Index Structures for Selective Dissemination of Information Under the Boolean Model *

Tak W. Yan and Hector Garcia-Molina

Department of Computer Science, Stanford University, Stanford, CA 94305

December 15, 1993

Abstract

The number, size, and user population of bibliographic and full text document databases are rapidly growing. With a high document arrival rate, it becomes essential for users of such databases to have access to the very latest documents; yet the high document arrival rate also makes it difficult for the users to keep themselves updated. It is desirable to allow users to subscribe profiles, i.e., queries that are constantly evaluated, so that they will be automatically informed of new additions that may be of interest. Such service is traditionally called Selective Dissemination of Information (SDI).

The high document arrival rate, the huge number of users, and the timeliness requirement of the service pose a challenge in achieving efficient SDI. In this paper, we propose several index structures for indexing profiles and algorithms that efficiently match documents against large number of profiles. We also present analysis and simulations results to compare their performance under different scenarios.

1 Introduction

With the improving cost effectiveness of secondary storage and the expanding volume of digitized textual data, the number and size of bibliographic and full text document databases are rapidly

*This research was sponsored by the Advanced Research Projects Agency (ARPA) of the Department of Defense under Grant No.MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of ARPA, the U.S. Government or CNRI.

growing. At the same time, the number of users of these databases is also multiplying, as a result of the proliferation of communication networks. Increasingly, such databases employ Information Retrieval (IR) techniques [12] for more effective searching.

With a high document arrival rate, it is essential for users of these databases to have access to the very latest documents. However, the high document arrival rate also makes it difficult for the users to keep themselves updated using only retrospective searches, i.e., searches for documents that are already in the database. It is thus desirable to allow users to subscribe *profiles*, or queries that are constantly evaluated, to capture new documents. This way, users are immediately and automatically informed of new additions that may be of interest. Such service should form an integral part of an IR system, complementary to retrospective search.

Traditionally, libraries and databanks (e.g. Dialog) provide such service under the name of *Selective Dissemination of Information (SDI)* [11]. Users subscribe interest profiles, which are saved and are periodically run to search for additions after the last run. This style of brute force batch processing becomes inefficient as the number of users, profiles, and arriving documents grows. Furthermore, it is becoming important to provide timely service, instead of once a week or once a month.

The Netnews [8] on the Internet is another example of SDI mechanism. However, in many cases it is not very effective because it only provides a fixed number of broad categories. Thus, a user interested in say "relational database systems" needs to receive and read all articles[1] in the newsgroup `comp.databases`, as well as other related newsgroups. Articles about relational database systems appearing in newsgroups not explicitly subscribed to will not be seen. Also, it is the site administrator who decides what newsgroups are locally available. Hence, the database newsgroup may not even be available for this user. It would be more effective if a user could specify profiles in the style of IR systems (e.g., find news items containing the words "relational" and "databases"). The system could then forward the matching documents to the user, regardless of what newsgroup they fall into or what the local site administrator thinks is important. A recent effort [14] to enhance the service provided by Netnews, as well as some other related work, will be discussed in Section 7.

In summary, a SDI service should have the following aspects:

- It should allow a rich class of queries as profiles, unlike Netnews.
- It should be able to evaluate profiles continuously and notify the user as soon as a relevant document arrives, not periodically.

[1] Certain filter mechanisms (e.g., "kill file") exist in news reader programs, but they are rudimentary.

- It should scale to very large number of profiles and large volume of new documents.

- It should efficiently and reliably distribute the documents to subscribers.

In this paper we address the first two aspects, and the third partially. We consider a server that receives new documents and matches them against a database of profiles. In particular, we present several index structures and matching algorithms that the server can use.

To motivate the need for efficient SDI data structures, it is illustrative to look at some statistics from Netnews. According to [10], as of January 1993, the total number of Netnews readers worldwide is estimated to be 1.9 million, and estimates for the average traffic are 49.5 MB and 19400 messages per day (counting cross-posted messages only once). If we consider a Netnews SDI server that served a small fraction (say 5%) of this user population, and each user had say ten profiles (of IR style), the server would have to handle hundreds of thousands of profiles. To match this large number of profiles against a daily influx of tens of thousands of documents in a timely fashion, it is apparent that efficient data structures and algorithms are needed. Furthermore, keep in mind that these Netnews numbers are for a *single* information source *today*. In the future, one would expect many more sources with even higher volumes.

Even though our matching algorithms will be shown to be efficient, the single server concept does not scale to truly large distributed systems. Consider a population of users and a number of information sources in a networked SDI environment. A SDI server collects information from a set of sources and routes it to interested users. There can be multiple servers on the network, each servicing a different set (maybe overlapping) of users and information sources. Profiles can be posted at one or more servers, and documents can be sent to one of more servers. We do not address the architecture of such a distributed system here; however, our index structures and algorithms can be directly applied to each of the distributed servers. Thus, this work can be seen as the first but important step in achieving efficient SDI on a global scale.

To illustrate the problems to be addressed in this paper, let us consider a simple example. Suppose a server has three profiles $P_1 = (a \wedge b)$, $P_2 = (a \wedge c)$ and $P_3 = (f)$. (The notation $P_1 = (a \wedge b)$ means that profile P_1 is subscribing to documents that contain both words a and b.) Say a new document D with words a, b, and e has arrived. One way to process D is to build a hash table for D that lets us quickly tell if it has a given word. We can then run through the profiles and check them. For example, we check that D has a and b and so we send D to the person that posted P_1.

An improvement is to build an inverted index for the profiles [2], which associates every word with a list of profiles that contain it. Hence the word a has a list containing profiles P_1 and P_2, the list of b has profile P_1, that of c has profile P_2, and that of f has P_3. The situation is analogous to conventional IR [12]. In that case, we receive a query and check it against an index of documents. In the SDI case, we get a document and check it against an index of profiles. In IR, to process the query against the index, we perform set operations on the lists of the queried words, and the result of the operations is the answer to the query. For example, the AND operator is processed by intersecting the lists; the OR operator is processed by merging (union) the lists. In processing inverted lists of profiles, we cannot use set operations directly. In our example, the intersection of the lists for the words in D (i.e., a, b, and e) gives us nothing (the list of e is empty). And if we merge the lists, we get both P_1 and P_2, but P_2 does not match D.

However, all is not lost. The result of the union of the lists contains profiles that potentially match the document: each shares at least one word with the document. The merged list is a superset of the desired answer, and so we can screen out the superfluous profiles by checking them against the document, using the hash table as described above. With this strategy we avoid checking all profiles, i.e., profiles that do not contain a, b, or e (P_3 in our example) will not be considered. In this paper we explore in detail changes that can be made to this basic index structure and algorithm to make efficient the screening of potentially matching profiles.

This study is part of the DARPA Electronic Library Project at Stanford. We have implemented two preliminary SDI servers at Stanford to disseminate Netnews articles and computer science technical reports. The reader is encouraged to try out these services. For instructions on how to use these services, send an electronic mail message to either `elib@db.stanford.edu` (for technical reports) or `netnews@db.stanford.edu` (for Netnews) with the word "help" in the message body. Instructions will be returned automatically. The current version of these servers is not efficient (it uses the Brute Force Method with batching, described later on). However, as more users subscribe to our servers, there is an obvious need for an efficient implementation, and this motivated the work reported in this paper.

The rest of the paper is organized as follows. Section 2 explains the terminology and assumptions we use. In Section 3 we describe several index structures and matching algorithms for profiles. Details of our analysis and simulations used to evaluate the various strategies are presented in Section 4. The performance evaluation results are given in Section 5. We then discuss some extensions to our index structures and algorithms in Section 6. Section 7 surveys related work. Finally, a conclusion appears in Section 8.

2 Terminology and Assumptions

2.1 Documents

A *document* consists of a collection of *words*. The set of words that can appear in documents form the *vocabulary*. Words in the vocabulary follow a frequency distribution that describes how often a word appears in a document. The words have *ranks*: the word that appears most (least) frequently has the highest (lowest) rank. In practice, such rank information can be collected from a document database.

Traditional SDI services process documents in batches. If the rate of document arrival is high, it may be reasonable to batch the documents to reduce processing overhead without sacrificing too much response time. On the other hand, if response time is critical (e.g. documents are news stories or stock market reports) it may be necessary to process documents individually. Hence we study both cases. For the bulk of the paper, we assume documents are processed one at a time as they arrive; batching and its performance will be discussed in Section 6.

2.2 Profiles

Research in IR has given rise to many retrieval models, e.g. the boolean model, the vector space model, and the probabilistic model, which are applicable to SDI [2]. In this paper, we focus on the boolean model, which is the one used by most commercial systems (e.g. Dialog, LEXUS) and major library systems (e.g. Melvyl at UC Berkeley, Folio at Stanford), and also supported by new information systems such as freeWAIS [3]. It is important to study other models also, and indeed, [17] reports our work on the vector space model (VSM). In Section 6.2 we give a summary of that work and also compare the performance of the strategies developed for the VSM with the strategies proposed in this paper.

In this paper we focus on conjunctive profiles. We assume that a *profile* P is a sequence of distinct words $(w_1, w_2, ..., w_K)$. A profile *matches* a document if all its words appear in the document. A user can subscribe a number of profiles and a profile may be identical to some other profiles in the system; we assume each profile has a unique integer identifier. The algorithms we present below can easily be extended to handle negation in profiles (e.g., select documents in which a word w does not appear). For simplicity we postpone a discussion of negation to Section 6. Profiles with logical ORs can be handled by converting them to disjunctive normal form, in which case they can be handled as collections of conjunctive profiles. (Example: Posting profile $a \wedge (b \vee c)$ is equivalent to posting

$(a \wedge b) \vee (a \wedge c)$, which is equivalent to posting two profiles, one for $a \wedge b$, the second for $a \wedge c$.) Handling general profiles as collections of conjunctive profiles may not be the most efficient strategy, but again, we postpone this discussion to Section 6.

IR systems have a number of extensions to boolean queries, such as truncation and thesaurus expansion [12]. The methods we present below can be augmented to handle these extensions. The details are given in Section 6.

2.3 Index Structures

In the index structures presented below, for each word w, its *inverted set* of profiles, i.e., profiles containing it, are organized into a list or a tree.[2] The mapping that maps a word to the location of its inverted set on disk is implemented as a hash table, called the *directory*. We assume that the inverted sets are stored on disk while the directory fits in main memory. The index structures do not store information about the subscribers of the profiles, such as their addresses (i.e., where a document is sent to once a match is made). We assume such information is stored elsewhere on disk, referenced by profile identifier. These assumptions are made to obtain a more concrete scenario for presentation and evaluation. Other scenarios could be feasible, but are simply not considered here.

We assume that words are encoded and stored in the index structures as integers. Another option is to represent them as strings of characters. By encoding them as integers, the space required for the index structures is less, but we have to pay the cost of translation when the profiles are entered into the index and when the documents come in. The tradeoff between these two options is not addressed in this paper. We assume that each word in the vocabulary is uniquely identified by an integer word identifier.

The focus of this paper is to devise index structures and algorithms that efficiently process new documents for selective dissemination. The issue of how to efficiently update the index as profiles are updated is not addressed. We assume that the updates are batched; when a document is processed, we check these updates also. Periodically the updates are installed into the index. (This is similar to the solution used by commercial IR systems to handle updates to a standard inverted index of documents.) As profiles represent relatively long-term interests of users [2], this seems a reasonable solution for now; we plan to study this issue further in the future.

By focusing on inverted sets, we are not saying that other search structures are not good for

[2] As detailed below in Section 3, the set may or may not contain all the profiles containing w.

SDI. For instance, signature-based retrieval methods [7] can also be used to speed up SDI processing (i.e., building a signature file of profiles). However, for concreteness, in this paper we only study structures of the "inverted set" category. We do believe that inverted structures are more appropriate for applications with uncontrolled vocabulary (e.g., full text searching where there are no restrictions on the text) like the ones we are interested in, but we are not addressing this claim here. Further work would need to be done to compare the performance of signature-based and inversion-based methods for SDI.

3 Index Structures and Algorithms

In this section we examine three indexing methods, starting with an enhanced version (called the Counting Method) of the basic solution outlined in Section 1. For comparison, we also include the brute force strategy mentioned there.

To facilitate the matching process, the algorithms below make use of one or both of these constructs: the *distinct words set*, which is the set of distinct words in a document, and the *occurrence table*, a hash table that maps a word w to T if w is in the document, F otherwise.

3.1 The Brute Force Methods

The *Brute Force Methods* simply store profiles sequentially on disk without any index structures. This way, all profiles must be evaluated whenever a document arrives.

When a document arrives, we construct the occurrence table. Then we examine each profile in turn. For a profile (conjunction-only) to match the document, every word it contains must be in the document. We look up the words one by one in the occurrence table and stop as soon as we find a word not in the document. If all the words appear in the document, the profile matches.

If we have information about the occurrence frequency of words in documents, then it is possible to improve performance. The idea is to first look up in the occurrence table the least frequent word in the profile, then the next less frequent and so on. This makes it possible to detect non-matching profiles faster, i.e., with fewer probes into the occurrence table. For a matching profile we do not save anything since all words in the profile need to be probed. We call this the *Ranked Brute Force Method.* If we do not have information about the frequency distribution, then effectively we have to look up the words in random order. We call this the *Random Brute Force Method.* We stress that word ranking information may or may not be available. Thus, it is important to study both types of strategies.

A profile is stored on disk as a variable-length *record* with several fields: the profile identifier, the length of the profile (i.e., the number of words in the profile), and the words.

3.2 The Counting Method

The problem with the Brute Force Methods is that they must examine all profiles for each arriving document. To improve, we must reduce the number of potential profiles that need to be looked at. The idea is to build an inverted index for the profiles. For each word, we collect all the profiles containing it to form its inverted set. The inverted set for word w is stored as an *inverted list* of *postings*; each posting in this method contains (only) the identifier of a profile involving w. Thus, a profile with K words will be found in K postings in K different lists. When processing a document D, we only need to examine those profiles in the inverted lists of the words that are in D. This way, the number of profiles looked at is substantially reduced, since we only look at profiles that are known to contain at least one word in the document.

Now with this structure, how do we match a document against the profiles? The solution outlined in Section 1 is to simply merge the lists and then check each profile in the resultant list. This checking is not necessary if we make use of this observation: an occurrence of a profile in an inverted list examined means that one word in that profile is matched. Thus, if we count the number of occurrences of a profile in the inverted lists looked at, we can determine if the profile matches the document. This is called the *Counting Method.*

To do the counting efficiently, we need two (main memory) arrays, TOTAL and COUNT. (This method uses more main memory than the others.) The number of entries in each array is equal to the number of profiles the system handles. Each profile has an entry in each array: the TOTAL entry stores the number of words in the profile, and the COUNT entry is initialized to 0 for each incoming document and is used to keep count of occurrences of the profile in the inverted lists. Assuming the number of words in a profile is less than 256, each entry takes one byte.

When a document D arrives, we first initialize the COUNT array to all 0's. Then we construct the distinct words set. For each distinct word, we use the directory to get its list. For each profile in the list, we increment its entry in the COUNT array by 1. A profile matches a document when its COUNT entry becomes equal to the TOTAL entry.

To illustrate, Figure 1 shows a set of sample profiles, a sample document to be matched, and the various main memory and disk data structures. For example, notice that the inverted list for word

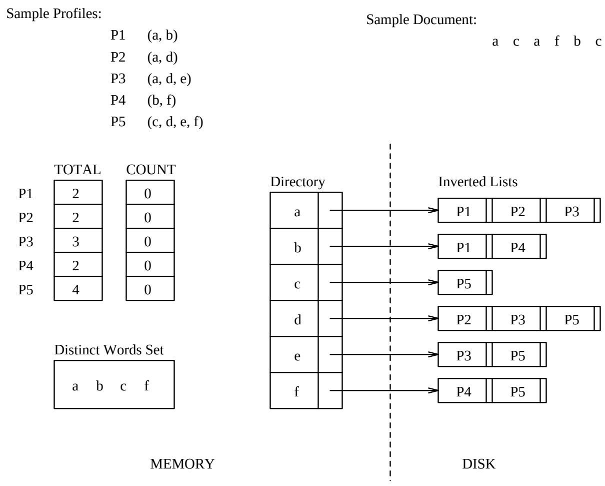

Figure 1: Data Structures for the Counting Method

a contains the identifiers for profiles P_1, P_2, P_3, indicating that these profiles refer to a. To process the sample document, we take the first word a in the distinct words set, retrieve its list, and add 1 to the COUNT entries for profiles P_1, P_2, P_3. We continue with b, c, f, obtaining COUNT entries for P_1, P_2, P_3, P_4, and P_5 of 2, 1, 1, 2, and 2 respectively. For P_1, its COUNT (2) is equal to its TOTAL (2), so P_1 matches. Similarly, P_4 matches.

As we have stated, we only need to keep the profile identifier in an inverted list posting. Postings in the same list are stored sequentially, so there is no need to have a pointer to the next posting.

3.3 The Key Methods

In the Counting Method, a profile $(w_1, ..., w_K)$ appears in K inverted lists. In the *Key Methods*, a profile only appears in the list of one of its words. This word is called the *key*. We can randomly pick one of the words as the key; this is the *Random Key Method*. If the frequency distribution of words in documents is known, we can also use the *Ranked Key Method* and store the profile in the list of the word with the lowest rank. The idea is to ensure that the more frequent words have fewer profiles associated with them, and thus, on the average, fewer profiles have to be examined per document. We emphasize that word rank information may or may not be available. Thus it is important to consider both types of strategies.

An inverted list posting for the Key Methods contains the profile identifier, the length of the profile, and the words except the key. Postings in the same list are stored sequentially in blocks.

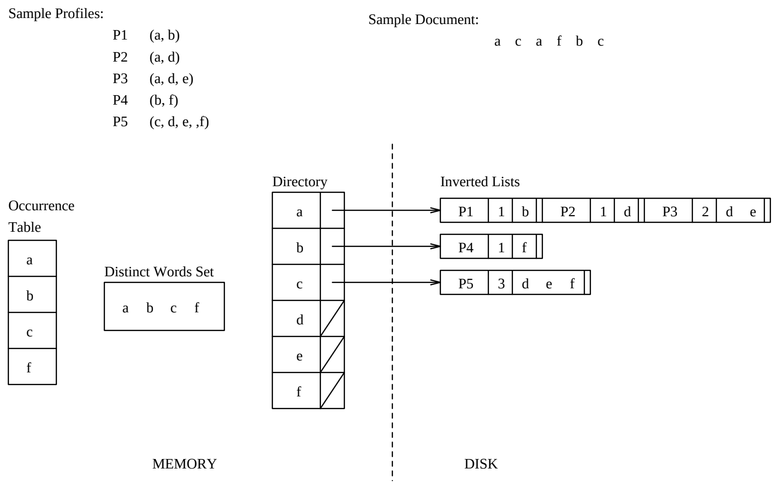

Figure 2: Data Structures for the Key Methods

For both Key Methods, when a document comes along, we build the distinct words set and the occurrence table. Then we index the directory to retrieve the inverted lists of each distinct word. For each profile in each list, we check if it matches the document (using the occurrence table).

Figure 2 shows the directory and the inverted lists (arranged using the Ranked Key Method, assuming word a has the lowest rank, b has the second lowest rank, and so on) for some sample profiles, and the occurrence table and distinct words set for a sample document. Notice for instance that P_1 is in the a list but not in the b list. To process the document, we look up words a, b, c, and f in the directory. Word f has no profiles associated with it, so its list is empty. We read in the list of, say, a first and check the profiles in the list. For the first profile in this list (P_1), we look up the word b in the occurrence table, which is in the document. As there are no more words in the profile, P_1 matches the document. Then we go on to the next profile, P_2. We look up d, which is not in the document; thus P_2 does not match. Finally, we look at P_3. There are two words we need to check: d and e. Suppose we look up d first. It is not in the document and so P_3 does not match the document. The lists for words b and c are processed in a similar fashion.

3.4 The Tree Methods

With users interested in the same area subscribing similar profiles, it is likely that a large number of profiles have similar words. Using this observation, it is possible to store the profiles more compactly than lists. A possibility is to use a trie-like structure [1].

Consider a profile P of K words, $(w_1, w_2, ..., w_K)$. We call $(w_1, ..., w_i)$ a *prefix* of P, $0 \leq i \leq K$; the corresponding *postfix* is $(w_{i+1}, ..., w_K)$. For example, $()$, (a), and (a, b) are all prefixes of profile (a, b), with corresponding postfixes (a, b), (b), and $()$ respectively. We also call $(w_1, ..., w_j)$ a prefix of $(w_1, ..., w_i)$, $i \geq j$. A prefix $(w_1, ..., w_i)$ *identifies* P if $i = K$ or there is no other profile, except those identical to P, that has $(w_1, ..., w_i)$ as a prefix. The shortest prefix that identifies a profile is the *identifying prefix* for that profile. Note that a prefix is the identifying prefix of two profiles if and only if they are identical.

The identifying prefixes of the profiles are organized into a tree. The root is at level 0. A node n at level i corresponds to a prefix $\sigma = (w_1, ..., w_i)$ of some identifying prefixes. All prefixes identical to σ are represented by the same node n. Its children are nodes corresponding to prefixes $(w_1, ..., w_i, v)$ of some identifying prefixes.

Node n has the following fields: children, which is a list of $(v,\ p_n(v))$ pairs, where v is a word such that $(w_1, ..., w_i, v)$ is the prefix corresponding to a child of n, and $p_n(v)$ is a pointer to that child; profiles, a list of profiles of which σ is the identifying prefix (note that profiles in the list must all be identical); length, the length of the postfix of the profiles σ identifies; and postfix, words that make up the postfix of the identified profile(s). The last two fields (length and postfix) exist only if profiles is not an empty list.

Figure 3A shows the tree of the identifying prefixes for the sample profiles in the previous example. For instance, the node labeled x represents the prefix (a). Node y is for prefix (a, b) and identifies P_1. Figure 3B shows the internal structure of the same tree in Figure 3A. For example, from the root, if we follow a's pointer, we get to node x representing the (a) prefix. Following b's pointer from this node, we get to node y for prefix (a, b). This node has an empty children list, a profiles list (identifying P_1), and a length field of 0, indicating the postfix is empty (i.e., P_1 is only (a, b)).

If we look at the tree as an index structure, we can see that the root corresponds to the directory, while each of its subtree forms a tree-structured inverted "list." For example, in Figure 3B, the uppermost subtree is formed by the inverted set of profiles that start with a. To store this index structure, we place the directory (root), which is implemented as a hash table, in main memory. Each subtree is packed into contiguous blocks. (This can be done in a variety of ways; for our simulations, we assume a particular layout in breadth first order, as detailed in [16].) The directory thus maps a word w to the disk location of the subtree of profiles that start with w.

When a document arrives, we first construct the distinct words set and occurrence table. We index the directory for each distinct word and read into memory its subtree. To find the matching

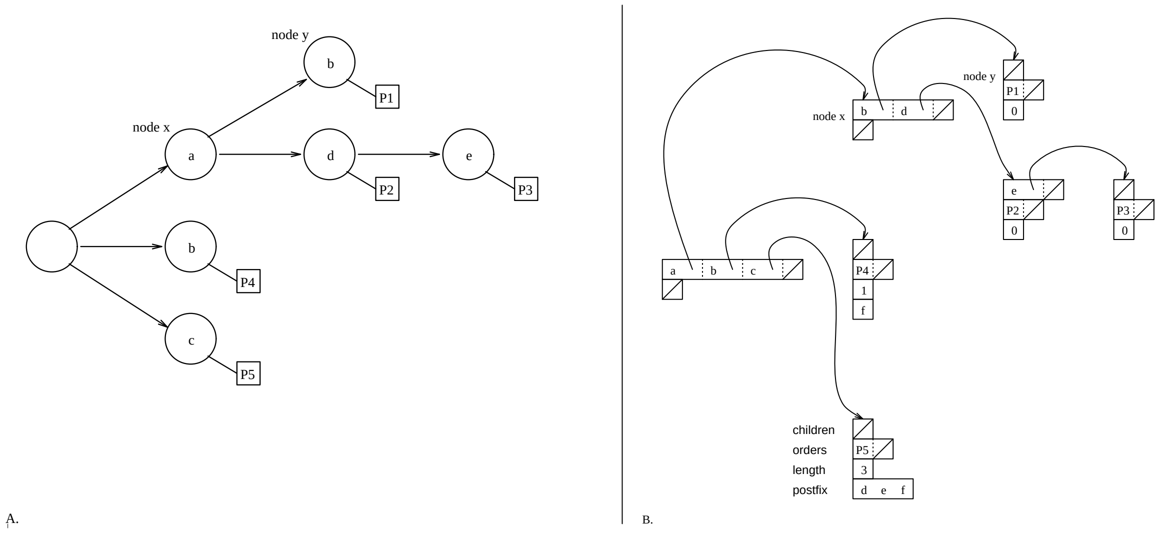

Figure 3: A Tree of Identifying Prefixes and Identified Profiles

profiles in the subtree, we do a breadth first search [1]. We keep a queue of pointers to nodes that are to be visited in the search. The queue is initialized to contain the pointer to the root of the subtree.

We repeat the following until the queue becomes empty. We get the first pointer from the queue and look at the node it points to. We check the children list for words that are in the document. Pointers that corresponds to words that are in the document are appended to the queue. Next we check the postfix (if not empty) to see if all the words are in the document. If yes, the profiles in profiles match the document. If the postfix is empty, then the profiles also match.

Consider the tree in Figure 3B and the sample document as in the previous example. Suppose we try to match the profiles in the subtree of a. We look at the root of the subtree (node x). We check the words in the children list. Since b is in the document and d is not, we append the pointer corresponding to b (which points to node y) to the queue. There is no postfix to check and no profile is identified, so we are done with this node. Then we get the first (and only) pointer in the queue, which points to node y. Node y does not have any children, so we look at the postfix. The postfix is empty, so the identified profile P_1 matches the document. The queue is now empty, and we are done with this subtree.

The tree structure saves space when there are a lot of common prefixes among the profiles. One heuristic to increase the number of common prefixes is to sort the words in profiles, e.g. with increasing word identifier. For example, if we sort the two profiles (b, a, d, c) and (c, b, e, a), we get the common prefix (a, b, c). If rank information is available, we can also sort them by reverse rank order, i.e., lowest ranked word first. This has the advantage that (similar to the Ranked Key Method) more profiles are put in the subtrees of the lower ranked words, which are looked up less often. This is called the *Ranked Tree Method*, and the former (sorting by some order without rank information) is called the *Random Tree Method.*

4 Performance Evaluation

We have evaluated the performance of the above methods through analysis and simulations. Below we first describe our document and profile models and the evaluation metrics. (These models are similar to the document and query models used in [13].) Then we present the analysis and details of the simulations.

4.1 Document Model

Let T be the size of the vocabulary V (i.e., $|V| = T$). Each word in V is uniquely represented by an integer w, $1 \leq w \leq T$. The probability that any word appears is described by the probability distribution Z. We rank the words in non-increasing order of frequencies, i.e., $\forall w, v, 1 \leq w < v \leq T$, we have $Z(w) \geq Z(v)$; for convenience, we use the rank to identify the words. We assume the frequency distribution follows Zipf's Law [18], i.e.,

$$Z(w) = \frac{1}{w \sum_{v=1}^{T} 1/v}.$$

A new document has W words and we assume its words follows the same distribution, Z; it is generated by a sequence of W independent and identically distributed trials. Each trial produces one word from V.

4.2 Profile Model

While the frequency distribution of words in large collection of text is well-studied [18], there is very little published about the frequency distribution of words in queries, and even less about words in profiles. In this paper, we adopt the query model used in [13].

The extremely infrequent words (in documents) typically are misspellings or typos. Thus, we assume profiles, which are long-term queries, do not use these words. On the other hand, sometimes even frequent words appear in queries; e.g., "The Who" or "NOW" (The National Organization of Women) [5]. We model this by assuming that profile words are chosen from the set $U = \{1, ..., S\}$, termed the queried vocabulary, out of the vocabulary $V = \{1, ..., T\}$, $S < T$. (Recall that we are identifying words by their ranks.) We further assume that each word in U is equally likely to be chosen for a profile. Hence, we assume that a profile is a set of K words chosen randomly without replacement from the queried vocabulary U.

Parameter	Base Value	Description
T	1.8 Million	size of vocabulary
W	12000	# words per document
S	18000	size of queried vocabulary
N	300000	# profiles
K	5	# words per profile
I	$\lceil \log_2 N \rceil$ bits	length of profile identifier
L	8 bits	# bits to represent length of profile
X	$\lceil \log_2 T \rceil$ bits	# bits to represent a word
P	20 bits	# bits for a pointer in the Tree Methods
B	4096 bits	# bits in a disk block
r	10	$\frac{\text{\# instructions for a hash table look-up}}{\text{\# instructions for an array access}}$

Table 1: Summary of Parameters Used in Analysis and Simulations

The number of profiles in the system is N. To simplify the study of the effect of profile size on performance, we assume all profiles have the same length, i.e., K is fixed for all profiles.

Table 1 summarizes our parameters, including ones for modeling the size of disk block and record/posting/node fields. The Base Value column represents the base case scenario for our study. (In Section 5 we vary these parameters over wide ranges.) The value for the size of the vocabulary (T) is from a 8.9 GB document database described in [4]. For the profile model, the size of the queried vocabulary (S) is chosen to to be 18000, covering over 96% of the occurrences of words in the document database. We assume I and X are the minimum possible. In actual implementations, these numbers could be somewhat larger, for example to simplify packing/unpacking operations, or to allow for growth in the number of profiles and the size of vocabulary.

4.3 Metrics

To evaluate our SDI index structures and algorithms we use three fundamental cost metrics, each reflecting the storage, CPU, and I/O resources utilized. For the storage cost metric, we look at how much disk space each structure takes. Although main memory space requirements of the methods differ, we will not consider these. (That is, we assume main memory requirements are relatively minor, and hence, all main memory structures would fit in a modern computer.)

For the I/O cost metric, we measure the number of I/Os it takes to process an expected document. For the CPU cost metric, we do not count every single CPU operation. Instead we focus on the "inner loop" of the document matching process and count the number of times our key main-memory data structures are probed. We count as one probe a directory access (to get the inverted set of a word), an occurrence table look-up (to check the presence of a word in a document), a read of the distinct word set (to get a distinct word from a document), and an access to the TOTAL/COUNT arrays. Note that counting probes is analogous to the way CPU costs of sorting algorithms are compared. (For sorting algorithms, one counts the number of compare instructions executed.) The higher the number of probes (or the number of compares for sorting), the higher the overall CPU cost is expected to be.

Although a directory probe is comparable to an occurrence table probe (both structures are hash tables indexed by word identifier), a probe to the distinct word set (which can be implemented as an array) or to the TOTAL/COUNT arrays should be less costly. To take this into account we introduce a scale factor r that specifies how many array probes are comparable to a hash table probe. We use the number of normalized probes as the CPU cost metric. The last row of Table 1 lists the r parameter.

In summary, we look at three metrics:

- the expected total disk space required (in number of blocks),
- the expected number of disk I/Os needed to match a document, and
- the expected number of normalized probes to main memory data structures performed in processing a document.

By evaluating these metrics we will understand the fundamental differences between the indexing strategies. In addition, they can identify good strategies for particular scenarios. For example, in an I/O bound system, the I/O metric would be critical, and hence the strategy which minimizes I/Os should be selected. If the number of profiles is expected to be relatively small, perhaps all of the disk structures can be cached in main memory. In this case, the CPU metric would be the decisive one. If the expected document arrival rate is low, then perhaps neither the CPU or the I/O metrics would be critical, and the main criteria should be the storage cost.

Note incidentally that we will not compute expected response times or throughputs for document processing. These would be useful metrics to compute after the fundamental tradeoffs between strategies are explored, and when one has a particular hardware configuration to evaluate. That is, computing response times and throughputs requires a large number of additional parameters (like CPU clock speed, disk seek times, disk scheduling policy, layout of blocks on the disk, number of CPU instructions for each operation, arrival rate of documents). We believe that brining in all these parameters in this paper would obscure the fundamental tradeoffs.

4.4 Analysis

Except those for the Tree Methods, the CPU metric of the Ranked Key Method, and those given in Section 5.5 (in which we modify the profile model), the results in the Section 5 were obtained by deriving analytical solutions and then numerically evaluating the expressions. This subsection contains the details of the analysis for the first two metrics. The analysis for the third metric has a similar flavor and is omitted here due to space limitations. The full analysis appears in [16]. (Relevant sections are included in Appendix A for refereeing.)

4.4.1 The Brute Force Methods

The two Brute Force Methods perform identically with respect to the first two metrics. Recall that we store in a record the profile identifier, the length of the profile, and the words in a profile, and that records are stored sequentially in contiguous blocks. Since profiles have K words, the space required to store all profiles (in blocks) is

$$M_{BruteForce} = \lceil \frac{N(I + L + KX)}{B} \rceil .$$

Since all the profiles have to be read to process a document, the number of blocks read per document is the same:

$$R_{BruteForce} = \lceil \frac{N(I + L + KX)}{B} \rceil .$$

4.4.2 The Counting Method

Before we analyze the Counting Method, let us derive a general expression for the expected length of an inverted list in the Counting and Key Methods. We consider the question: given $\mathcal{N}$ postings, each of size $\mathcal{R}$, that are to be placed in a number of lists, what is the expected number of blocks in a certain list, if the block size is $\mathcal{B}$ and the probability that a posting falls in this list is p? Let us denote this expression by $\mathcal{F}(\mathcal{N}, p, \mathcal{R}, \mathcal{B})$.

Let random variable $\mathbf{X}$ be the number of postings in the list. $\mathbf{X}$ follows the binomial distribution $\mathrm{Bin}[\mathcal{N}, p]$. Let random variable $\mathbf{Y}$ be the number of blocks in the list. $\mathbf{X}$ and $\mathbf{Y}$ are related by

$$\mathbf{Y} = \lceil \frac{\mathcal{R}\mathbf{X}}{\mathcal{B}} \rceil.$$

We want to find $E[\mathbf{Y}]$. First we compute the following probability.

$$\begin{aligned}
\Pr\{\mathbf{Y} = y\} &= \Pr\{\lceil \frac{\mathcal{R}\mathbf{X}}{\mathcal{B}} \rceil = y\} \\
&= \Pr\{y - 1 < \frac{\mathcal{R}\mathbf{X}}{\mathcal{B}} \leq y\} \\
&= \Pr\{\frac{(y-1)\mathcal{B}}{\mathcal{R}} < \mathbf{X} \leq \frac{y\mathcal{B}}{\mathcal{R}}\} \\
&= \sum_{\frac{(y-1)\mathcal{B}}{\mathcal{R}} < x \leq \frac{y\mathcal{B}}{\mathcal{R}}} \mathrm{Bin}[x; \mathcal{N}, p].
\end{aligned}$$

To efficiently evaluate the last sum, we use the normal approximation when appropriate, and the poisson approximation when that is not applicable. Finally, the expression that we are after is thus

$$\begin{aligned}
\mathcal{F}(\mathcal{N}, p, \mathcal{R}, \mathcal{B}) &= E[\mathbf{Y}] \\
&= \sum_{y \geq 0} y \Pr\{\mathbf{Y} = y\} \\
&= \sum_{y \geq 0} (y \sum_{\frac{(y-1)\mathcal{B}}{\mathcal{R}} < x \leq \frac{y\mathcal{B}}{\mathcal{R}}} \mathrm{Bin}[x; \mathcal{N}, p]).
\end{aligned} \qquad (1)$$

Now we proceed with the analysis of the Counting Method. For a particular list, the maximum number of postings that can be placed in it is N. (Although the total number of postings in the index structure is NK, at most only N of them can be on the same list.) The probability that a posting is in a list is $\frac{K}{S}$. Only the profile identifier is kept in a posting, so the posting size is I.

The expected number of blocks in each list is thus $\mathcal{F}(N, \frac{K}{S}, I, B)$. As there are S inverted lists, the expected total space requirement is (in number of blocks)

$$M_{Counting} = \mathcal{F}(N, \frac{K}{S}, I, B) \times S.$$

For the expected number of I/Os required per document, we need to know what is the expected number of inverted lists retrieved per document. First we note that, given a document D, the probability that a particular word w (recall that we identify words by ranks) is in D is given by

$$(1 - (1 - Z(w))^W). \tag{2}$$

This is because for any word in D, the probability that it is not w is $(1 - Z(w))$; so the probability that w does not appear in D is $(1 - Z(w))^W$, as D has W words. Using (2), we derive W_S, the expected number of distinct words in a document that appear in some profiles, as

$$W_S = \sum_{w=1}^{S} (1 - (1 - Z(w))^W).$$

Since the inverted lists of these words are read in, the expected number of blocks read per document is

$$R_{Counting} = \mathcal{F}(N, \frac{K}{S}, I, B) \times W_S.$$

4.4.3 The Key Methods

In the Key Methods, a posting contains the profile identifier, the length of the profile minus 1, and the words in the profile, except the key. The posting size is $(I + L + (K - 1)X)$.

For the Random Key Method, the probability that a posting is in a list is $\frac{1}{S}$. Using (1), the expected number of blocks per list is $\mathcal{F}(N, \frac{1}{S}, I + L + (K - 1)X, B)$. Thus the total space required is

$$M_{RandomKey} = \mathcal{F}(N, \frac{1}{S}, I + L + (K - 1)X, B) \times S,$$

and the average number of blocks read to process a document is

$$R_{RandomKey} = \mathcal{F}(N, \frac{1}{S}, I + L + (K - 1)X, B) \times W_S.$$

Using the Ranked Key Method, the probability that a posting is in the list for a word w is $\binom{w-1}{K-1}/\binom{S}{K}$. This is the probability that a profile has w as its lowest ranked word. Thus the expected length of the inverted list of a word w is (in number of blocks)

$$\mathcal{F}(N, \frac{\binom{w-1}{K-1}}{\binom{S}{K}}, I + L + (K-1)X, B).$$

If we add up the expected size of each list, we obtain the total space required:

$$M_{RankedKey} = \sum_{w=1}^{S} \mathcal{F}(N, \frac{\binom{w-1}{K-1}}{\binom{S}{K}}, I + L + (K-1)X, B).$$

The expected number of blocks read to process a document D is, using (2),

$$\begin{aligned} R_{RankedKey} &= \sum_{w=1}^{S} \Pr(w \text{ appears in } D) \times (\text{number of blocks in list of } w) \\ &= \sum_{w=1}^{S} (1 - (1 - Z(w))^W) \times \mathcal{F}(N, \frac{\binom{w-1}{K-1}}{\binom{S}{K}}, I + L + (K-1)X, B). \end{aligned}$$

4.5 Simulations

Since we found the analysis of the Tree Methods and the CPU metric of the Ranked Key Method intractable, we use simulations to obtain the results. In Section 5.5, we modify the profile model and the results there are also obtained by simulations. Simulations are also used to validate the analysis in the above section. The simulation results did match the analytical ones. All results in the paper are analytical ones when they are available; simulation results are given when not.

We wrote our simulation program in C. The program first generates N profiles according to the profile model, and then computes the size of the index structures needed to store the profiles. Next the simulation program generates a document according to the document model and counts the number of disk reads needed to match it against the N profiles. For each scenario we have tested, the program is run enough times (with different random number generator seeds) to make sure that the results are within $\pm 5\%$ of the true values, with a 95% level of confidence.

5 Results

The results for the base case scenario are given in Table 2. For space requirements, we can see the Brute Force Methods require the least disk space. In the Brute Force Methods, profiles are packed into sequential blocks, so no space is wasted (except in the last block). In other methods, as profiles are stored in different lists or subtrees, and these lists or subtrees are packed into blocks, disk space is wasted in the last block of every list or subtree. This internal fragmentation leads to an increase of about 75% to 85% in the space requirements. However, we remark that some of this wastage could be reduced by packing several of the short lists into a single block.

Method	Size (Blocks)	I/Os	Normalized Accesses
Random Brute Force	9668	9668	356375
Ranked Brute Force	9668	9668	317492
Counting	18000	2849	60913
Random Key	18000	2849	63547
Ranked Key	16475	1139	24789
Random Tree	18029	2841	62958
Ranked Tree	18029	1213	24737

Table 2: Results for the Base Case

The expected size of the tree for the Random Tree Method is equal to that for the Ranked Tree Method. This is because profiles are generated without rank information, so no matter what sorting order we use to organize the tree, the expected size should be the same.

With respect to the number of blocks read in the matching process, the indexing methods outperform the Brute Force Methods by a factor of 3 to 8. Note that the number of blocks read in the Counting Method is the same as the Random Key Method. This number is equal to the expected number of distinct document words that appear in profiles, and is thus also the expected number of lists read in to process a document. This can be explained by the fact that, under the base case values, the inverted lists in these two methods all fit in 1 block.

The Ranked Key Method performs the best in terms of number of I/Os per document. The Ranked Tree Method requires slightly more I/Os. We attribute this to the fact that the model of random independent profiles does not generate a lot of profiles that share common prefixes, even though the number of profiles is so large.

The number of I/Os required to process a document appears to be quite high for the various methods. However, we remark that caching of blocks between successive documents will reduce these

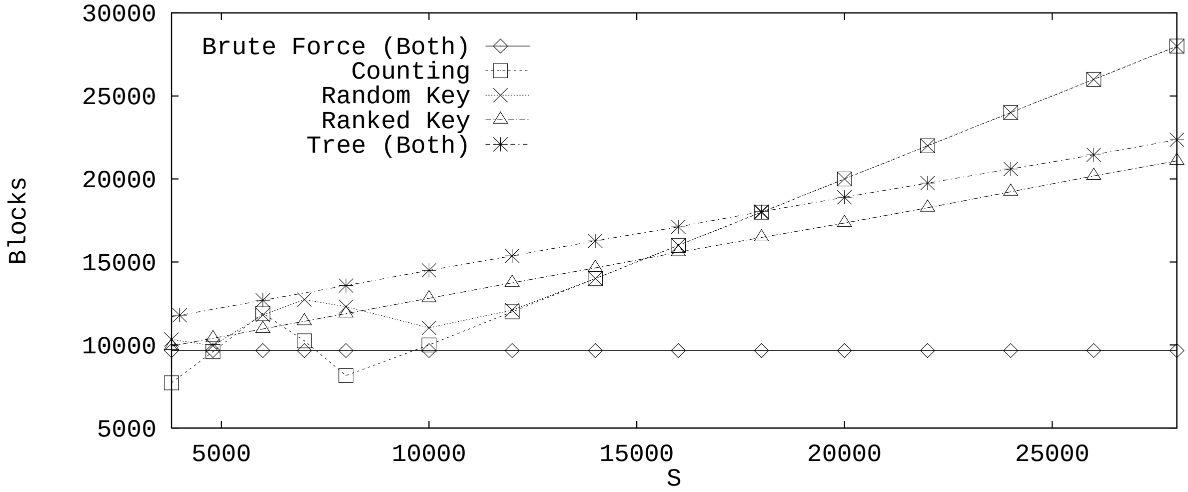

Figure 4: Total Disk Space Required vs S

numbers. Another reason the required number of I/Os is large is that in our base case documents are large (12000 words is roughly twice the size of a conference paper). As documents become smaller, the number of I/Os for the indexing methods decreases in direct proportion (but not for the Brute Force Methods).

For the number of normalized accesses, the Ranked Tree Method performs the best, and all indexing methods are superior to the Brute Force Methods.

Next we perform a sensitivity analysis, investigating the impact of several important parameters on performance. (The size of the incoming document, W, does not affect the relative performance of the methods, and so we omit the results here.) We also modify the profile model to study how similarity among profiles affects performance.

5.1 Varying S

The first parameter that we vary is S, the size of the queried vocabulary. The other parameters are kept at their base values. Figures 4, 5, and 6 show the performance of the methods. The range of S studied is actually wider than what is shown, but is truncated here to display the interesting range in more detail.

In Figures 4 and 5, there are some peculiar zig-zig patterns in the Counting and Random Key graphs. This can be explained as follows. Consider the last spike in the Random Key graph in

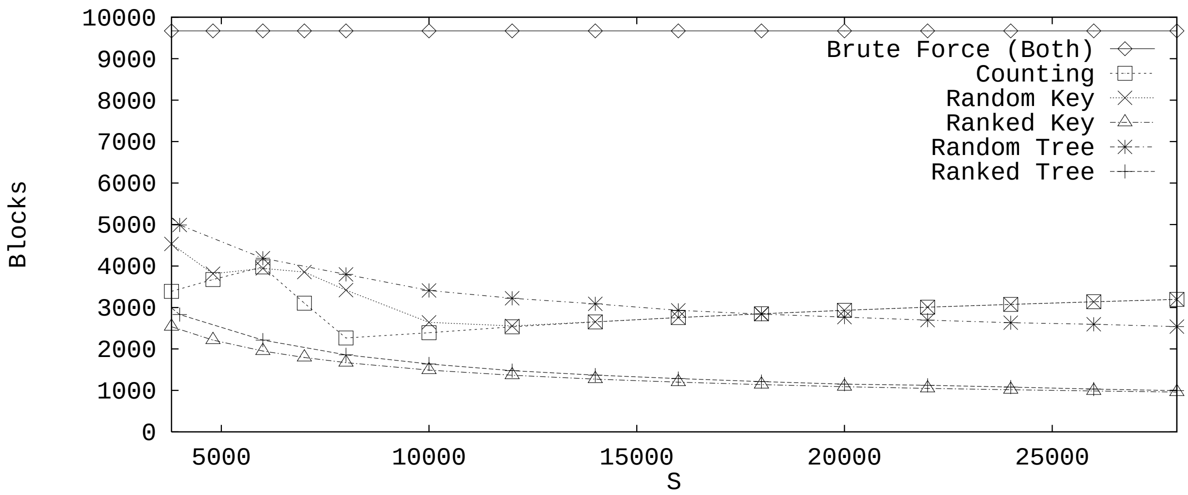

Figure 5: Number of Blocks Read Per Document vs S

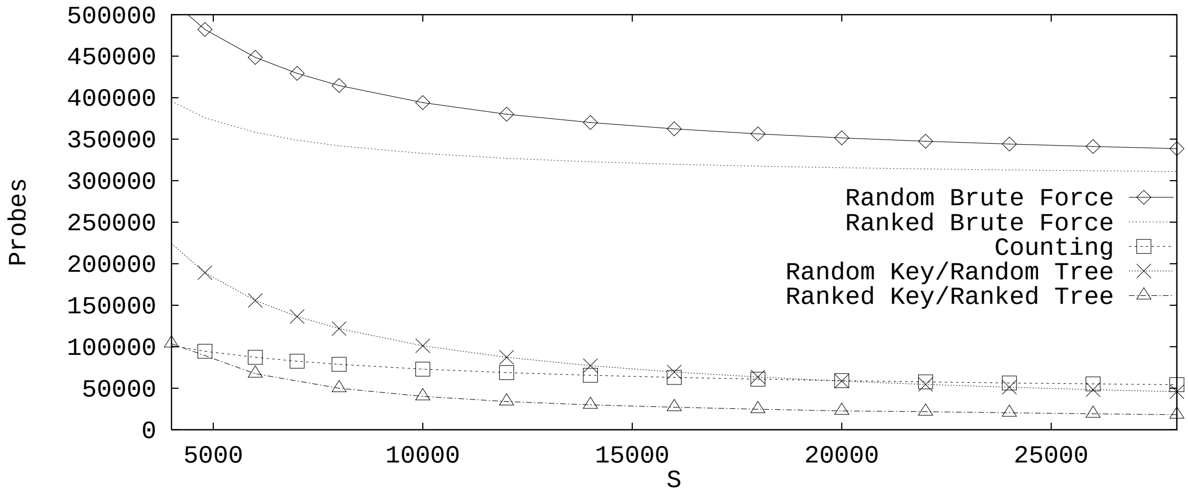

Figure 6: Number of Normalized Accesses vs S

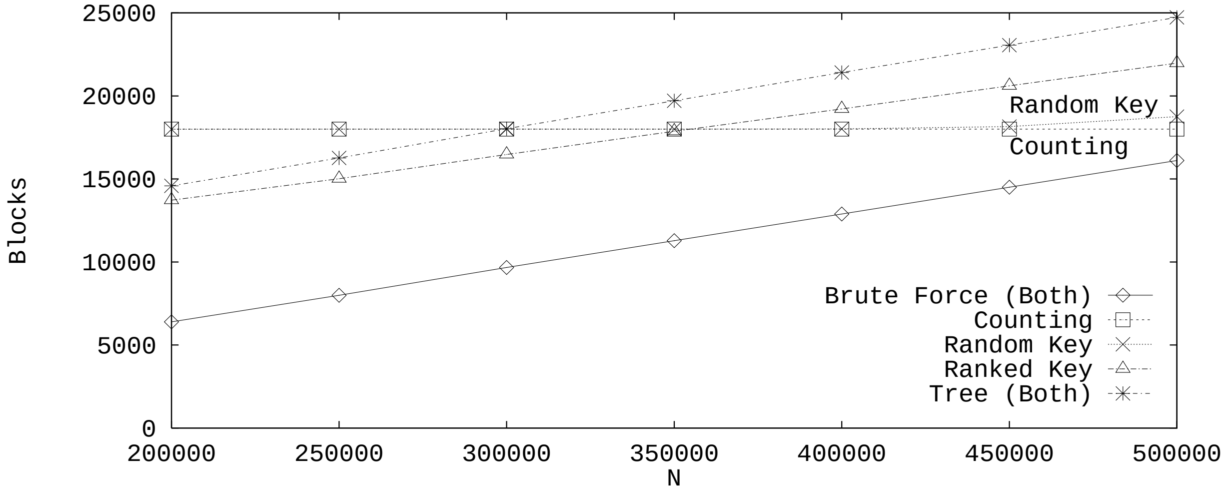

Figure 7: Total Disk Space Required vs N

Figure 4. Before the spike, there is a steady rise in the total size. This is because as S increases, so does the number of inverted lists. This leads to an increase in the number of blocks. At the same time, the number of profiles per list decreases (the total number of profiles is fixed). At some point, the lists begin to shrink their lengths by one block, and this leads to the drop in the total size.

In general, the indexing methods require more space as S increases, due to an increasing number of inverted sets.[3] On the other hand, the general trends for the number of disk reads and the number of normalized accesses per document are downward. This is because when S is large, it becomes more likely that profiles contain infrequent words that do not appear often in documents.

The ranked strategies always perform better than their counterparts without rank information. This will be seen to be true under other scenarios studied below. (Of course, if rank information is not available, then ranked methods cannot be implemented.)

5.2 Varying N

Next we study the effect of varying N, the number of profiles. The base values are used for the other parameters. Figures 7 and 8 show the performance of the various methods with respect to the first two metrics as N is varied. The relative performance of the methods with respect to the number of

[3] Again, we stress that such increase due to internal fragmentation can be reduced by packing several short lists into one block.

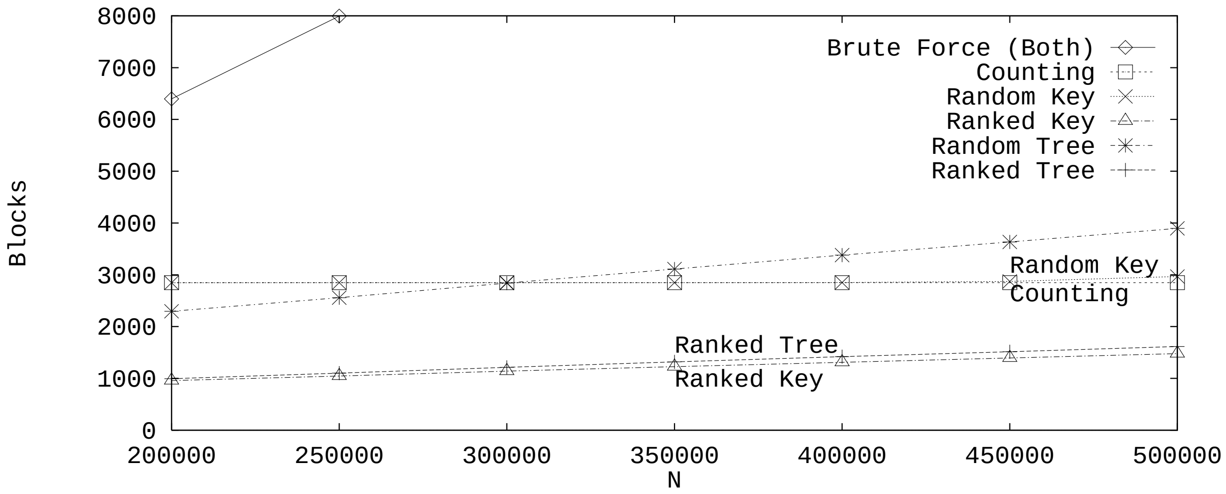

Figure 8: Number of Blocks Read Per Document vs N

normalized accesses does not vary with N and thus the results are omitted.

For a wide range of values of N, the inverted lists in the Counting and Random Key Methods all fit in one block. This explains why their graphs stay flat at the constant value of S in Figure 7. Similar reasoning explains why in Figure 8 the number of I/Os for the two methods is equal to the expected number of distinct document words that appear in profiles. However, notice that for very large values of N (greater than 450000), the space requirement and the number of I/Os for the Random Key Method begin to rise, as some of the lists begin to occupy more than 1 block. We expect to see a similar rise in the graphs of the Counting Method for larger N.

One would expect that as N grows, there would be more overlapping profiles, and the Tree Methods would do better. However, the Tree Methods do not perform very well, even when the number of profiles is very large. This again shows that the independent profile model does not generate enough similarity among profiles to make the Tree Methods attractive.

5.3 Varying K

Next we investigate the effect of varying the length of the profiles. We vary K from 1 to 10 and compare the performance of the various methods. The results are shown in Figures 9, 10, and 11.

In Figures 9 and 10, we notice some surprising results: with profile size increasing from 1, the space requirement and the number of disk accesses for all the indexing methods not only do not

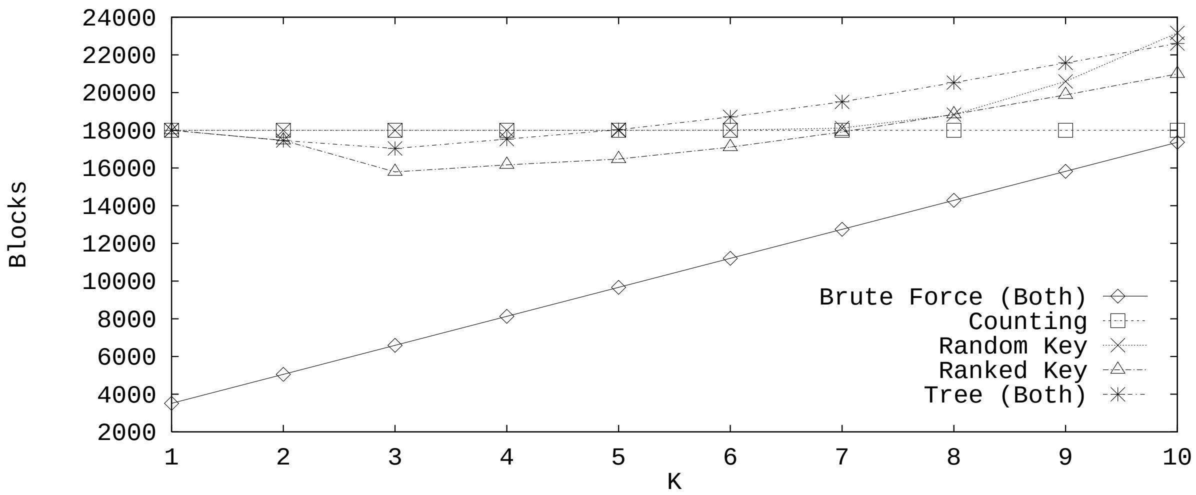

Figure 9: Total Disk Space Required vs K

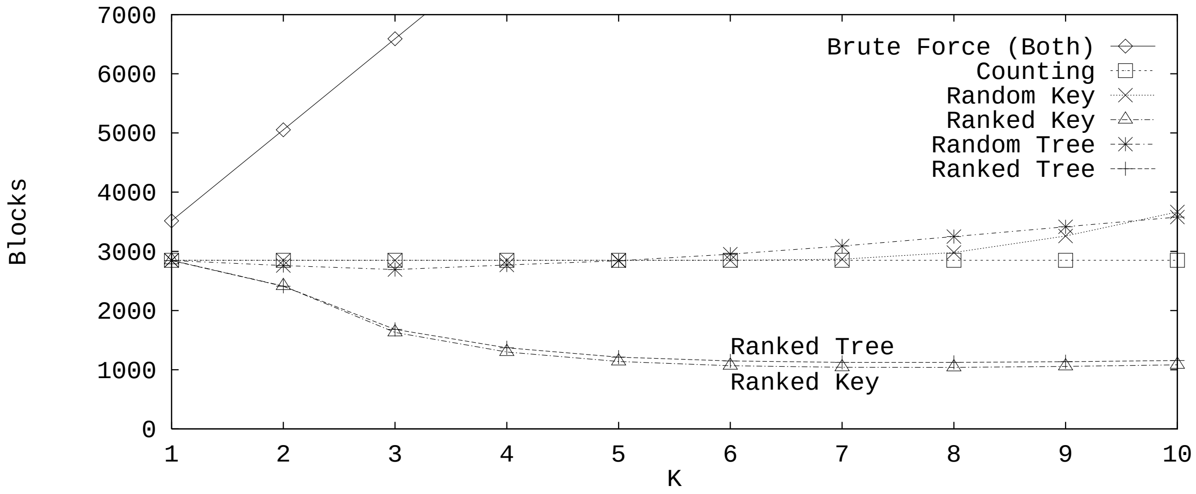

Figure 10: Number of Blocks Read Per Document vs K

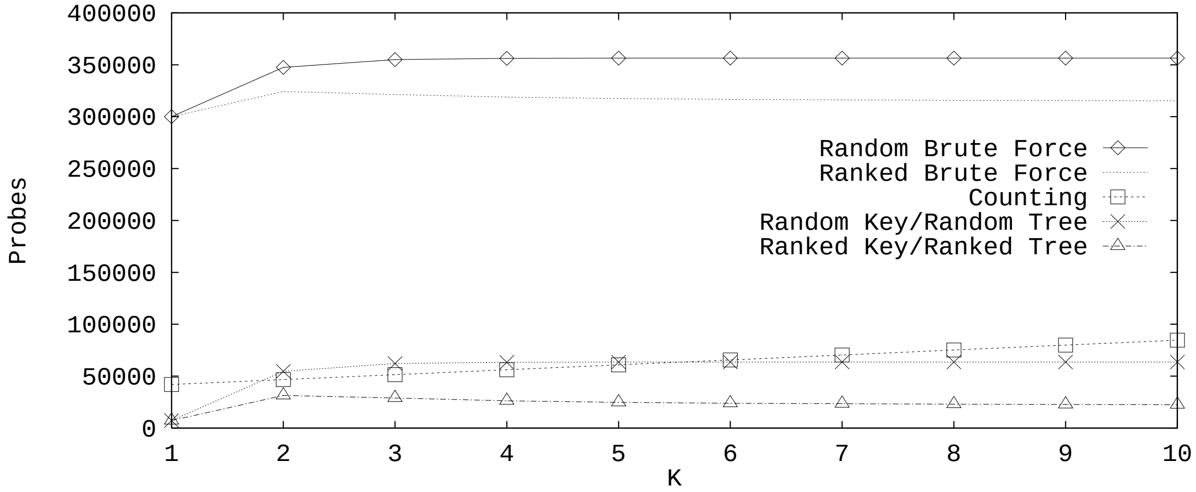

Figure 11: Number of Normalized Accesses vs K

increase, but actually decrease for some methods. This can be explained as follows. For $K = 1$, all inverted sets fit in 1 block because the profiles are short. Thus the number of blocks is equal to S in all indexing methods and the number of I/Os is equal to the expected number of distinct words in a document that are queried by some profiles. As K increases to 2, for the Counting and Random Key Methods, the lists can still fit in one block, and thus their graphs remain flat. However, for the Ranked Key and the Tree Methods, there is a choice as to where to place a profile. Some of the sets become empty while others that grow still fit in one block. The net result is a decrease in the total number of blocks required. The effect is even more prominent in the number of I/Os required for the ranked methods, since the empty lists are associated with the more frequent words and are thus looked up more often. As K becomes larger, the sets begin to occupy more than 1 blocks, leading to an increase in the space requirement or the number of I/Os.

For the number of normalized accesses, we would expect it to grow with the profile size. However, the results show that all methods, except Counting, are relatively insensitive to K, and some even require fewer accesses with increasing K. This is because during the matching process, we stop processing a profile as soon as a mismatch is found. Thus the increasing number of words per profile only leads to a very slight increase in the number of accesses required. For the methods using rank information, as K increases, it is more likely that there are infrequent words in a profile. As they are looked up first, the matching process can terminate early, resulting in even fewer accesses.

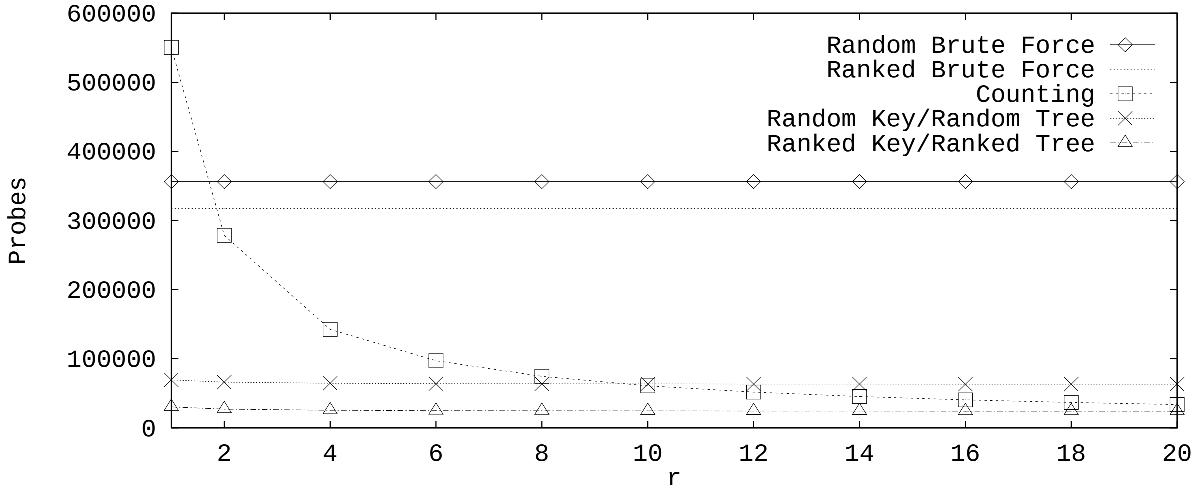

Figure 12: Number of Normalized Accesses vs r

5.4 Varying r

The last parameter that we vary is the normalizing factor r, which affects the performance of the methods with respect to the the number of normalized accesses per document. The results are shown in Figure 12. When $r = 1$, it means a hash-table lookup is comparable to an array access. When $r = 10$, it means a hash-table lookup is comparable to 10 array accesses. We have carried out an experiment and estimated r to be about 1.6 on our machine, a DECstation 5000/240.

As we can see, the performance of the Counting Method is very sensitive to r. On a machine where the hash function evaluation is fast (e.g., a machine with a math coprocessor), the Counting Method is bad, even worse than the Brute Force Methods. However, on a machine where a hash table lookup is a much more expensive operation than a simple array access (e.g., division required for hashing is done by software), the Counting Method performs better than the other methods without rank information.

5.5 Modifying the Profile Model

To model the similarity among profiles, we modify the basic profile model as described below. We still assume that a profile is a conjunction made up of K words, where K is fixed for all profiles.

Among the N profiles, a certain fraction (fixed at 10% in our experiments) of them are generated

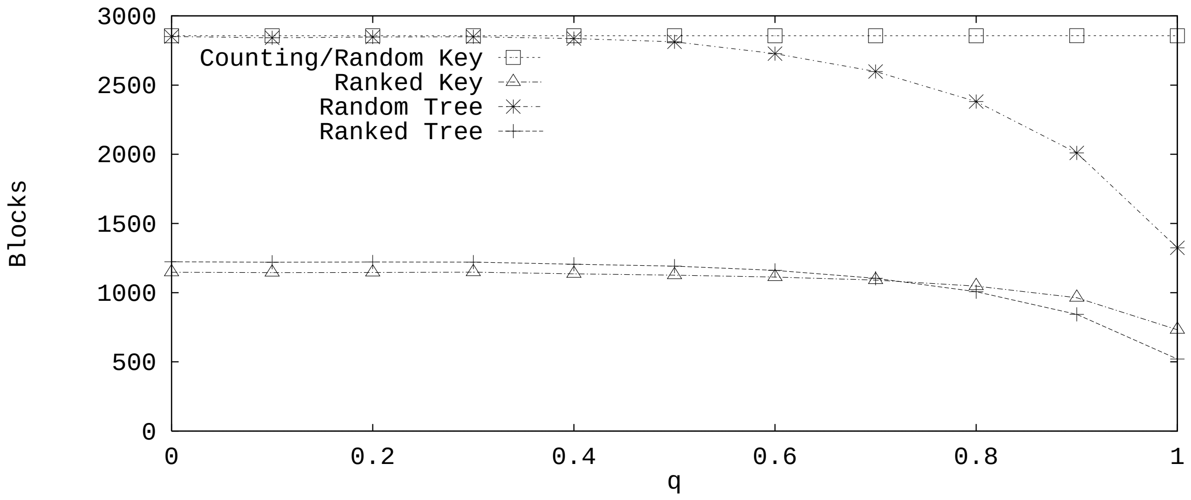

Figure 13: Number of Blocks Read Per Document vs q

using the basic model, i.e., assuming they are independent. They form the different areas of interests that the users may have, and are called the *base profiles*. For each of the rest of the profiles, we assume that it will be similar to a base profile. We randomly pick one base profile and mold the new profile after it. The similarity parameter q controls how similar the new profile and the base profile are. For each word in the new profile, there is a probability q that it is the same as the corresponding word in the base profile. If it is not, then a word is randomly drawn from the queried vocabulary. Hence by varying q from 0 to 1, we can control the similarity among the profiles. If q is 0, this model degenerates to the original model.

Simulation results show that indeed both Tree Methods improve as the degree of similarity grows. Figure 13 shows the performance with respect to the number of I/Os per document under the modified model, as q is varied from 0 to 1. (The graph of the Brute Force Methods is omitted to show the variations in the other graphs better.) The Random Tree Method becomes better than the other methods without rank information for q above 0.5. The Ranked Tree Method becomes the best method when the similarity parameter is above, say, 0.8.

5.6 Summary

Here we summarize the strengths and weaknesses of the various methods.

The Brute Force Methods are good for storage, but not for execution. The number of disk I/Os and the number of normalized accesses required per document are often an order of magnitude worse than the indexing methods, especially when the number of profiles is large. However, if there are relatively few profiles and they are short (e.g., say less than 50000 profiles, each with fewer than 5 words) and the size of documents is large, then the Brute Force Methods may be a good choice.

Under the basic profile model, the Ranked Key Methods almost always requires the fewest disk I/Os per document, and requires the second fewest normalized accesses per document (slightly more the Ranked Tree Method). For methods without rank information, the Random Key Method seems to be a good compromise between the total space required, the number of I/Os, and the number of normalized accesses. The Counting Method also performs well in terms of the total space required and the number of I/Os per document, but is very sensitive to the normalizing factor r with respect to the number of normalized accesses per document.

With independent profiles, the Random Tree Method generally requires more space than the other indexing methods without rank information. This also leads to a higher number of disk I/Os per document. The same is true in the comparison between the Ranked Tree and Ranked Key Methods. However, when there is a moderate degree of similarity between profiles, the Random Tree Method becomes best among methods without rank information; and when there is a high degree of similarity, the Ranked Tree Method surpasses the Ranked Key Method to become the best overall. Real user profiles may or may not have such similarity and we plan to investigate this issue further.

6 Extensions

6.1 Extensions Under the Boolean Model

6.1.1 Negation and Disjunction

Negation can be easily handled in the methods discussed above with slight modifications. For all the methods except Counting, the main addition is that, during matching, a negated word is taken to be "in" a document if and only if the word itself is not in the document. A full discussion of what needs to be changed in all the methods can be found in [16].

If negation is allowed, the space requirements for all the methods will be slightly higher (e.g., in the Brute Force Methods, we need a bit per every word identifier to indicate if the word is negated). As for the number of disk I/Os, the effect is unclear as it depends on the frequency of the NOT operator in profiles. If the NOT operator is not frequently used in profiles, then our results are still applicable.

As discussed in Section 2, a profile with logical ORs can be transformed into disjunctive normal form and treated as a collection of conjunctive profiles. Hence, the algorithms we have presented can still be used. However, this strategy (splitting profiles) may not be the most efficient, since more disk space is required to store the profiles, and this possibly will lead to an increase in the number of I/Os per document. To fully understand OR profiles, two issues need to be addressed:

- Are the losses for processing profiles as sets of conjunctive profiles really significant?
- How can the indexing algorithms be modified to handle OR profiles directly?

We plan to study these questions in the future. For now, our intuition is that more complex algorithms for OR processing will not be worth it, especially since the most common type of OR profile, using thesaurus expansion or truncation, can be handled as a special case (see below). Under this scenario, AND will arguably be the dominant operator used in profiles, and thus it is reasonable to optimize the index structures and algorithms for the AND operation.

6.1.2 Truncation and Thesaurus Expansion

A traditional information retrieval system allows extensions such as truncation and thesaurus expansion to enrich the class of queries. Conceptually, the word is expanded into a disjunction of a number of matchable words. For example, the profile with truncated term `PSYCH*` matches documents containing words like psychology, psychologist, psychiatry, and others.

To process a profile in which one or more of the words are to be expanded to a thesaurus class, we can either do the expansion when the profile is first entered, and store it in expanded form, or we can do the expansion at profile evaluation time. To process a profile with a thesaurused word using the second option, we look up a dictionary to find its thesaurus class. Then we check the words in the class one by one. The thesaurused word is taken to be in the document if and only if any one of words in the class is in the document.

Since the thesaurus class can be large, the first option may lead to a big increase in space required for the index structures. In the the second option, the expansion may not be needed at all if we do not look up thesaurused words until other words in the profile are looked up – some of them may not be in the document. Thus the second option seems more attractive.

All the methods we have presented, except Counting, can be easily extended to handle thesaurus expansion using the second option. For the Key and Tree Methods, the organizations of the index structures are just as before, except that a profile can only be put in the list of a word that is not thesaurused. During matching, thesaurused words are looked up last, and are taken to be "in" if any one of the words in the thesaurus class is in the document.

For the Counting Method, run-time expansion is not feasible. Hence, the first option must be used to handle thesaurus expansion.

For truncation, we have to do run-time evaluation since we do not know beforehand what the truncated word expands to. To match a truncated word, we need to be able to do range search over the words in the document, e.g., to match the truncated term `PSYCH*`, we do the range search for words that are "greater than or equal to" `PSYCH` and "less than" `PSYCI`. To allow this, words should be stored as strings of characters in the index structures and the the occurrence table is implemented as some data structure (e.g., binary tree) that allows range search. With this change, all the methods except Counting can be extended to handle truncation.

6.1.3 Batching

To process a number of documents as a batch instead of processing them one at a time may produce savings in terms of total number of I/Os required, at the expense of the timeliness of document dissemination. Below we briefly describe what needs to be done for the methods to handle batches, and then we evaluate the benefits of batching.

Suppose we process a batch of b documents, $(D_1, ..., D_b)$, at one time. We can extend the methods above to efficiently match the profiles against the b documents simultaneously. The idea is to build a simple combined occurrence table for the batch. The entry in this hash table for a word w contains a list of the documents in the batch that contain w. In other words, this combined occurrence table is an inverted index of the type used for conventional IR, except that here we assume that the index is small and can fit in memory. In particular, the list of documents where w occurs can be represented as a bit vector of b bits, with the i-th set to T if w is in D_i. Using this table, the methods can process batches in a way analogous to the no-batching case.

To illustrate, consider the Key Methods. First we find all the distinct words in the batch and build the combined occurrence table. Then, for each distinct word w in the batch, we read in its inverted list. To process a profile in the list, we start off with the bit vector of w and look up the remaining words in the occurrence table. The bit vectors are ANDed together. We stop as soon as all bits become F's. Bits that remain T after all words in the profile are looked up correspond to

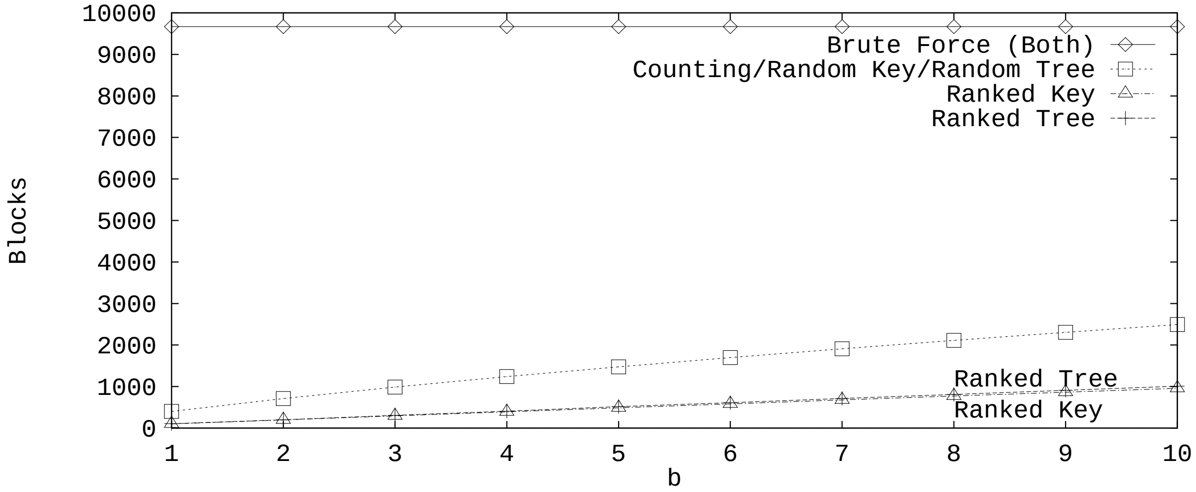

Figure 14: Number of Blocks Read Per Batch vs b, Small Document Case.

documents that match the profile. Other methods can similarly be extended; [16] covers the details.

The performance of batching depends a lot on the document size. Hence we study batching for small documents ($W = 1000$) and large documents ($W = 12000$). First, we investigate how the methods perform under different batch sizes with respect to the number of I/Os required per batch. Figure 14 shows the results for the small document case. The indexing methods remain far superior to the Brute Force Methods throughout. For the large document case, however, the results indicate that the indexing methods approach the performance of the Brute Force Methods as the batch size increases to 10. This is because as the batch size grows, there are lots of (distinct) words in the batch, and thus most of the inverted sets need to be examined.

Next, we assess the savings obtained by batching. That is, we compare the total number of I/Os required to match a number of documents with and without batching. Our results show that when documents are small, batching does not save much for the ranked indexing methods. For 10 documents, batching reduces the number of I/Os by about 3% for ranked methods. The reductions for methods without rank information are more significant, at 40%. Also, savings are more impressive when documents are large, with a reduction of 40 to 60% for a batch of 10 documents.

We conclude that batching may not be good for the ranked indexing methods, especially when documents are small, since timeliness is sacrificed for little savings in the total number of I/Os required. However, when methods without rank information are used, documents are large, and timeliness is not critical, it may be beneficial to use batching. Yet, when the batch size is large, the indexing methods approach the performance of the Brute Force Methods.

6.2 Extending to the Vector Space Model

In this subsection, we briefly discuss how we generalize two of the methods presented for the boolean model to work under the VSM. First we give a brief summary of the VSM (the reader is refered to [12] for an in-depth description of the model), then we sketch the generalized strategies, and finally we compare the performance of the strategies under the two models. In reference [17], we give the details of the strategies and compare their performance in many different scenarios.

6.2.1 VSM applied to SDI

In the VSM, documents are represented by vectors. The dimension of such document vectors is equal to the total number of words available to identify their content. For each word in a document, a weight is assigned as an estimate of the "importance" of the word to this document. A profile is similarly represented. For convenience, we follow the convention of writing a document or profile as a vector of (word, weight) pairs; those words not listed have weights equal to 0. Thus, a profile P with p non-zero weighted words can be written as $P = ((w_1, x_1), ..., (w_p, x_p))$. For instance, in the profile $P = ((\text{"queue"}, 0.93), (\text{"system"}, 0.37))$, word "queue" has a weight 0.93, "system" has 0.37, and all other words have a zero weight. The weights again describe the "importance" of each word.

The similarity between a profile and a document is calculated by some *similarity measure*, usually the cosine measure, i.e., the dot product between the two. In an IR setting, the similarites between the documents and a query are ranked and the top documents are returned as answers. However, in a SDI setting, it is more appropriate to use a *relevance threshold*, denoted by θ, to judge relevance: the similarity between an incoming document and each of the profiles is computed; if it is above a certain relevance threshold, the document is considered to be relevant to the profile. This way, as argued in [17], instantaneous matching can be done (there is no need to batch the documents for ranking), and also, the precision and recall of the matching are independent of when it is done.

6.2.2 Index Structures Under the VSM

We now show how to generalize the Counting and the Key Methods to work under the VSM. Clearly, we can also extend the Brute Force Methods, but we omit the details here.

The **Profile Indexing (PI) Method** generalizes the Counting Method. In each profile posting

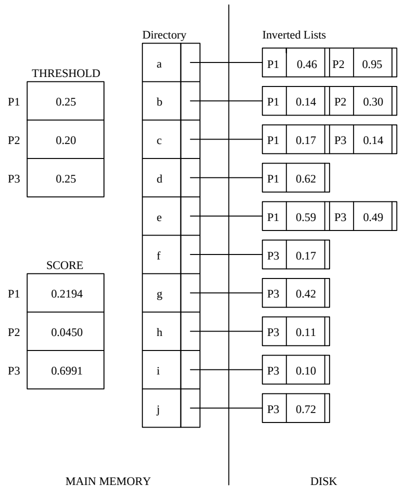

Figure 15: Data Structures for the PI Method

in the inverted list of a word w, we store in addition the weight of w in the profile. Also, instead of having the TOTAL and COUNT arrays, we keep the THRESHOLD and SCORE arrays: the THRESHOLD entry stores the relevance threshold, and the SCORE entry is used to keep the similarity score of the profile.

When a document D arrives, we initialize the SCORE array to all 0's. For each word w with weight x in the document, we use the directory to retrieve w's inverted list. Then we process each profile P in the list. That is, if the weight of w in P is y, we increment SCORE$[P]$ by the product of $x \times y$. After all document words are processed, a profile whose SCORE entry is greater than the THRESHOLD entry matches the document.

To illustrate, consider three profiles:

$$\left| \begin{array}{l|l} P_1 = ((a, 0.46), (b, 0.14), (c, 0.17), (d, 0.62), (e, 0.59)) & \theta_1 = 0\ .25 \\ P_2 = ((a, 0.95), (b, 0.30)) & \theta_2 = 0.20 \\ P_3 = ((c, 0.14), (e, 0.49), (f, 0.17), (g, 0.42), (h, 0.11), (i, 0.10), (j, 0.72)) & \theta_3 = 0.25 \end{array} \right|$$

The θ_i's denote the user-specified relevance thresholds. The inverted index for these profiles is shown in the right-hand side of Figure 15. For example, the a list contains the postings for P_1 and P_2. The 0.46 value in the first entry in this list is the weight of a in P_1. Now suppose this document arrives:

$$D = ((b, 0.15), (d, 0.32), (f, 0.21), (h, 0.14), (j, 0.90)).$$

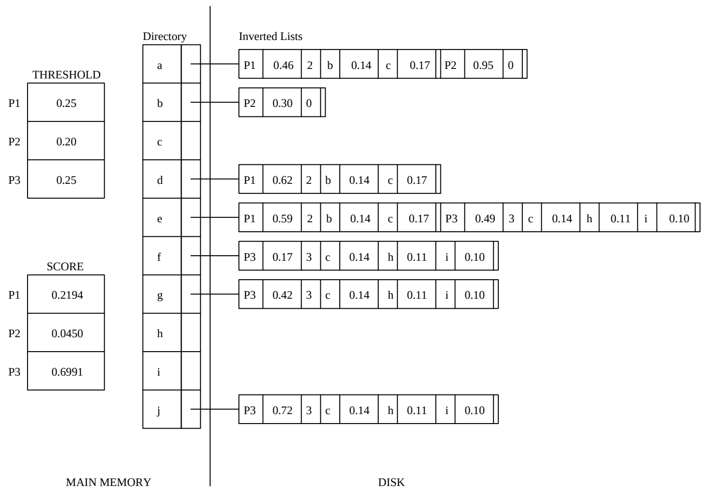

Figure 16: Data Structures for the SPI Method

To process this document, first we read the b list, and increment the SCORE entries of P_1 and P_2 by $0.15 \times 0.14 = 0.021$ and $0.15 \times 0.30 = 0.045$ respectively. The lists of d, f, h, and j are processed similarly. The final values of the SCORE array are as shown in the figure. This document is relevant to P_3.

The **Selective Profile Indexing (SPI) Method** generalizes the Key Methods. In the Key Methods, we index a profile using one of its words. However, this is not possible in the VSM: a document that does not contain the indexed word can still match the profile, if some other words are present and the similarity score exceeds the threshold. Thus, we need to derive a criteria to determine the minimum number of words that need to be indexed. In [17], we present the details of such a criteria. Here, we just assume that we know what *significant* words in a profile must be indexed. In each posting, we include the insignificant words and their weights; i.e., they are duplicated in the lists of all the significant words. Figure 16 shows the data structures for the SPI Method. For instance, in P_1, a, d, and e are the significant words, b and c are the insignificant ones.

We also require the THRESHOLD and SCORE arrays as in the PI method. When a document comes along, we construct its vector representation. Next we initialize the SCORE array to all 0's. Then we index the directory to retrieve the inverted lists of each word. Suppose we are processing the word w with weight x in the document. For each profile P in the w list, suppose the weight of w in P is y, and the insignificant pairs are (v_{i_1}, y_{i_1}), ..., (v_{i_s}, y_{i_s}). We examine P's SCORE entry. There are two cases: if the SCORE entry is zero, we first add the product $x \times y$. Then we look up each word v_{i_j} in the document vector. Suppose its weight in the document is x_{i_j}. We add the product

$x_{i_j} \times y_{i_j}$ to the SCORE entry. In the second case, the SCORE entry is not zero, meaning that we have already added the contribution of the insignificant words in some earlier computation. Thus we only add the product $x \times y$. After all document words have been processed, a profile matches the document if its SCORE entry is greater than the THRESHOLD entry.

For example, in Figure 16, suppose we are processing the first pair (b, 0.15) from the arriving document vector. The list of b has only one posting, that of P_2. We add the product $0.15 \times 0.30 = 0.045$ to P_2's SCORE entry. As there is no insignificant subvector, we are done with this posting and also with the b list. Next we process the pair (d, 0.32). Only P_1's posting is in the d list. First we add the product $0.32 \times 0.62 = 0.1984$ to SCORE[P_1]. Then we process the insignificant subvector ((b, 0.14), (c, 0.17)). To do this, we look up the word b in the document vector, getting a weight of 0.15. Thus we increment SCORE[P_1] by the product $0.15 \times 0.14 = 0.021$. Next, we look up c, which is not in the document vector. We are now done with this list. The other pairs are processed similarly. The final values for SCORE are as shown in the figure.

6.2.3 Performance Comparison

In [17], we compare these two methods, together with the extended Brute Force Method under many different scenarios. Here, we compare them with the methods proposed in this paper. Of course, we are aware that the underlying models are different, and they have different retrieval effectiveness. However, the results should be illuminating on what the costs of increased effectiveness are. In [17], when we evaluate the VSM strategies, we use a different CPU metric (number of floating-point multiplications) from the one used in this paper. Thus, we only report the results for the first two metrics here.

We use the base case parameters in Section 4.2. In addition, we need a value for the relevance threshold, which we set to 0.2, as is done in [17]. Table 3 shows the results (the corresponding results for the Boolean methods are also duplicated here).

VSM Method	Size (Blocks)	I/Os	Boolean Method	Size (Blocks)	I/Os
Brute Force	23731	23731	Brute Force	9668	9668
PI	29275	4495	Counting	18000	2849
SPI	33670	3878	Ranked Key	16475	1139

Table 3: Results for the VSM Strategies

We see that the VSM strategies in general require more disk space and also more I/Os per document. This is because we need to store the weights in the index. In the boolean model, the Ranked Key Method requires less space than the Counting Method, but here, SPI requires more space than PI, since we need to duplicate the insignificant words a number of times. This consequently makes SPI not as attractive as the Ranked Key Method.

7 Related Work

Related to the idea of selective dissemination is the idea of an active database (see e.g. [9]), one which automatically executes certain actions under certain conditions, according to some situation/action rules. Work on active databases typically focuses on structured data in the realm of relational databases. Another recent work by Terry et. al. [14] proposes the notion of continuous queries in relational databases. Users issue continuous queries, which are rewritten into incremental queries and run periodically. Their work concentrates on relational databases, while ours is concerned with the dissemination of unstructured data (documents) using IR techniques.

Wyle and Frei [15] give an overview of a WAN information server that extracts information from information sources periodically and disseminates relevant information to passive users. User profiles, which consists of a number of example messages, are examined periodically and all profiles are examined. This falls into the brute force paradigm in our comparison.

Danzig et. al. [6] present a distributed indexing scheme as a way to provide efficient retrospective search of a large number of retrieval systems. Special sites, called index brokers, maintain indexes of remote retrieval systems. They subscribe generator queries (similar to profiles) that keep them informed of changes in these systems. However, these generator queries are not user-specified profiles discussed in our work. Though segment trees are proposed to index queries over Library of Congress numbers (e.g., all new items in the range QA76 to QA77), index structures for boolean queries are not addressed.

8 Conclusion

We propose in this paper several index structures for conjunctive profiles and algorithms to match profiles against documents. We compare their performance, together with the Brute Force Methods that do not use any index structures. We show that while the Brute Force Methods require less disk space, they perform poorly in terms of number of disk I/Os and CPU processing required and thus execution time. We also demonstrate that rank information of the words can be used to significantly improve the performance of the various methods; and the relative performance of the indexing methods under different scenarios are presented. Finally, we show how to extend the methods to cover negation, disjunction, thesaurus search, truncation, batching, and also vector space model profiles.

We will be collecting profiles submitted to the two experimental SDI servers, in order to study the frequency distribution of words in profiles, the degree of similarity among profiles, and the profile update frequency and pattern. We also plan to extend our SDI indexing algorithms to cover dynamic profile update.

References

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

[2] N.J. Belkin and W. B. Croft, "Information Filtering and Information Retrieval: Two Sides of the Same Coin?" *Communications of the ACM*, 35(12):29-38, December 1992.

[3] CNIDR, "Z39.50-92 Development Efforts," *CNIDR News*, January 1993.

[4] D. Chapman and S. DeFazio, "Statistical Characteristics of Legal Document Databases," Technical Report, Mead Data Central, Miamisburg, Ohio, 1990.

[5] S. DeFazio and J. Hull, "Toward Servicing Textual Database Transactions on Symmetric Shared Memory Multiprocessors," *Proceedings of the International Workshop on High Performance Transaction Systems*, Asilomar, 1991.

[6] P. Danzig, J. Ahn, J. Noll, K. Obraczka, "Distributed Indexing: A Scalable Mechanism for Distributed Information Retrieval," *Proceedings of the ACM SIGIR Conference*, Chicago, October 1991, 220-229.

[7] C. Faloutsos, "Signature-Based Text Retrieval Methods: A Survey," *IEEE Database Engineering*, 13(1):27-34, March 1990.

[8] M. Horton, "How to Read the Network News," *UNIX Documentation*, AT&T Bell Laboratories.

[9] D. McCarthy and U. Dayal, "The Architecture of an Active Database Management System," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Portland, May 1989, 215-224.

[10] B. Reid, "USENET Readership Summary Report for January 1993," USENET Newsgroup `news.lists`, February 8, 1993.

[11] G. Salton, *Automatic Information Organization and Retrieval*, McGraw-Hill, New York, 1968.

[12] G. Salton, *Automatic Text Processing*, Addison Wesley, Reading, Massachusetts, 1989.

[13] A. Tomasic and H. Garcia-Molina, "Performance of Inverted Indices in Distributed Text Document Retrieval Systems," *Proceedings of the International Conference on Parallel and Distributed Information Systems*, San Diego, January 1993, 8-17.

[14] D. Terry, D. Goldberg, D. Nichols, B. Oki, "Continuous Queries over Append-Only Databases," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, May 1992, 321-330.

[15] M.F. Wyle and H.P. Frei, "Retrieving Highly Dynamic, Widely Distributed Information," *Proceedings of the ACM SIGIR Conference*, Cambridge, Massachusetts, June 1989, 108-115.

[16] T.W. Yan and H. Garcia-Molina, "Index Structures for Selective Dissemination of Information," Technical Report STAN-CS-92-1454, Stanford University, 1992.

[17] T.W. Yan and H. Garcia-Molina, "Index Structures for Information Filtering Under the Vector Space Model," to appear in *Proceedings of the International Conference on Data Engineering*, 1994.

[18] G.K. Zipf, *Human Behavior and the Principle of Least Effort*, Addison-Wesley Press, Cambridge, Massachusetts, 1949.

A Details of Analysis for the CPU Processing Metric

A.1 Preliminary Calculations

Here we derive some useful expressions used later in the analysis.

Using (2), we compute the expected number of distinct words in a document D as

$$\begin{aligned} W_T &= \sum_{w=1}^{T} \Pr(w \text{ is in } D) \\ &= \sum_{w=1}^{T} (1 - (1 - Z(w))^W). \end{aligned}$$

Also, recall from Section 4.4.2 that the expected number of distinct words in D that appear in some profiles is

$$W_S = \sum_{w=1}^{S} (1 - (1 - Z(w))^W).$$

A.2 The Brute Force Methods

For the expected number of occurrence table look-ups per document for the Random Brute Force Method, we note that the probability that a word randomly chosen from U is in a document D is

$$\alpha = \frac{W_S}{S}.$$

Now consider matching an arbitrary profile against D. With probability $1-\alpha$ the first word, which is a word randomly drawn from U, is not in D and the match ends (1 look-up). With probability $\alpha(1-\alpha)$, the first word matches and the second does not, so we do 2 look-ups. In general, the expected number of look-ups per profile is

$$\begin{aligned} K_{RandomBruteForce} &= 1(1-\alpha) + 2(1-\alpha)\alpha + 3(1-\alpha)\alpha^2 + \ldots + \\ &\quad +(K-1)(1-\alpha)\alpha^{K-2} + K\alpha^{K-1} \\ &= 1 + \alpha + \alpha^2 + \ldots + \alpha^{K-1} \\ &= \frac{1-\alpha^K}{1-\alpha.} \end{aligned}$$

Hence the expected number of occurrence table look-ups per document, or the expected number of normalized accesses per document, is

$$A_{RandomBruteForce} = NK_{RandomBruteForce}.$$

Next we consider the expected number of occurrence table look-ups for the Ranked Brute Force Method. The probability that the i-th lowest ranked word of a randomly picked profile is in a document D is

$$\begin{aligned}
\alpha_i &= \sum_{v=1}^{S} \Pr(\text{the } i\text{-th word is } v \text{ and } v \text{ is in } D) \\
&= \sum_{v=1}^{S} \Pr(\text{the } i\text{-th word is } v) \times \Pr(v \text{ is in } D) \\
&= \sum_{v=1}^{S} \frac{\binom{S-v}{i-1}\binom{v-1}{K-i}}{\binom{S}{K}} \times (1-(1-Z(v))^{W}).
\end{aligned}$$

This is because, to pick a profile with the word v as its i-th lowest ranked word, we have to pick $i-1$ words from the $S-v$ words ranked lower than v, and $K-i$ words from the $v-1$ words ranked higher than v. So if we look at all the possible profiles obtained by picking K words out of S words, $\binom{S-v}{i-1}\binom{v-1}{K-i}$ of them have v as the i-th lowest ranked word. Thus, the probability that a randomly picked profile has v as its i-th lowest ranked word is $\binom{S-v}{i-1}\binom{v-1}{K-i}/\binom{S}{K}$.

The expected number of look-ups per profile is

$$\begin{aligned}
K_{RankedBruteForce} &= 1(1-\alpha_1) + 2\alpha_1(1-\alpha_2) + \ldots + \\
&\qquad +(K-1)\alpha_1 \ldots \alpha_{K-2}(1-\alpha_{K-1}) + K\alpha_1 \ldots \alpha_{K-1} \\
&= 1 + \alpha_1 + \alpha_1\alpha_2 + \ldots + \alpha_1 \ldots \alpha_{K-1}
\end{aligned}$$

and the expected number of look-ups, i.e., the expected number of normalized accesses, per document is

$$A_{RankedBruteForce} = NK_{RankedBruteForce}.$$

A.3 The Counting Method

In the Counting Method, we look up every distinct word from the distinct words set and map it to its inverted list using the directory, which is a hash table. Thus there are W_T array accesses and W_T hash table look-ups. There are no occurrence table look-ups for this method. Instead, there are accesses to the TOTAL/COUNT arrays. First we have to initialize the COUNT array with N accesses. Then for every word in every posting, we we have to probe the TOTAL/COUNT arrays. As there are $\frac{NK}{S}$ postings per list, the expected total number of array accesses is

$$W_T + N + \frac{NK}{S} \times W_S.$$

Using the normalizing ratio r, the total expected number of normalized accesses is

$$A_{Counting} = W_T + \frac{W_T + N + \frac{NKW_S}{S}}{r}.$$

A.4 The Random Key Method

For the Random Key Method, first we have to get the distinct words and index the lists; this takes W_T array accesses and W_T hash table look-ups. Next, we consider the number of occurrence table look-ups. For each posting in an inverted list, the expected number of look-ups needed is

$$\begin{aligned}
K_{RandomKey} &= 1(1-\alpha) + 2(1-\alpha)\alpha + \ldots + \\
&\qquad +(K-2)(1-\alpha)\alpha^{K-3} + (K-1)\alpha^{K-2} \\
&= 1 + \alpha + \ldots + \alpha^{K-2} \\
&= \frac{1-\alpha^{K-1}}{1-\alpha.}
\end{aligned}$$

As W_S lists are examined, and each list has an expected number of profiles of $\frac{N}{S}$, the expected total number of normalized accesses is

$$A_{RandomKey} = W_T(1 + \frac{1}{r}) + \frac{NW_S}{S} \times K_{RandomKey}.$$