slot for each profile. Before JointMatch is called, *count*'s contents are all set to zero. Whenever a profile p's attribute i is found to match the respective attribute of d, p's entry in *count* is increased by one. At the end of the matching process, the contents of *count* are compared with the respective contents of *total*. The matching profiles are the ones the entry of which in *count* is equal to the respective entry in *total* (the ones that have all their attributes matched). This is a very fast method to find the matching profiles and helps us avoid set intersection (among the *success_list*s that each run of AwpsMatch would produce). This method was inspired by the Count algorithm proposed in [37]. The algorithm used for matching is presented in Figure 6.14.

6.4.1 Space Complexity

For the complexity calculations of the method for $\mathcal{AWPS}$, let us assume that each profile contains A attributes. Let us also assume that in an average profile, there are B boolean atomic profiles with proximity. In this method, the idea of *word_pool* of size $O(L \cdot |V_p|)$ is also employed. According to section 6.3.1, the space complexity of each slot of $ATTS$ is $O(S \cdot |I_P| + 2 \cdot K + N \cdot S)$, so the space employed for $ATTS$ is $O(N \cdot S \cdot |I_P| + 2 \cdot N \cdot K + N^2 \cdot S)$. Also, the space used by *total* is $O(N)$. Thus, the space needed by the attribute enabled method is:

$$O(L \cdot |V_p| + N \cdot S \cdot |I_P| + 2 \cdot N \cdot K + N^2 \cdot S + N) =$$

$$O(L \cdot |V_p| + N \cdot S \cdot |I_P| + 2 \cdot N \cdot K + N^2 \cdot S)$$

where K and $|I_P|$ are as defined in Section 4.1.

6.4.2 Update Complexity

According to Section 6.3.2, the insertion of a boolean atomic profile with proximities needs $O(K + S^2)$ time. According to the same section, the insertion of a vector space atomic profile takes $O(S)$ time. So, the update complexity is:

$$O(B \cdot (K + S^2) + (A - B) \cdot S) =$$

$$O(B \cdot (K + S^2))$$

K is the variable defined in Section 4.1.

6.4.3 Filtering Complexity

According to Section 6.3.3, the time needed to match a single document attribute is $O(T + min(T, |V_d|) + N \cdot S^2 + K \cdot R^2 \cdot S^2)$, where T is the average size of the text contained in an attribute and R is the average number of repetitions of a word in a document. Therefore, in AttributeMatch, the loop of lines 3 to 6 take $O(A \cdot (T + min(T, |V_d|) + N \cdot S^2 + K \cdot R^2 \cdot S^2))$ time. The loop of lines 7 to 10 consumes $O(N)$ time. So, the filtering complexity is:

$$O(A \cdot (T + min(T, |V_d|) + N \cdot S^2 + K \cdot R^2 \cdot S^2) + N) =$$

$$O(D + min(D, A \cdot |V_d|) + A \cdot N \cdot S^2 + A \cdot K \cdot R^2 \cdot S^2)$$

6.5 Experimental Evaluation

In this section, we evaluate the Joint method and we compare it with a Brute Force algorithm.

6.5.1 $\mathcal{AWPS}$ Profiles Generation

A profile under $\mathcal{AWPS}$ is a conjunction of atomic profiles. These atomic profiles may be either boolean profiles with proximities or vector space profiles. We can create boolean atomic profiles with proximities by applying the method of Section 5.4.2. A vector space atomic profile is created using the following procedure:

Recall that a vector space atomic profile consists of one or more words (some text) and a real number in the interval $[0, 1]$, the similarity threshold. So, we have two challenges: To generate text sensible to be used for searching under the vector space model and to generate logical thresholds. By using the term "sensible text", we mean text that a user would enter in order to set a profile.

In order to create the profile text, we use the units available in three unit sets MS, NS and AS, presented in Section 4.2.2. The following explains the process followed to generate a vector space atomic profile for an attribute, given the attribute.

First of all, we have to select the number of words S contained in the atomic profile. S is an integer randomly drawn from the interval $[S_{min}, S_{max}]$. S_{min} and S_{max} are different for each attribute. Their values in each attribute are shown in Table 6.1. S follows the normal distribution for the Title, Abstract and Body attributes. The values of its standard deviation and mean value for each attribute are also shown in Table 6.1. The distribution of the number of words in profiles for the title, abstract and body attribute are shown in Figures 6.15, 6.16 and 6.17 respectively.

The generation of vector space atomic profiles for the author attribute follows exactly the same procedure as the generation of boolean profiles with proximities for this attribute (see Section 5.4.2). Again, we choose one or two authors from the author set AS. The possibility of choosing one author is 80%, while the possibility of choosing two authors is 20%. The only difference is that the resulting profile is a vector space profile and a boolean one.

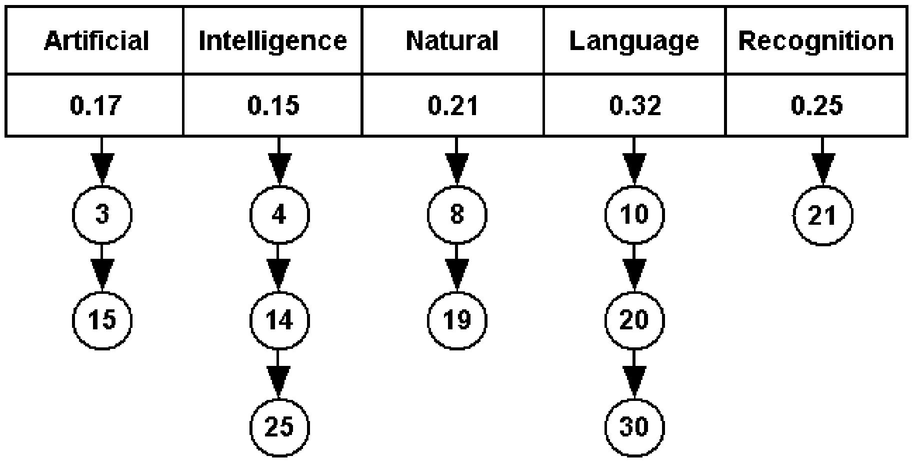


Figure 6.8: An example of an OT used in Joint method

Attribute	S_{min}	S_{max}	Mean	Standard Deviation
Title	1	3	2	1
Abstract	8	16	12	4
Body	20	40	30	10
Author	1	2	-	-

Table 6.1: Distribution parameters for number of words for each attribute

algorithm JointMatch
input: a document d, a joint hash table JT, a threshold table TT, a proximity table PT
output: *success_list* (list of matching profile identifiers)

```
create the occurrence table with weights and position information $OT(d)$ of $d$
create the distinct word list $DWL(d)$ of $d$
$success\_list \leftarrow \emptyset$
$Q \leftarrow \emptyset$
set all slots of $score$ table equal to 0
for each word $w$ with weight $g$ in $DWL(d)$ do
    if there exists an entry $e$ indexed by $w$ in $WD$ then
        for each posting $post$ in $e$ do
            let $p$ be the profile represented by $post$
            if $score[p] = 0$ then
                for each word $w_i$ contained in $insignificant[TT[p]]$ do
                    let $g_i$ be the weight of $w_i$ in $d$
                    let $weight(w_i)$ be the weight of $w_i$ in $p$
                    add $weight(w_i) \cdot g_i$ to $score[p]$
                end for
            end if
            add $weight(post) \cdot g$ to $score[p]$
        end for
    end if
    if there exists a trie $T$ in $H$ with root node $x$ that contains $w$ then
        enqueue($Q$,$x$)
    end if
end for
```

Figure 6.9: The algorithm used for matching under the Joint method

```
24        for each profile $p$ in $TT$ do
25            let $thres$ be the threshold stored in $TT$ for $p$
26            if $score[p] \geq thres$ then
27              add $p$ to $success\_list$
28            end if
29        end for
30        while $Q \neq \emptyset$ do
31            $x \leftarrow$ dequeue($Q$)
32            for each pair $(u, ptr\_y)$ in $children[x]$ do
33                if word $u$ exists in $OT(d)$ then
34                  let $y$ be the node of $T$ pointed to by $ptr\_y$
35                  enqueue($Q$,$y$)
36                end if
37            end for
38            if $profiles[x] \neq \emptyset$ then
39              if $remainder[x] = \emptyset$
                  or all words of $remain[x]$ exist in $OT(d)$ then
40                for each profile $p_i$ in $profiles[x]$ do
41                    if EvaluateProximity($PT[p_i], OT(d)$) $= True$ then
42                      add $p_i$ to $success\_list$
43                    end if
44                end for
45              end if
46            end if
47        end while
48        return $success\_list$

end algorithm
```

Figure 6.10: The algorithm used for matching under the Joint method (continued)

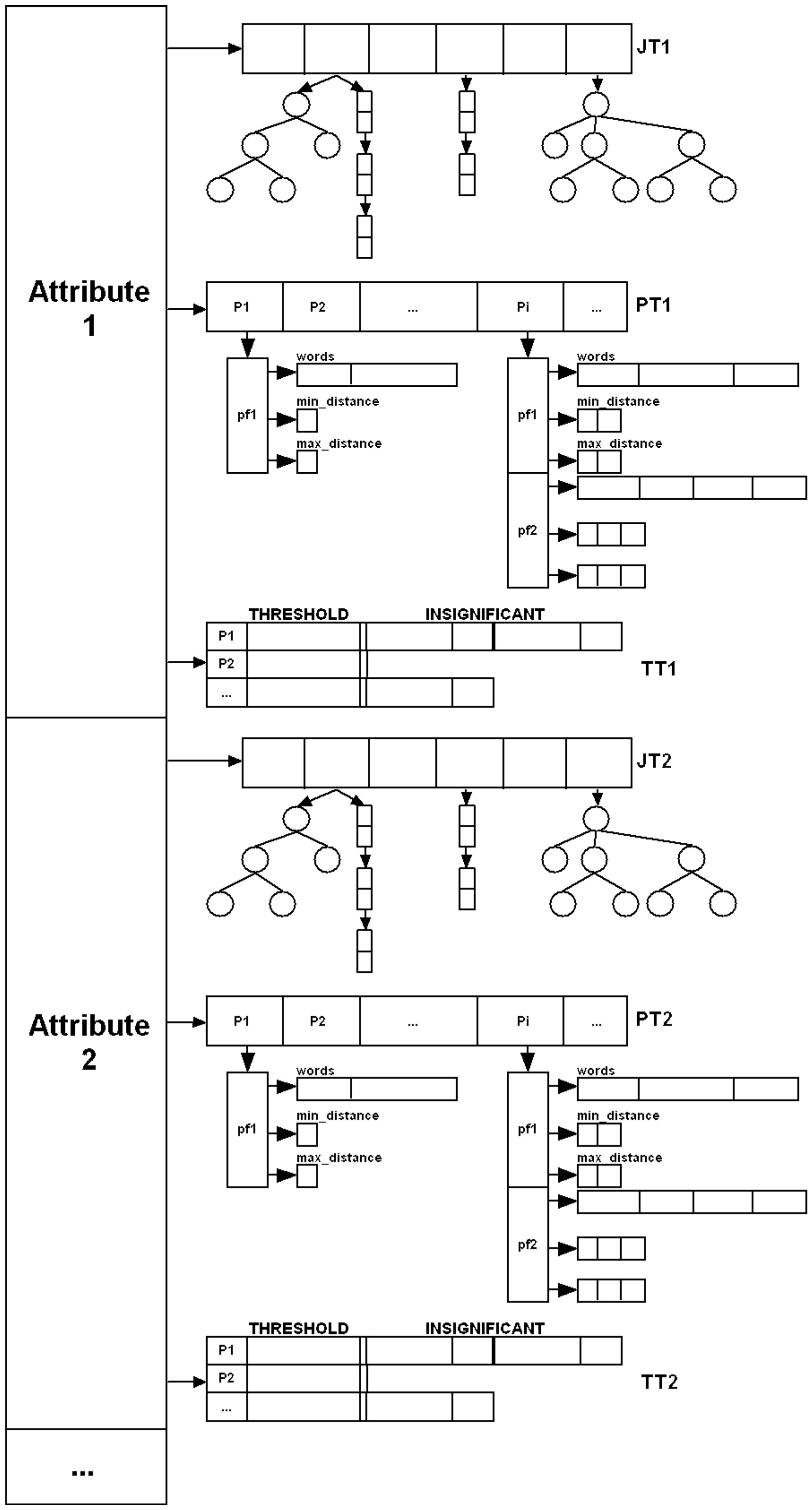


Figure 6.11: The shape of a table able to store $\mathcal{AWPS}$ profiles

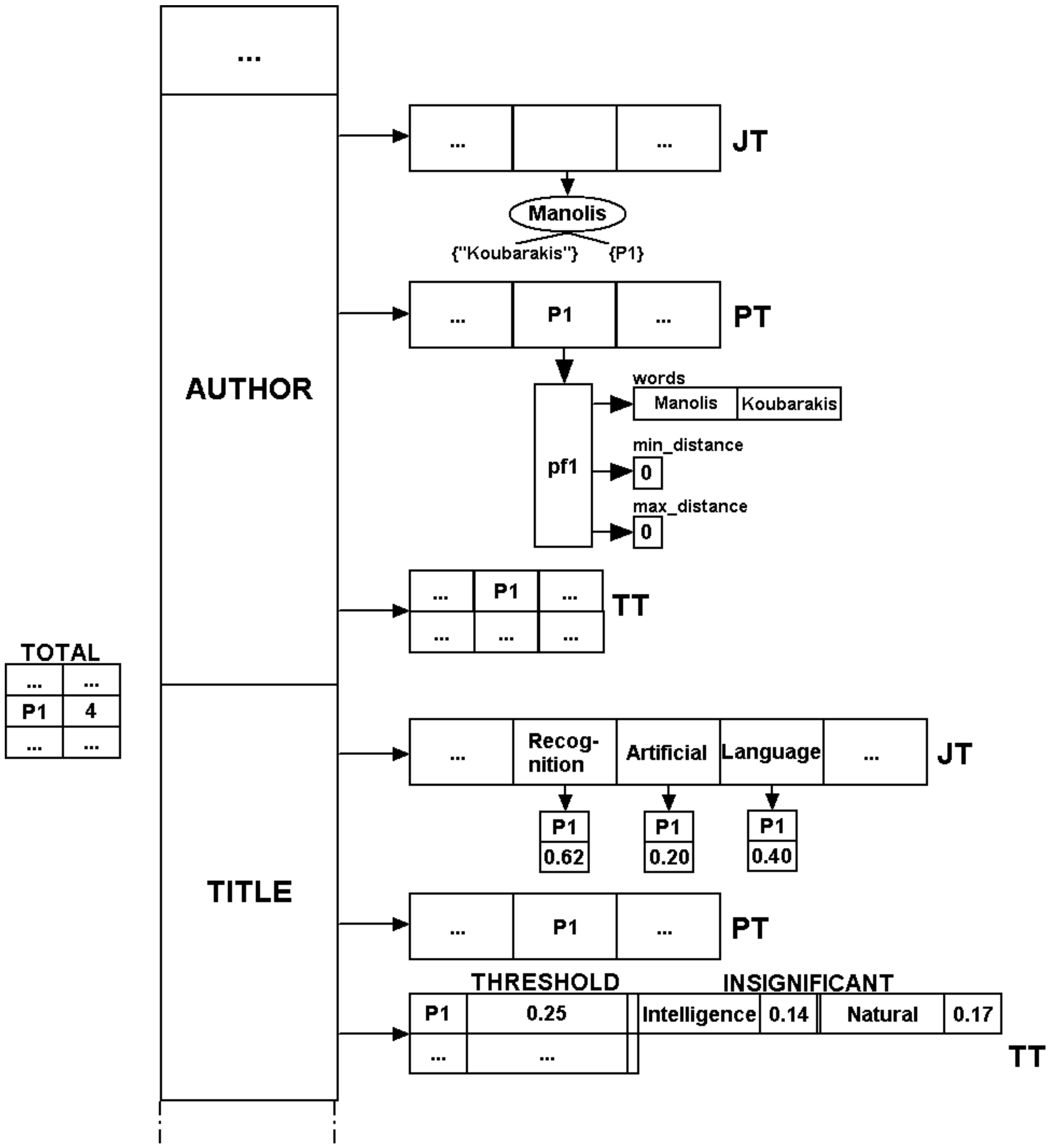


Figure 6.12: An example attribute table with an $\mathcal{AWPS}$ profile inserted

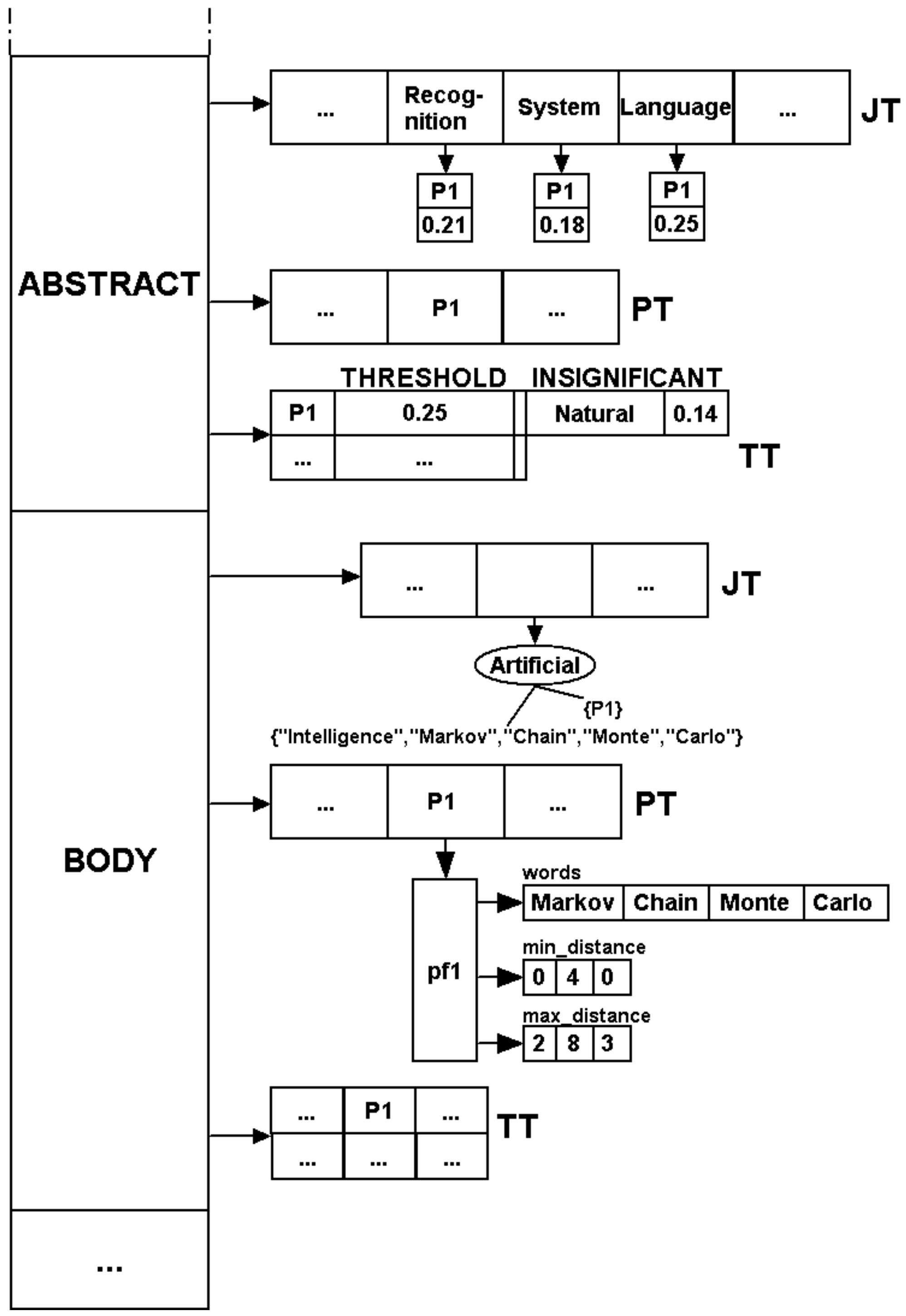


Figure 6.13: An example attribute table with an $\mathcal{AWPS}$ profile inserted (continued)

```
algorithm AwpsMatch
input: a document $d$, an attribute table $ATTS$, a $total$ table
output: $success\_list$ (list of matching profile identifiers)

1     $success\_list \leftarrow \emptyset$
2     set all entries of $count$ equal to 0
3     for each attribute $i$ of $d$ do
4         let $t$ be the text of attribute $i$
5         JointMatch($t$,$JT[ATTS[i]]$,$TT[ATTS[i]]$,$PT[ATTS[i]]$)
6     end for
7     for each entry $p$ in $count$ do
8         if $count[p] = total[p]$ then
9             add $p$ into $success\_list$
10        end if
11    end for
12    return $success\_list$

end algorithm
```

Figure 6.14: The algorithm used for matching under $\mathcal{AWPS}$

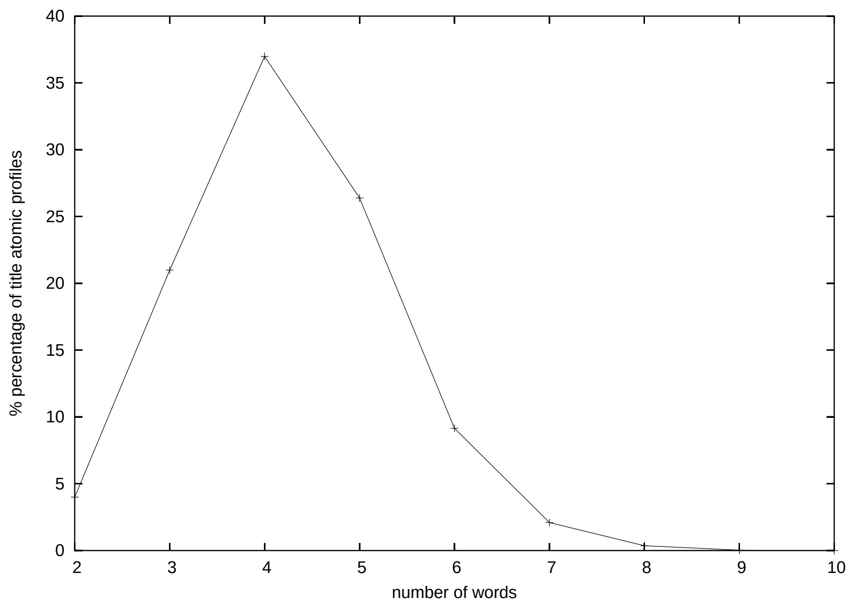


Figure 6.15: The distribution of the number of words in profiles of title attribute

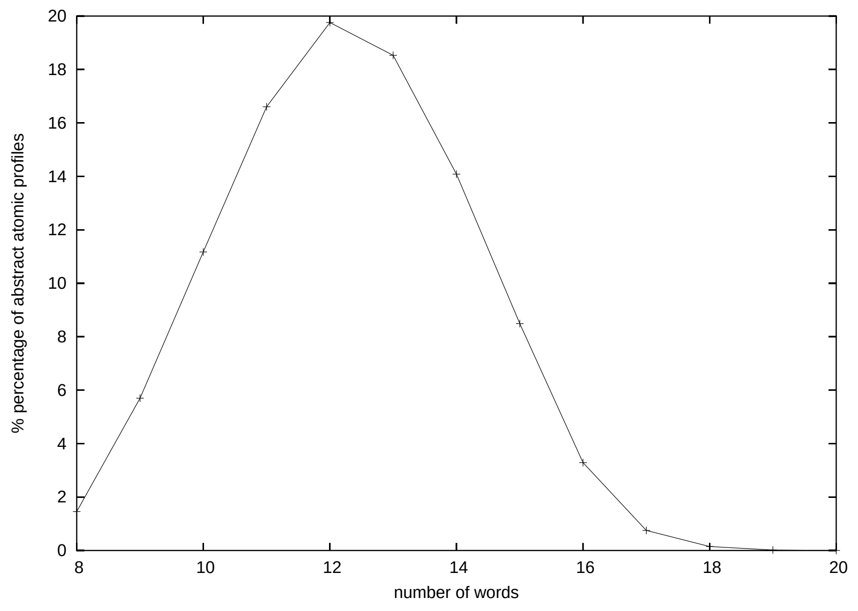


Figure 6.16: The distribution of the number of words in profiles of abstract attribute

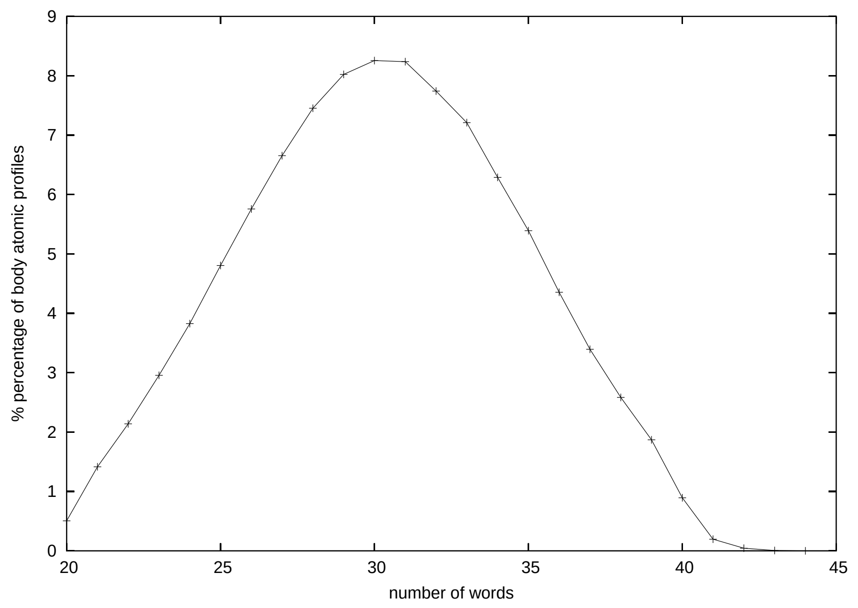


Figure 6.17: The distribution of the number of words in profiles of body attribute

In order to generate atomic profiles for the title attribute, we use units from the abstracts' nouns NS and multi-word terms MS set. The probability of choosing a multi-word term is 80%, while an abstract noun is chosen in the rest 20% of the cases.

The process followed to generate profiles for the abstract attribute is similar to the previous one. A multi-word term is used from MS in 80% of the cases, while a noun from NS is used in the rest 20%. But this happens only in 40% of the generated profiles. In the rest 60%, a sentence or a part of a sentence is chosen from the abstracts of the documents to be the body of the profile. This is because we consider that in many cases the user would like to search using a sentence or a phrase met in a document as a criterion.

The same to the above happens when generating profiles for the body attribute. In 60% of the cases, a text part from a document abstract is chosen, while in the rest, the profile is filled with multi-word terms of MS (selected in 80% of the cases) and abstract nouns from NS (selected in 20% of the cases). We use text from document abstracts and not bodies to generate profiles for the body attribute, as we get more discriminative and logical profiles.

The similarity threshold that a user would enter should be neither too high (or no documents would match, even the ones the user would be interested in), nor too low (because too many unwanted documents would be returned to the user). Using the conclusions of [38], we assume that an average value of 0.2 is logical for a similarity threshold. We also understand that this average value will not be entered by all users. Most of them will enter thresholds near to 0.2 though. Considering these, we generate the thresholds of the profiles using a normal distribution. So, the thresholds of the profiles are randomly generated, following a normal distribution with a mean value equal to 0.2 and a standard deviation equal to 0.2. Of course, the thresholds are restricted to the interval $[0, 1]$. The distribution of the thresholds is shown in Figure 6.18.

What is left is to define the attributes for which a profile will contain a non-empty atomic profile and the type of that atomic profile. The first problem is solved as it did in Section 5.4.2. We require that each attribute will appear in 20% of the documents. After a profile is decided to contain an atomic profile for an attribute, we have to select the atomic profile's type. For half of the generated atomic profiles, we choose them to be boolean profiles with proximities. The rest of the atomic profiles are chosen to be vector space profiles. When the atomic profile is decided to be boolean with proximities, the procedure of Section 5.4.2 is followed for its generation. When it is chosen to be a vector space profile, the process described above is applied to generate the atomic profile.

6.5.2 Simulation Results

In this section, we present the data that came out of the analysis of experimental data for the Joint method for $\mathcal{AWPS}$. These data were the result of simulations ran with the settings of Section 4.2.4. The efficiency of Joint was measured and compared to that of a Brute Force algorithm. The profiles for these simulations were generated with the method described in Section 6.5.1.

The memory space needed for Joint and Brute Force is shown in Figure 6.19. We can see that Joint needs more space than Brute Force. The memory requirements for the two algorithms increase at about the same rate for different profile database sizes.

The average match time for each of the algorithms is presented in Figure 6.20. Filtering time for Joint is much less than filtering time of the Brute Force algorithm. Also, the time needed by Brute Force increases more rapidly with profile database size.

In Figure 6.21, the processing capability of each algorithm (throughput) is presented. The entry *Algorithm-xM* means "performance of Algorithm with a database size of x millions of profiles". We can clearly see once more that Joint is the better than Brute Force. What is most impressing is that even with a database size twice the database size of Brute Force, Joint is still faster.

Motivated by the results of Section 5.4.3, we applied tested the *irank* heuristic with HashTrie and compared it with the original HashTrie method with proximities. Moreover, we evaluated a method that combines Tree with proximities in a similar way with HashTrie. The results were disappointing, as the vector space atomic profiles dominate in matching time requirements, so the speedup of the system is too small to be seen.

6.6 Conclusions

In this chapter we further extended the HashTrie method by adding vector space profiles support. The resulting method is called Joint method. Its space and time complexities are also calculated, while (one more) experimental evaluation shows us that the algorithm used in the method is much faster than the respective Brute Force algorithm. One think that draws our attention here, is that filtering of vector space atomic profiles needs much more time than filtering of boolean atomic profiles with proximities. This does not allow us to have significant improvement of matching speed (the speedup is practically zero compared to the time needed for vector space matching).

The algorithms and data structures of the Joint method presented in this chapter are going to be integrated in *P2P-DIET* [30], a distributed information dissemination system that also offers a variety of other services.

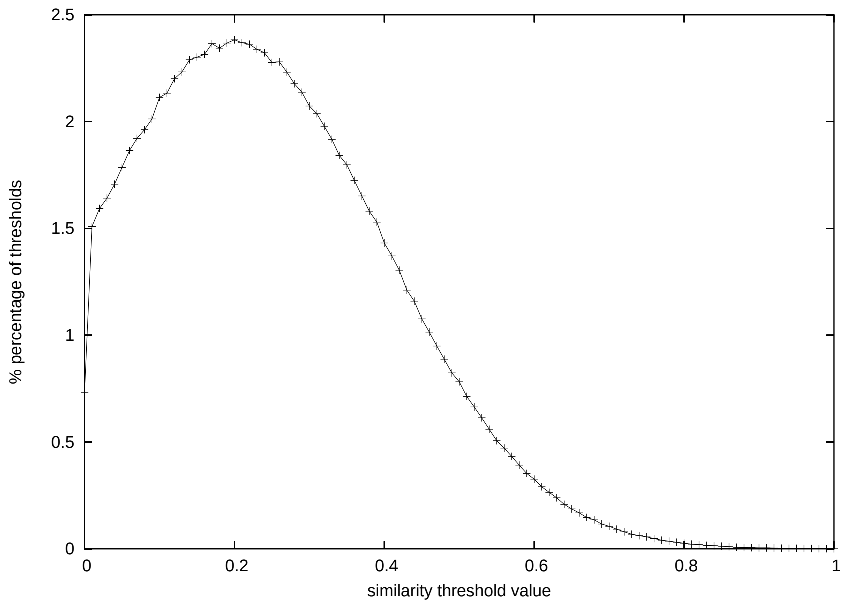


Figure 6.18: Threshold distribution of vector space atomic profiles

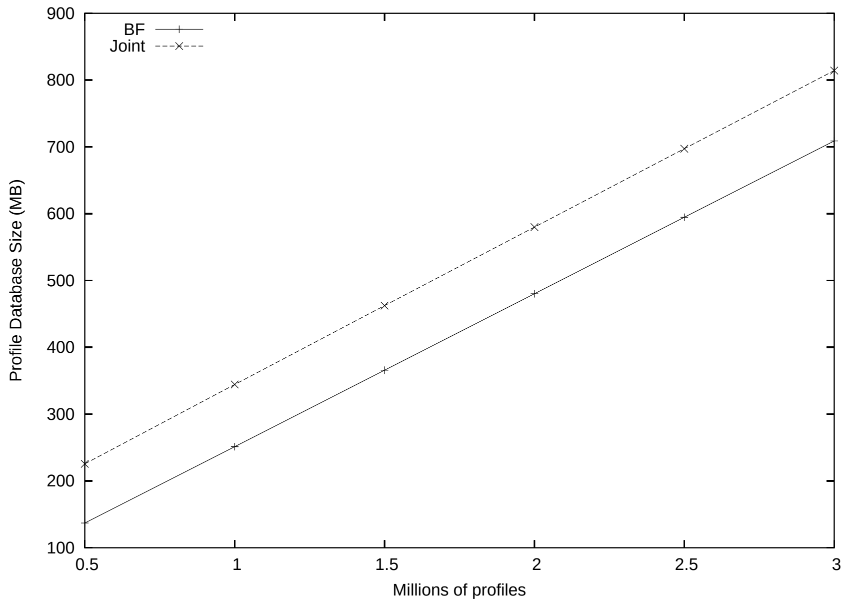


Figure 6.19: Effect of database size in allocated memory space for $\mathcal{AWPS}$

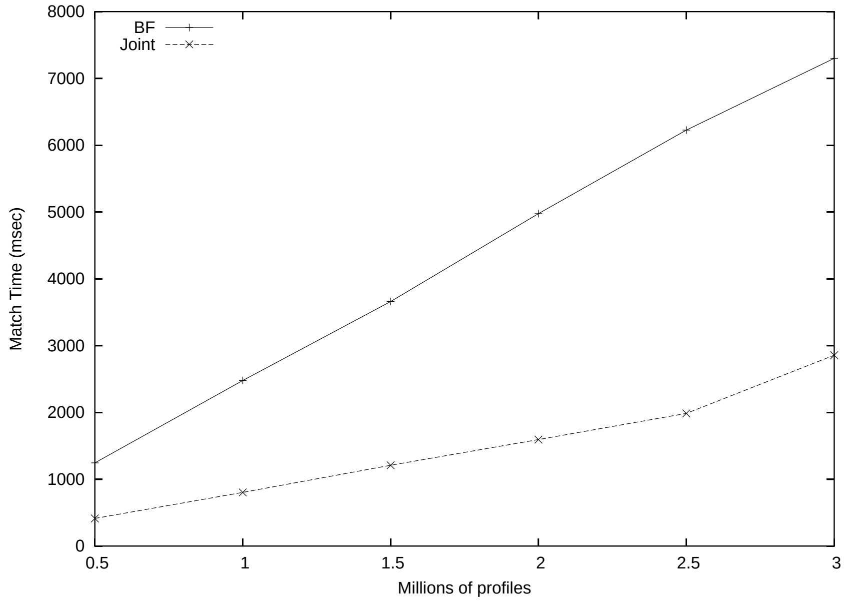


Figure 6.20: Effect of database size in filtering time for $\mathcal{AWPS}$

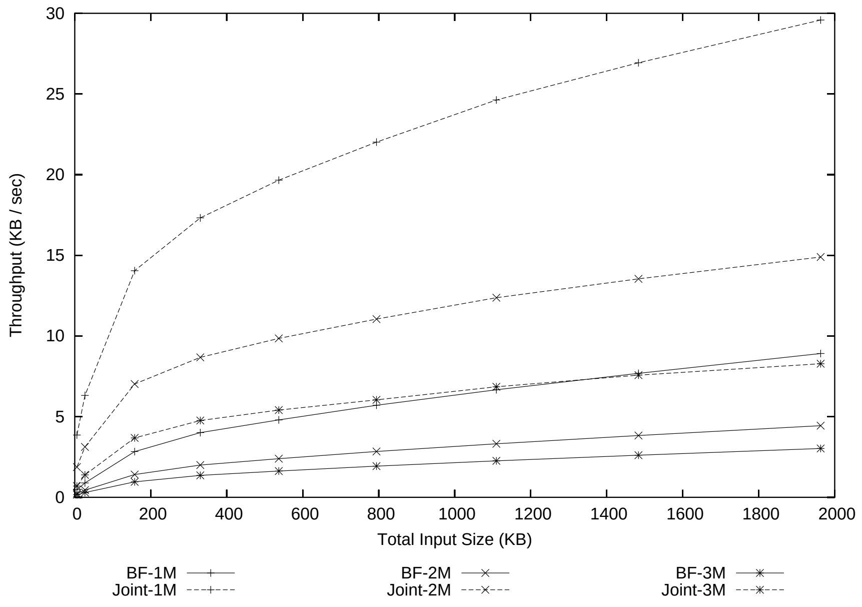


Figure 6.21: Throughput of algorithms for $\mathcal{AWPS}$

Chapter 7

Conclusions and Future Work

This dissertation considered the problem of textual information dissemination in peer-to-peer systems. Initially, we studied work that has been done in the field and presented the advantages and disadvantages of each solution. Then, we dealt with the problem of document filtering in such a system. We considered some well-defined data models and languages for information dissemination. We studied and implemented methods appropriate to support these languages. Finally, we applied a method to generate realistic user profiles and using these profiles, we evaluated the various algorithms.

There is much left to be done. The algorithms and data structures implemented in this dissertation need to be incorporated in system a real information dissemination system, such as DIAS [21]. An interesting extension would be the support of XML based documents supporting XPath [33] or XQuery [34]. Relative work in this field is presented in [2, 8]. Finally, P2P systems architecture is an area with many problems remaining to be solved. Relative work in peer-to-peer systems by our research group is presented in [31].

Bibliography

[1] ResearchIndex: The NEC Research Institute scientific literature digital library. http://www.researchindex.com.

[2] M. Altinel and M.J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th VLDB Conference*, 2000.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 1999.

[4] Eric Brill. A simple rule-based part-of-speech tagger. In *Proceedings of ANLP-92, 3rd Conference on Applied Natural Language Processing*, pages 152–155, Trento, 1992. URL: citeseer.nj.nec.com/brill92simple.html.

[5] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficent filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, 2001.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'2000)*, pages 219–227, 2000.

[7] U. Cetintemel, M.J. Franklin, and C.L. Giles. Self-adaptive user profiles for large-scale data delivery. In *ICDE*, pages 622–633, 2000.

[8] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proceedings of the 28th VLDB Conference, Hong Kong, China*, 2002.

[9] C.-C. K. Chang, H. Garcia-Molina, and A. Paepcke. Boolean Query Mapping across Heterogeneous Information Sources. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):515–521, 1996.

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

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

[12] William W. Cohen. WHIRL: A word-based information representation language. *Artificial Intelligence*, 118(1-2):163–196, 2000.

[13] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[14] Y. Dalal and R. Metcalfe. Reverse Path Forwarding of Broadcast Packets. *Communications of the ACM*, 21(12):1040–1048, 1978.

[15] M. Koubarakis et. al. Project DIET Deliverable 7 (Information Brokering), December 2001.

[16] D. Faensen, L. Faulstich, H. Schweppe, A. Hinze, and A. Steidinger. Hermes – A Notification Service for Digital Libraries. In *Proceedings of the Joint ACM/IEEE Conference on Digital Libraries (JCDL'01), Roanoke, Virginia, USA*, 2001.

[17] P.W. Foltz and S.T. Dumais. Personalised information delivery: An analysis of information filtering methods. *Communications of the ACM*, 35(12):29–38, 1992.

[18] K. Frantzi, S. Ananiadou, and H. Mima. Automatic recognition of multiword terms:the c-value/nc-value method. *International Journal on Digital Libraries*, 5(2), 2000.

[19] M. Koubarakis. Boolean Queries with Proximity Operators for Information Dissemination. Proceedings of the Workshop on Foundations of Models and Languages for Information Integration (FMII-2001), Viterbo, Italy , 16-18 September, 2001. In LNCS.

[20] M. Koubarakis. Textual Information Dissemination in Distributed Event-Based Systems. Proceedings of the International Workshop on Distributed Event-Based systems (DEBS'02), July 2-3, 2002, Vienna, Austria. Available from: http://www.intelligence.tuc.gr/~manolis/publications.html.

[21] 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 Digital Libraries (ECDL2002)*, volume 2458 of *Lecture Notes in Computer Science*, pages 527–542, September 2002.

[22] M. Koubarakis, C. Tryfonopoulos, P. Raftopoulou, and T. Koutris. Data models and languages for agent-based textual information dissemination. In *Proceedings of 6th International Workshop on Cooperative Information Systems (CIA 2002)*, volume 2446 of *Lecture Notes in Computer Science*, pages 179–193, September 2002.

[23] T. Koutris. Textual Information Dissemination in Distributed Agent Systems: Architectures and Efficient Filtering Algorithms. Master's thesis, Department of Electronic and Computer Engineering, Technical University of Crete, Greece.

[24] Steve Lawrence, C. Lee Giles, and Kurt Bollacker. Digital libraries and autonomous citation indexing. *IEEE Computer*, 32(6):67–71, 1999.

[25] Z. Manna and R. Waldinger. *The Logical Basis of Computer Programming*, volume 1. Addison Wesley, 1985.

[26] C.D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.

[27] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[28] M.F. Porter. An Algorithm for Suffix Striping. *Program*, 14(3):130–137, 1980.

[29] Friedbreg S., Insel A., and Spence L. *Linear Algebra*. Prentice-Hall Inc., 1989.

[30] Stratos Idreos. P2P-DIET: A Query and Notification Service Based on Mobile Agents for Rapid Implementation of P2P Applications. Diploma Thesis, Department of Electronic and Computer Engineering, Technical University of Crete, Intelligent Systems Laboratory, June 2003.

[31] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Towards high-performance peer-to-peer content and resource sharing systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, January 2003.

[32] C. Tryfonopoulos. Agent-Based Textual Information Dissemination: Data Models, Query Languages, Algorithms and Computational Complexity. Master's thesis, Department of Electronic and Computer Engineering, Technical University of Crete, Greece, 2002.

[33] W3C. XML Path Language (XPath) 1.0. http://www.w3.org/TR/xpath, 1999.

[34] W3C. XQuery 1.0. http://www.w3.org/TR/xquery, 2002.

[35] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kauffman Publishing, San Francisco, 2nd edition, 1999.

[36] T.W. Yan and H. Garcia-Molina. Distributed selective dissemination of information. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 89–98, 1994.

[37] T.W. 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.

[38] T.W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM Transactions on Database Systems*, 24(4):529–565, 1999.