Index Structures for Selective Dissemination of Information

Tak W. Yan and Hector Garcia-Molina

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 issue queries that are constantly evaluated, called standing orders, so that they will be automatically informed of new additions that may be of interest. Such service is traditionally called Selective Dissemination of Information.

The huge number of users and the timeliness requirement of the service pose a challenge. In this paper, we propose several index structures for standing orders and algorithms that efficiently match documents against large number of orders. 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 growing. 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 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 issue *standing orders*, 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 information retrieval system, complementary to retrospective search.

Traditionally, libraries and databanks (e.g. Dialog) provide such standing order service under the name of *Selective Dissemination of Information (SDI)* [11]. Users subscribe interest profiles, which contain a number of standing orders. These orders 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, orders, and arriving documents grows. Furthermore, it is becoming important to provide timely service, instead of once a week or once a month.

The Netnews [6, 7] 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 news group "comp.databases," as well as other related news groups. Articles about relational database systems appearing in news groups not explicitly subscribed to will not be seen. Also, it is the site administrator who decides what news groups are locally available. Hence, the database news group may not even be available for this user. It would be more effective if a user could specify orders in the style of information retrieval 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 news group they fall into or what the local site administrator thinks is important. A recent effort [17] 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 standing orders, unlike Netnews.
- It should be able to evaluate orders continuously and notify the user as soon as a relevant document arrives, not periodically.
- It should scale to very large number of orders 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

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

receives new documents and matches them against a database of standing orders. In particular, we present several index structures and matching algorithms that the server can use.

Even though our matching algorithms will be shown to be efficient, the single server concept does not scale to truly large distributed systems. In that environment one needs a collection of servers, with documents being distributed to some or all servers, and orders being subscribed at some or all servers. We do not address this scenario 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 standing orders $O_1, O_2, ..., O_N$. This server receives all new documents as they are produced. Say a new document D 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 all orders and check them. For example, if O_1 is for documents with words a and b, we can check if D has a and b; if so, we send D to the person that posted O_1.

An improvement is to build an index structure for the orders [3], which associates every word with a list of orders that contain it. Then, if we know D has, for instance, words a, b, c, we can search the index for orders containing these words. We obtain a list of all the orders involving a, a list of all the b orders, and a list of all the c orders. After merging the lists, we check the orders as described above. With this strategy we avoid checking all orders, i.e., orders that do not contain a, b, or c will not be considered.

The situation is the opposite to conventional information retrieval. In that case, we receive a query and check it against an index of documents. In the standing order case, we get a document and check it against an index of orders. Thus, the index structures and processing algorithms are different. In this paper we explore in detail solutions such as the ones we sketched here.

The rest of the paper is organized as follows. Section 2 explains the terminology and assumptions we use. In Section 3 we present several index structures and matching algorithms for standing orders. We evaluate the various strategies through analysis and simulations; the analysis and details of the simulations 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, except those very frequent noise words (e.g., "and," "the"), 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 6.5.

2.2 Orders

Most commercial information retrieval systems use boolean queries to specify the content of the documents to be retrieved. Some "advanced" information retrieval approaches (e.g., [2, 13]) also use boolean queries at their lowest level. That is, these strategies may first convert their high level queries into boolean queries and use a boolean query search engine to retrieve potential documents and then perform post-processing as needed. Thus, we assume standing orders are boolean queries.

In this paper we focus on conjunctive orders. We assume that an *order* O is a sequence of distinct words $(w_1, w_2, ..., w_K)$. An order *matches* a document if all its words appear in the document. A user can subscribe a number of orders and an order may be identical to some other orders in the system; we assume each order has a unique integer identifier. The algorithms we present below can easily be extended to handle negation in orders (e.g., select documents in which a word w does not appear). For simplicity we postpone a discussion of negation to Section 6.1. Orders with logical ORs can be handled by converting them to disjunctive normal form, in which case they can be handled as collections of conjunctive orders. (Example: Posting order $a \wedge (b \vee c)$ is equivalent to posting $(a \wedge b) \vee (a \wedge c)$, which is equivalent to posting two orders, one for $a \wedge b$, the second for $a \wedge c$.) Handling general orders as collections of conjunctive orders may not be the most efficient strategy, but again, we postpone this discussion to Section 6.2.

Information retrieval systems have a number of extensions to boolean queries, such as proximity, truncation, and thesaurus expansion [12]. The methods we present below can be augmented to handle these extensions. The discussion is presented in Section 6.3 and 6.4.

2.3 Index Structures

In the index structures presented below, for each word w, its *inverted set* of orders, i.e., orders containing it, is 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 orders, 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 order 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 orders 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 structures as orders 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.

3 Index Structures and Algorithms

In this section we describe several methods that make use of index structures to generate the identifiers of orders that match an incoming document. For comparison, we also include the brute force strategy mentioned earlier.

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*

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

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 the standing orders sequentially on disk without any index structures. This way, all orders must be evaluated whenever a document arrives.

When a document arrives, we construct the occurrence table. Then we examine each order in turn. For an order (conjuction-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 order 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 order, then the next less frequent and so on. This makes it possible to detect non-matching orders faster, i.e., with fewer probes into the occurrence table. For a matching order we do not save anything since all words in the order 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 available. Thus, it is important to study both types of strategies.

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

3.2 The Counting Method

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

Now with this structure, how do we match a document to orders? Note that we cannot, as in

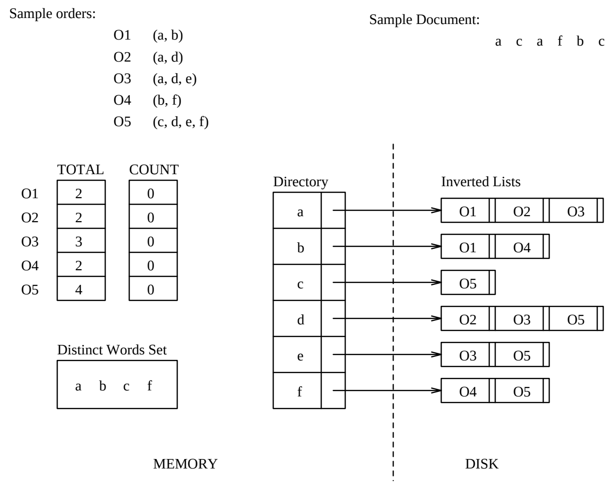

Figure 1: Data Structures for Counting Method

the case of the standard inverted index, process the lists using set intersections. Our solution makes use of this observation: an occurrence of an order in an inverted list looked at means that one word in that order is matched. Thus, if we count the number of occurrences of an order in the inverted lists looked at, we can determine if the order matches the document. This is called the *Counting Method*.

To do the counting efficiently, we need two (main memory) arrays, TOTAL and COUNT. The number of entries in each array is equal to the maximum number of orders the system handles. Each order has a entry in each array. The TOTAL entry stores the number of words in the order, and the COUNT entry is initialized to 0 and is used to keep count of occurrences of the order in the inverted lists. Assuming the number of words in an order is less than 256, each entry takes one byte. To illustrate, Figure 1 shows a set of sample orders, a sample document to be matched, and the the various main memory and disk data structures. For example, notice that the inverted list for word a contains the identifiers for orders O_1, O_2, O_3, indicating that these orders refer to a.

When a document D arrives, we first initialize the COUNT array to all 0's. Then we construct the distinct words set. (Note that we do not need the occurrence table. However, to efficiently construct the distinct words set, we may as well use a hash table, which is essentially what the occurrence table is.) For each distinct word, we use the directory to get its list. For each order in the list, we increment its entry in the COUNT array. An order matches a document when its COUNT entry becomes equal to the TOTAL entry.

To continue our example (Figure 1), say the sample document arrives and the distinct words set is constructed. We take the first word, a, retrieve its list, and add 1 to COUNT for orders O_1, O_2, O_3. We continue with b, c, f, obtaining COUNT entries for O_1, O_2, O_3, O_4, and O_5 of 2, 1, 1, 2, and 2 respectively. For O_1, its COUNT (2) is equal to its TOTAL (2), so O_1 matches. Similarly, O_4 matches.

As we have stated, we only need to keep the order identifier in a record. Records in the same inverted list are stored sequentially in blocks, so there is no need to have a pointer to point to the next record. The size of a record is thus equal to the size of an identifier.

3.3 The Key Methods

The Counting Method duplicates an order in the lists of all the words it contains, but this is not necessary – we only need to associate an order with any one of the words. This is because, for a match, every word in the order must appear in the document, and no matter which list it is in, that list will be looked at. We can randomly pick one of the words, called the *key*, and place the record in its list; this is the *Random Key Method.* If the frequency distribution of words in documents is known, we can also use *Ranked Key Method* and store the record in the list of the word with the lowest rank. The idea is to ensure that the lists that are looked at more often are the shorter ones. Figure 2 show 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 orders, and the occurrence table and distinct words set for a sample document. Notice for instance that O_1 is in the a list but not in the b list.

As in the Counting Method, an inverted list consists of records, but here a record contains the order identifier, the length of the order, and the words except the one indexed. We store records sequentially in blocks, so there is no need to have next record pointers.

When a document comes along, we build both the distinct words set and the occurrence table. Then we index into the inverted lists corresponding to each distinct word in the document. For each of the orders in the list, we check if it matches the document. Again, depending on whether frequency information is available, the look-up order can be random or ranked. We assume the Random Key Method uses random look-up, and the Ranked Key Method uses ranked look-up.

For example, consider the sample document and orders in Figure 2. To process the document, we look up words a, b, c and f in the directory. Word f has no orders associated with it, so its list is empty. We read in the list of, say, a first and check the orders in the list. For the first order in this list (O_1), we look up the word b in the occurrence table, which is in the document. As there are no

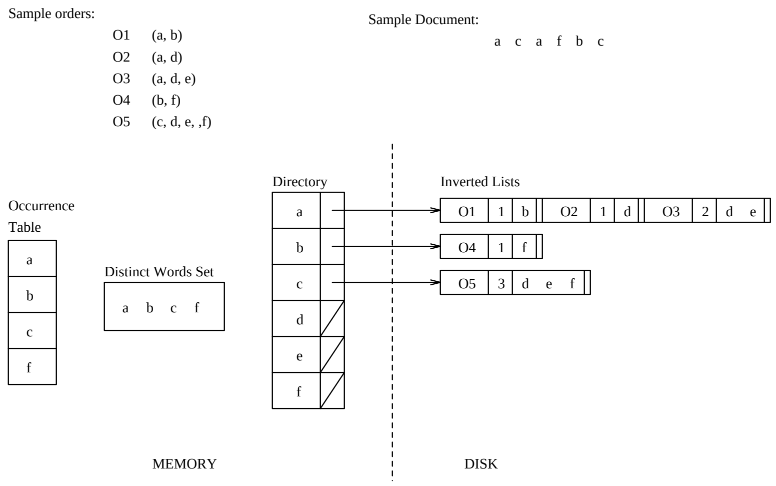

Figure 2: Data Structures for Index Method

more words in the order, O_1 matches the document. Then we go on to the next order, O_2. We look up d, which is not in the document; thus O_2 does not match. Finally, we look at O_3. There are two words we need to check: d and e. Using ranked look-up and assuming d has a lower rank than e, we look up d first. It is not in the document and so O_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 orders, it is likely that a large number of orders have similar words. With this observation, it may be possible to store the orders more compactly than lists. A possibility is to use a trie-like structure [8].

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

The identifying prefixes of the orders 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

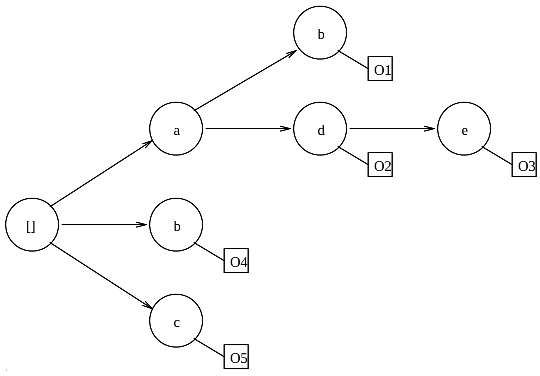

Figure 3: A Tree of Identifying Prefixes and Identified Orders

σ are represented by the same node n. Its children are nodes corresponding to prefixes $(w_1, ..., w_i, v)$ of some identifying prefixes. Figure 3 shows the tree of the identifying prefixes for the sample orders in the previous example. For instance, the top-most node represents the prefix (a, b) and identifies order O_1. The lowest node is for prefix (c) and identifies O_5. Notice that it is not necessary to have nodes for prefixes (c, d) or (c, d, e), as the shorter (c) prefix is sufficient to identify O_5.

A node n corresponding to a prefix $\sigma = (w_1, ..., w_i)$ 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; orders, a list of orders of which σ is the identifying prefix (note that orders in the list must all be identical); length, the length of the postfix of the orders σ identifies; and postfix, words that make up the postfix of the identified order(s). The last two fields (length and postfix) exist only if orders is not an empty list.

Figure 4 shows the internal structure of the same tree in Figure 3. For example, the leftmost node only contains a children list. If we follow a's pointer, we get to the node representing the (a) prefix (labeled x in Figure 4). Again, this node only has a children list. Following d's pointer, we get to node called z for prefix (a, d). This node has a children list, plus orders entry (identifying O_2), plus a length field of 0, indicating there is no postfix (i.e., O_2 is only (a, d)).

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 4, the uppermost subtree is formed by the inverted set of orders 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 as detailed below. The directory thus maps a word v to the disk location of the subtree of orders that start with v.

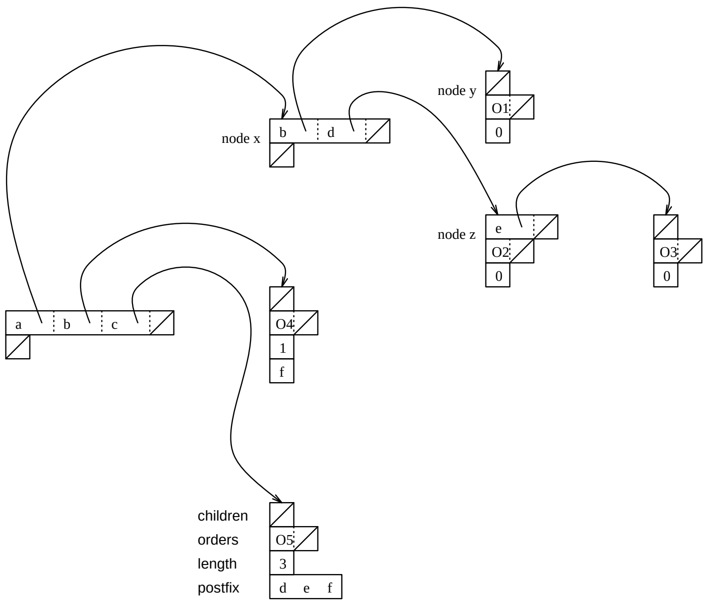

Figure 4: Internal Structure of the Tree

A subtree can be laid out on disk in a variety of ways. Since we need to evaluate the storage requirement (Section 4), here we detail one reasonable layout, in breadth first order. We first store the root of the subtree, then the first node on the next level, then the second node, and so on. When finished with a level, we store the next level. Each node is layed out in the following manner. The list children consists of (word identifier, pointer) pairs stored sequentially. Each pair is preceded by a 1 bit; the end of the list is indicated by a 0 bit. Then, the list orders is similarly packed: each order identifier is preceded by a 1 bit; a 0 preceding bit indicates the end of the list. Next (if orders is not empty) we store the length of the postfix and finally the postfix as a sequence of word identifiers. The pointers can be represented as bit offsets into the structure. Figure 5 shows the layout of the tree in Figure 4.

When a document arrives, we first construct the distinct words set and occurrence table. For each distinct word, we index the directory and read into memory the subtree for that word. To find the matching orders in the subtree, we do a breadth first search [1]. We keep a queue of pointers which is made up 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.

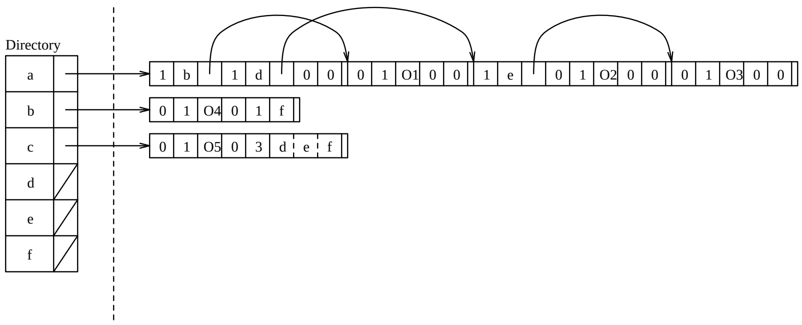

Figure 5: Disk Layout of the Tree

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 orders in orders match the document. If the postfix is empty, then the orders also match.

Consider the tree in Figure 4 and the sample document as in the previous example. Suppose we try to match the orders 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 order 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 order O_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 orders. One heuristic to increase the number of common prefixes is to sort the words in orders, e.g. in increasing word identifier sequence. For example, if we sort the two orders (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. This has the advantage that (similar to the Ranked Key Method) more orders 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 Analysis and Simulations

We compare the performance of the above methods through analysis and simulations. We first describe the document and order model we have adopted and the evaluation metrics. (These models are similar to the document and query models used in [16].) Then we present the analysis and describe the details of the simulations.

4.1 Document Model

Let the size of the vocabulary V be T (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 nonincreasing 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 [19], i.e.,

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

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.

The top part of Table 1 summarizes these parameters. The Base Value column represents the base case scenario. The values are from a legal document database described in [4]

4.2 Order Model

The extremely infrequent words (in documents) typically are misspellings or typos and are not used in queries. Thus, we assume orders do not use these words either. We model this by assuming that order words are chosen form the set $U = \{1, ..., S\}$ out of the vocabulary $V = \{1, ..., T\}, S < T$. (Notice that we are identifying words by their ranks.) We further assume that each word in U is equally likely to be chosen for an order. This uniform distribution is justfied as queries tend to use a mix of frequent and relatively infrequent words [16]. Hence, we assume that an order is a set of K words chosen randomly without replacement from the queried vocabulary U.

The number of orders in the system is N. To allow easy study of the effect of order size on performance, we assume all orders have the same length, i.e., K is fixed for all orders.

These parameters are tabulated in the second part of Table 1. The Base Value column shows the values used in the base case. The size of the queried vocabulary (S) is chosen to be 18000. This value of S accounts for 96.3% of the cumulative occurrences of words in the document database described in [4].

The third part of Table 1 summarizes the parameters for the sizes of the various fields in records/nodes and the disk block size. We assume I and X are the minimum possible. In actual implementations, these number could be somewhat larger, for example to simplify packing/unpacking operations, or to allow for growth in the number of orders and the size of vocabulary.

Parameter	Base Value	Description
T	1.8 Million	size of vocabulary
W	12000	no. of words per document
S	18000	size of queried vocabulary
N	300000	no. of orders
K	5	no. of words per order
I	$\lceil \log_2 N \rceil$ bits	length of order identifier
L	8 bits	no. of bits to represent length of order
X	$\lceil \log_2 T \rceil$ bits	no. of bits to represent a word
P	20 bits	no. of bits for a pointer in the Tree methods
B	4096 bits	no. of 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

4.3 Metrics

We compare the methods with respect to their space and time requirements. For space requirement, we look at how much disk space each structure takes. Although main memory space requirements of the methods differ, we assume all fit in main memory. For time requirement, in an I/O bound system, the critical measure is the number of I/O's to process a document; in a CPU bound system, the amount of computation is the critical component. Hence, we look at both aspects in our comparison.

In each of the methods, the inner loop of the matching process is to access the main memory data structures: the directory (to get the inverted set of a word), the occurrence table (to check the presence of a word in the document), the distinct words set (to get a distinct word from a document), and the TOTAL/COUNT arrays. Analogous to the compare instruction in sorting algorithms, these accesses can be used as the counting unit to measure the amount of CPU processing needed.

Although a directory look-up is comparable to an occurrence table look-up (both are hash tables indexed by word identifiers), an access to the distinct words set, which can be implemented as an array, or the TOTAL/COUNT arrays takes much less work. Thus we introduce a scale factor, r, that specifies how many array accesses are comparable to a hash-table look-up and use the number of normalized accesses as the metric for measuring CPU processing. The last row in Table 1 describes this 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 accesses to main memory data structures performed for processing a document.

Note that in all methods, we need to construct the distinct words set and the occurrence table. As these are upfront costs required by all methods, we ignore them. Also ignored is the cost to encode words in documents as integer identifiers.

4.4 Analysis

4.4.1 Preliminary Calculations

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

Given a document D generated under the document model, the probability that a particular word w is in D is

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

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 (1), we compute the expected number of distinct words in D as

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

Similarly, the expected number of distinct words in D that appear in some orders, i.e., the expected number of distinct words in D that are in U, is

$$W_S = \sum_{w=1}^{S} (1 - (1 - Z(w))^W). \tag{3}$$

Next we derive an expression that allows us to compute the expected number of blocks in an inverted list in the Counting and Key Methods. We consider the question: given $\mathcal{N}$ records, 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 record falls in this list is $\mathcal{P}$? Let us denote the desired expression by $\mathcal{F}(\mathcal{N}, \mathcal{P}, \mathcal{R}, \mathcal{B})$.

Intuitively, we can compute the expected number of records in the list as $\mathcal{N}\mathcal{P}$ and compute the expected number of blocks as

$$\frac{\mathcal{N}\mathcal{P}\mathcal{R}}{\mathcal{B}}.$$

However, this is incorrect as it neglects the internal fragmentation that results when the records do not fully occupy an integral number of blocks. The formula

$$\lceil \frac{\mathcal{N}\mathcal{P}\mathcal{R}}{\mathcal{B}} \rceil$$

is incorrect also, as it always overestimates the number of blocks required. (For example, if $\mathcal{P}$ is very very small, the expected number of blocks should be small (less than 1), yet the formula gives 1 no matter how small $\mathcal{P}$ is.)

Let us now derive a correct expression for the value. Let random variable $\mathbf{X}$ be the number of records in the list. $\mathbf{X}$ follows the binomial distribution $\mathrm{Bin}[\mathcal{N}, \mathcal{P}]$. 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.

$$\begin{aligned} \Pr\{\mathbf{Y} = y\} &= \Pr\{\lceil \frac{\mathcal{R}\mathbf{X}}{\mathcal{B}} \rceil = y\} \\ &= \Pr\{y - 1 < \frac{\mathcal{R}\mathbf{X}}{\mathcal{B}} \leq y\} \end{aligned}$$

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

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

$$
\begin{aligned}
\mathcal{F}(\mathcal{N}, \mathcal{P}, \mathcal{R}, \mathcal{B}) &= E[\mathbf{Y}] \\
&= \sum_{y \ge 0} y \Pr\{\mathbf{Y} = y\}. \qquad (2)
\end{aligned}
$$

4.4.2 The Brute Force Method

The analysis of the space requirement for the Brute Force Methods is straightforward. Recall that we store in a record the order identifier, the length of the order, and the words in an order, and that records are stored sequentially in contiguous blocks. Since orders have K words, the space required to store all orders (in blocks) for both Brute Force Methods is

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

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

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

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, using (3),

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

Now consider matching an arbitrary order 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 order 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

$$A_{RandomBruteForce} = N K_{RandomBruteForce}.$$

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 order is in a document D is

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

This is because, to pick an order 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 orders 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 order 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 order is

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

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

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

4.4.3 The Counting Method

To analyze the Counting Method, we make use of (4). For a particular list, the maximum number of records that can be placed in it is N. (Although the total number of records in the index structure is NK, at most only N of them can be on the same list.) The probability that a record is in a list is $\frac{K}{S}$. Only the order identifier is kept in an record, so the record 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 lists, the expected total space requirement is (in blocks)

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

The inverted lists of the distinct document words that appear in orders are read in, so the expected number of blocks read per document is

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

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 record, we we have to probe the TOTAL/COUNT arrays. As there are $\frac{NK}{S}$ records 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

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

4.4.4 The Key Methods

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

Using the Random Key Method, the probability that a record is in a list is $\frac{1}{S}$. 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 D 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 record is in the list for a word w is $\frac{\binom{w-1}{K-1}}{\binom{S}{K}}$. This is the probability that an order has w as its lowest ranked word. Adding up the lengths of each list, the total space required is

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

The expected number of blocks read is

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

We now compute CPU processing costs. 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 record in an inverted list that we look at, there is one less word that we need to look at than in the Brute Force Methods. Hence, if we use the Random Key Method, the expected number of look-ups per order is

$$K_{RandomKey} = 1(1-\alpha) + 2(1-\alpha)\alpha + \ldots +$$

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

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

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

On the other hand, if we use the Ranked Key Method the expected number of look-ups needed is smaller. The average number of look-ups per order is different for orders on different lists. Let us denote by $\alpha_{w,i}$ the probability that the i-th lowest ranked word of a random order with lowest ranked word w is in a document D. First, the probability that a word v is the i-th lowest ranked word of a random order with lowest ranked word w is equal to

$$\frac{\binom{w-1-v}{i-2}\binom{v-1}{K-i}}{\binom{w-1}{K-1}},$$

since the number of all possible orders with w as the lowest ranked word is $\binom{w-1}{K-1}$, and the number of all possible orders with w as the lowest ranked word and v as the i-th lowest ranked word is $\binom{w-1-v}{i-2}\binom{v-1}{K-i}$. Hence,

$$
\begin{aligned}
\alpha_{w,i} &= \sum_{v=1}^{S} \Pr(v \text{ is the } i\text{-th lowest ranked word in an order with lowest ranked word } w) \\
&\qquad \times \Pr(v \text{ is in } A) \\
&= \sum_{v=1}^{S} \frac{\binom{w-1-v}{i-2}\binom{v-1}{K-i}}{\binom{w-1}{K-1}} \times (1-(1-Z(v))^W) \\
&= \sum_{v=1}^{w-1} \frac{\binom{w-1-v}{i-2}\binom{v-1}{K-i}}{\binom{w-1}{K-1}} \times (1-(1-Z(v))^W).
\end{aligned}
$$

The expected number of look-ups per order in an inverted list of word w is

$$
\begin{aligned}
K_{RankedKey}(w) &= 1(1-\alpha_{w,2}) + 2\alpha_{w,2}(1-\alpha_{w,3}) + \ldots + \\
&\qquad +(K-1)\alpha_{w,2}\ldots\alpha_{w,K-2}(1-\alpha_{w,K-1}) + K\alpha_{w,2}\ldots\alpha_{w,K-1}
\end{aligned}
$$

$$= \quad 1 + \alpha_{w,2} + \alpha_{w,2}\alpha_{w,3} + \ldots + \alpha_{w,2}\ldots\alpha_{w,K-1}.$$

The list of w has an expected number of records of

$$\frac{\binom{w-1}{K-1}}{\binom{S}{K}}N,$$

so the expected total number of normalized accesses is

$$A_{RankedKey} = W_T(1 + \frac{1}{r}) + \sum_{w=1}^{S}(1 - (1 - Z(w))^W)\frac{\binom{w-1}{K-1}}{\binom{S}{K}}NK_{RankedKey}(w).$$

Although we are able to derive this expression for $A_{RankedKey}$, it turns out to be tedious to evaluate. Hence, the values presented in Section 5 are simulation results.

4.5 Simulations

Except those for the Tree Methods, the last metric of the Ranked Key Method, and those given in Section 5.5, our results are obtained by numerically evaluating the above expressions. Simulation results are given where analytical results are not derived. We also constructed simulations to validate the anlaysis. The simulation results did match the analytical ones.

We have written our simulation program in C. The program first generates N orders according to the order model, and then computes the size of the index structures needed to store the queries. Next the simulation program generates a random document according to the document model and counts the number of disk reads and the number of normalized accesses to different data structures needed to match it against the N orders.

For each scenario we have tested, the program is run enough times (with different random number generator seeds) to make sure 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.

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

Table 2: Results for the Base Case

For the space requirement, we can see the Brute Force Methods require the least disk space. In the Brute Force Methods, orders are packed into sequential blocks, so no space is wasted (except in the last block). In other methods, as orders 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.

On the other hand, the indexing methods outperform the Brute Force Methods by a factor of 3 to 8 with respect to the number of blocks read in the matching process. 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 orders, 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 orders does not generate a lot of orders that share common prefixes, even though the number of orders 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 numbers. The issue of caching is ignored for this comparison, as we expect it to benefit all methods. Another reason the required number of I/Os is large is that in our base case documents are large (12,000 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).

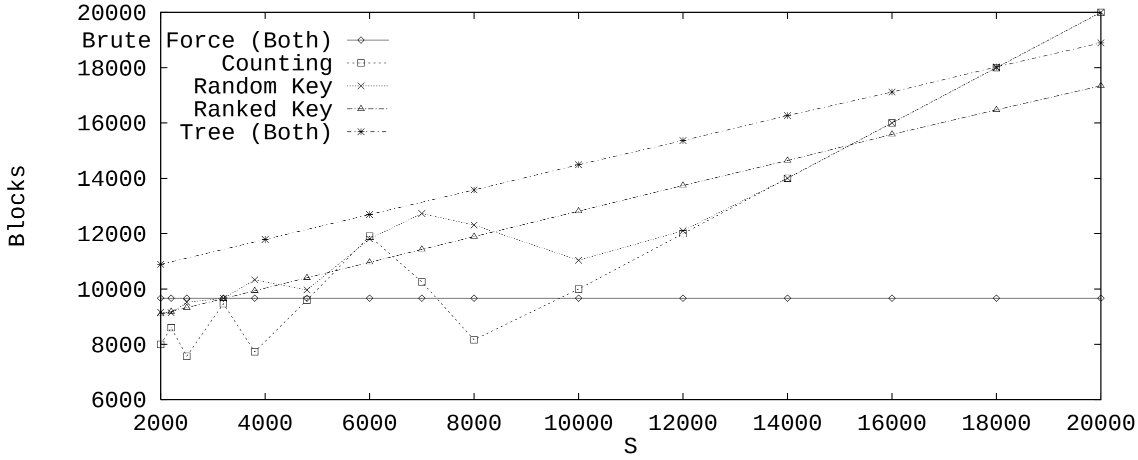

Figure 6: Total Disk Space Required vs S

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

For the number of normalized accesses, the Ranked Tree Method performs the best, and all indexing methods are superior to the Brute Force Methods. Although the Counting Method seems to require more normalized accesses than the other indexing methods, we note that the result is very sensitive to the normalizing factor r.

Next we perform a sensitivity analysis, investigating the impact of four important parameters on performance. We also modify the order model to study how similarity among orders 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 6, 7, and 8 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 6 and 7, 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

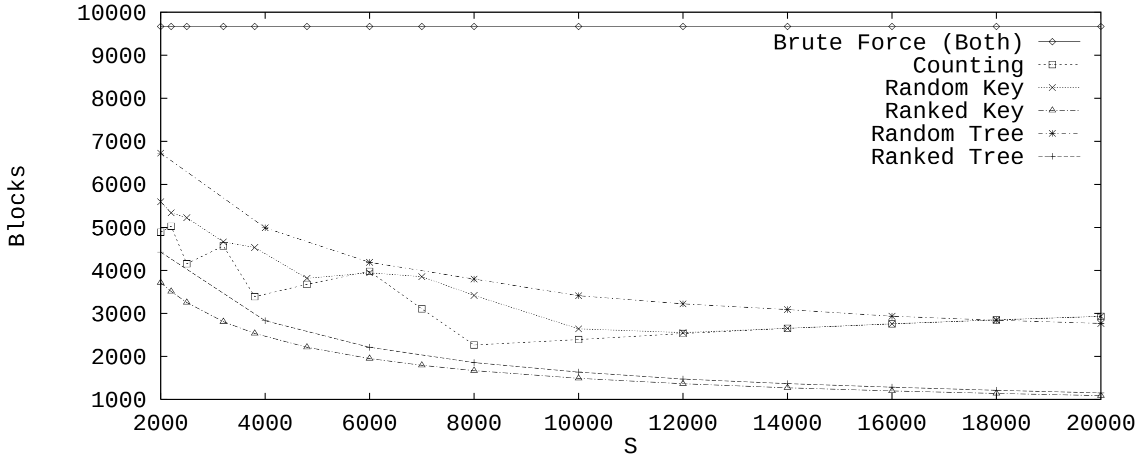

Figure 7: Number of Blocks Read Per Document vs S

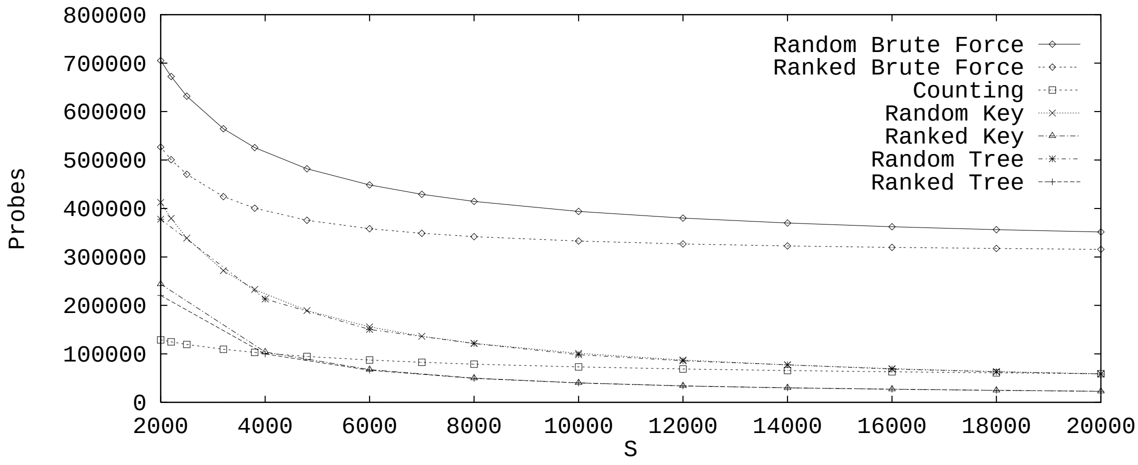

Figure 8: Number of Normalized Accesses vs S

Figure 6. 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 orders per list decreases (the number of orders 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 orders contains 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.)

The Counting Method performs well when S is small. With S small, it is more likely that orders contain words that appear frequently in documents. Hence, more orders are read in and more words are looked up in all indexing methods. As the Counting Method requires the least disk space per order (internal fragmentation is insignificant when S is small) and the least processing per word, it does well with respect to the two metrics.

5.2 Varying N

Next we study the effect of varying N, the number of orders. The base values are used for the other parameters. Figure 9, 10, 11 show the performance of the various methods as N is varied.

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 6. Similar reasoning explains why in Figure 7 the number of I/Os for the two methods is equal to the expected number of distinct document words that appear in orders. 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 orders, and the Tree Methods would do better. However, the Tree Methods do not perform very well, even when the number of orders is very large. This again shows that the independent order model does not generate

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

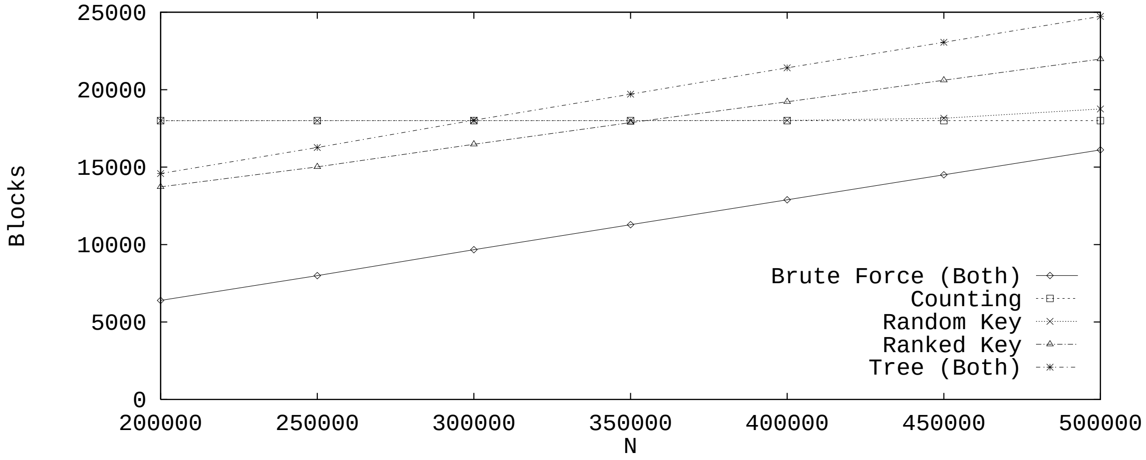

Figure 9: Total Disk Space Required vs N

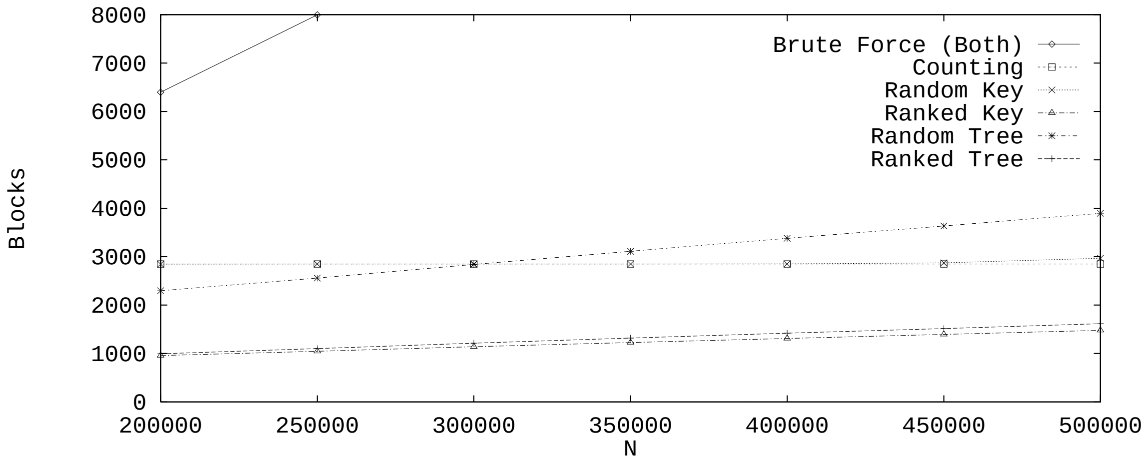

Figure 10: Number of Blocks Read Per Document vs N

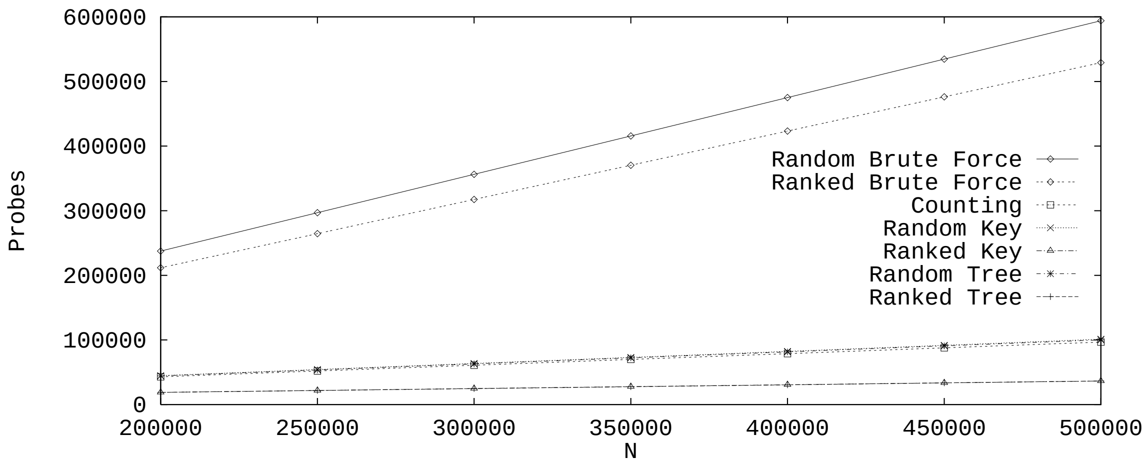

Figure 11: Number of Normalized Accesses vs N

enough similarity among orders to make the Tree Methods attractive.

5.3 Varying K

Next we investigate the effect of varying the length of the orders. We vary K from 1 to 10 and compare the performances of the various methods. The results are shown in Figures 12, 13, and 14.

In Figures 12 and 13, we notice some surprising results: with order size increasing from 1, the space requirement and the number of disk accesses for all the indexing methods not only do not 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 orders 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 orders. 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 an order. 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.

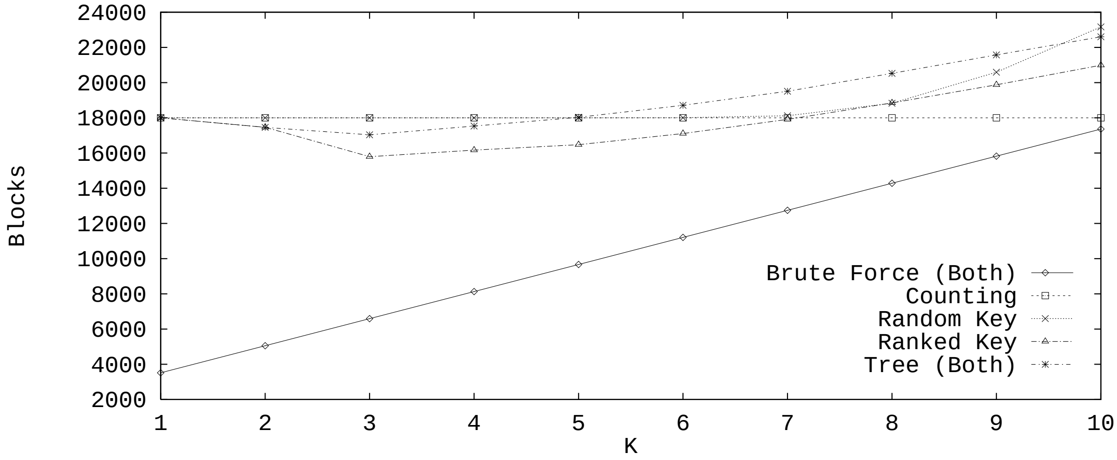

Figure 12: Total Disk Space Required vs K

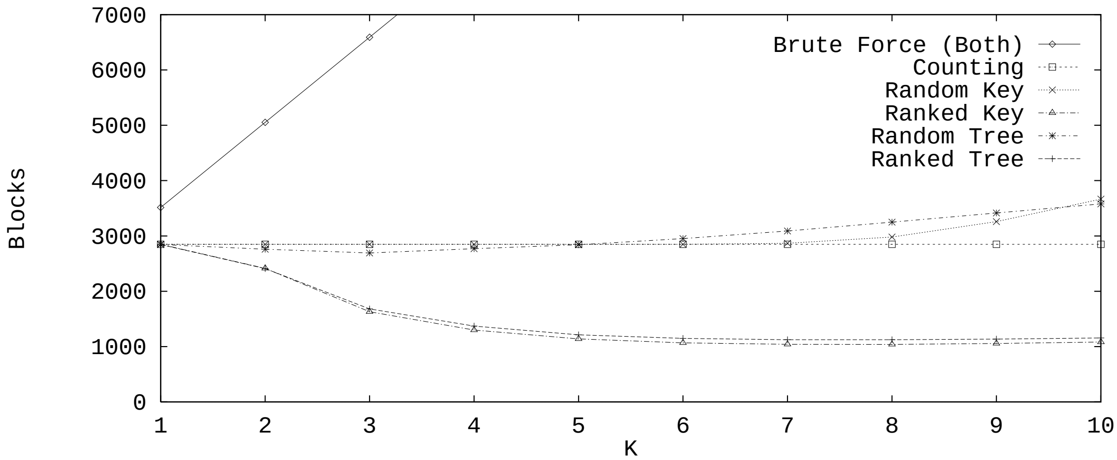

Figure 13: Number of Blocks Read Per Document vs K

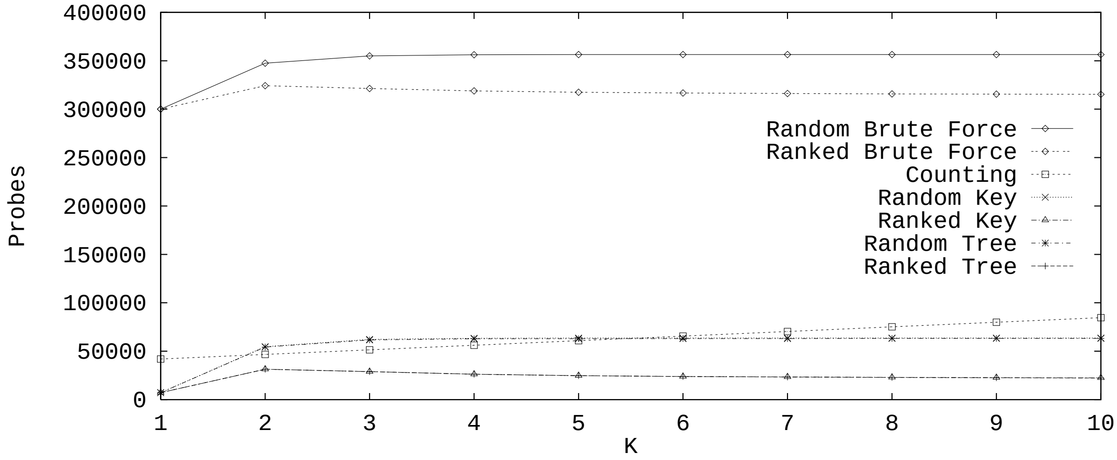

Figure 14: Number of Normalized Accesses vs K

For the number of normalized accesses, we would expect it to grow with the order 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 an order as soon as a mismatch is found. Thus the increasing number of words per order 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 an order. As they are looked up first, the matching process can terminate early, resulting in even fewer accesses.

5.4 Varying W

The size of the incoming document does not affect the space requirement of the different methods, but has a large impact on the number of disk reads and the number of normalized accesses. Figures 15 and 16 show the results when W is varied from 1000 to 18000. As expected, a smaller document size means a document contains fewer distinct words and the indexing methods have to do less processing. Hence, the indexing methods perform even better in both metrics than the Brute Force Methods when the document size is small.

For the number of normalized accesses, the overhead of initialization makes the Counting Method a bad choice when the documents are small. However, as W increases, the number of words looked up increases for all methods. The Counting Method becomes better than the other methods without

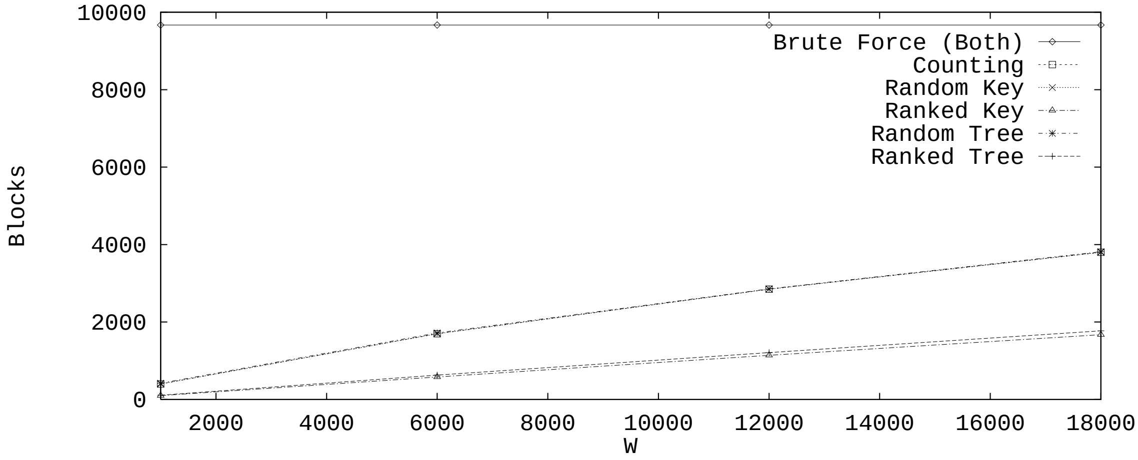

Figure 15: Number of Blocks Read Per Document vs W

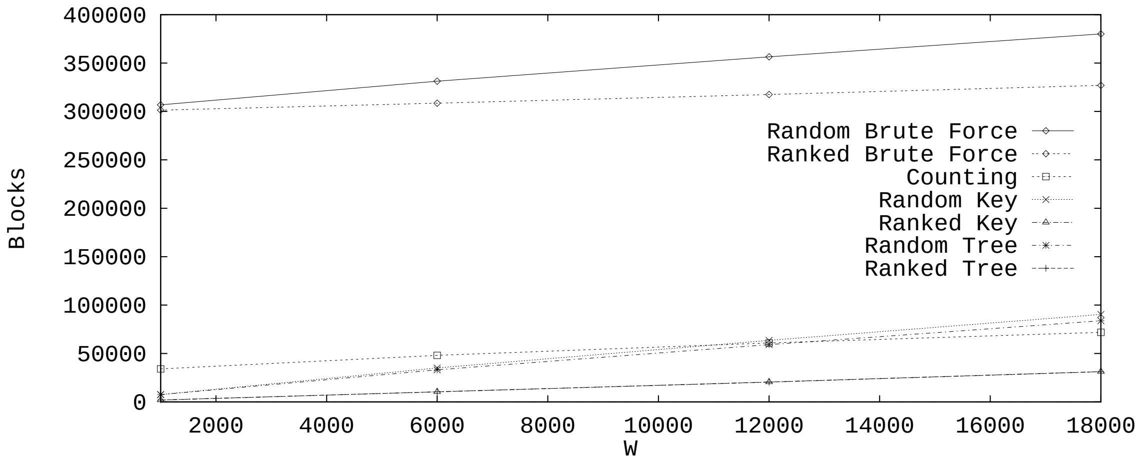

Figure 16: Number of Normalized Accesses vs W

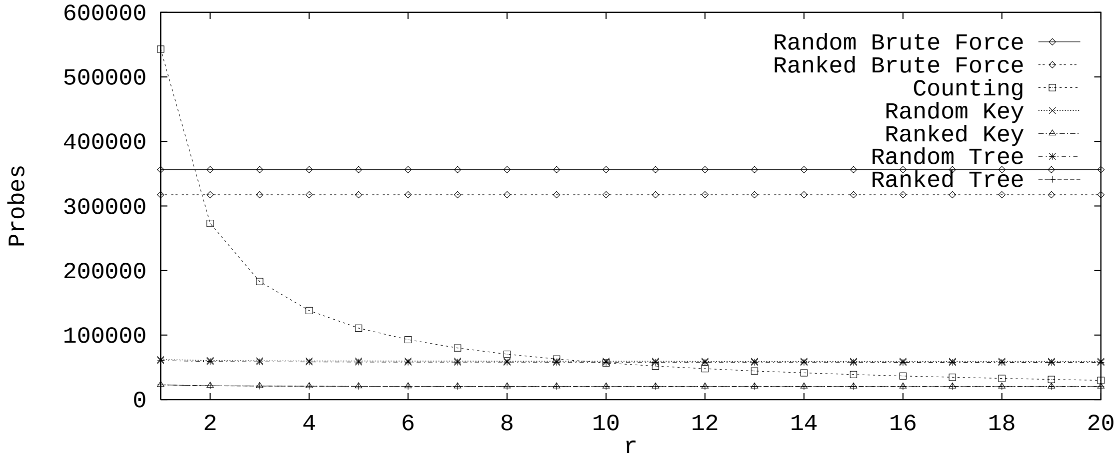

Figure 17: Number of Normalized Accesses vs r

rank information as it requires the least amout of work done per word looked up.

5.5 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 17. 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 Method. 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.

5.6 Modifying the Order Model

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

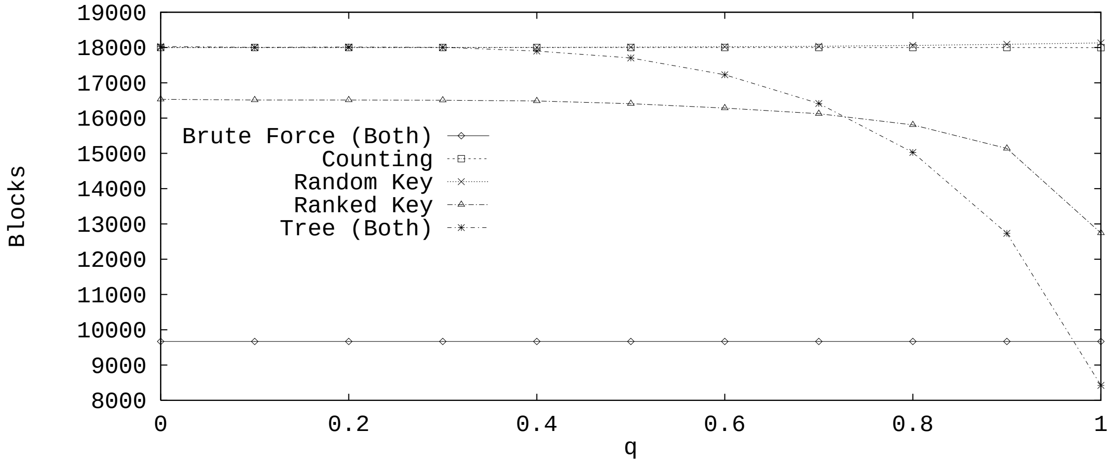

Figure 18: Total Disk Space Required vs q

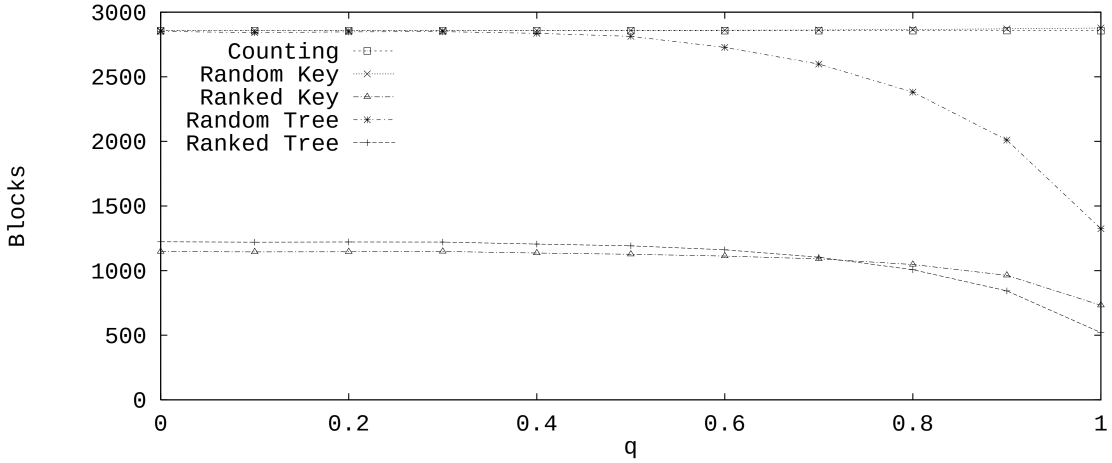

Figure 19: Number of Blocks Read Per Document vs q

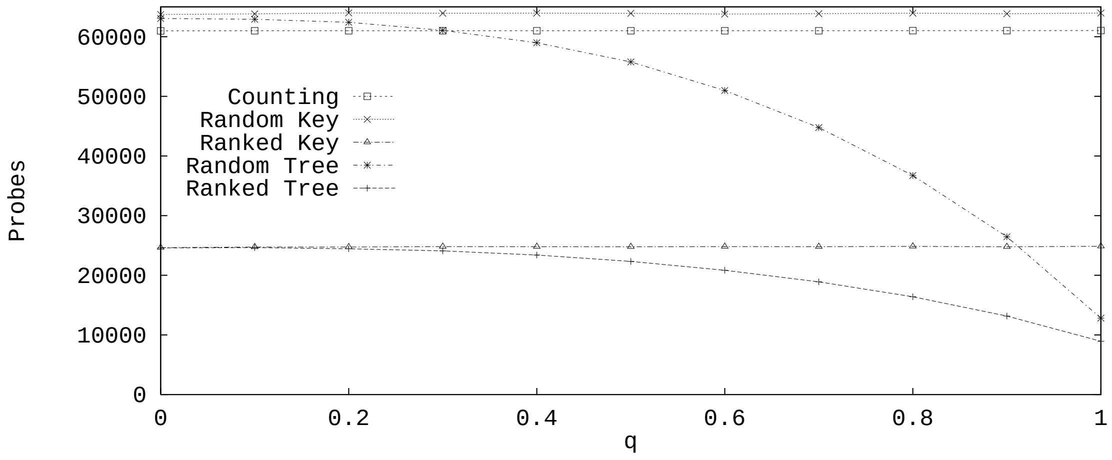

Figure 20: Number of Normalized Accesses vs q

Among the N orders, a certain fraction (fixed at 10% in our experiments) of them are generated 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 orders*. For each of the rest of the orders, we assume that it will be similar to a base order. We randomly pick one base order and mold each new order after it. The similarity parameter q controls how similar the new order and the base order are. For each word in the new order, there is a probability q that it is the same as the corresponding word in the base order. 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 orders. If q is 0, this model degenerates to the original model. If q is 1, then every order will be exactly the same as one of the base orders.

Figures 18, 19, and 20 show the performance of the different methods under the new model, as q is varied from 0 to 1; we see indeed both Tree methods improve as the degree of similarity grows. For instance, Figure 19 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 for 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.

It is interesting to see that the Ranked Key Method also improves in terms of total space required and the number of I/Os per document as q grows. As orders are clustered into fewer lists, the internal fragmentation becomes insignificant, resulting in less disk space required and a corresponding drop in the number of I/Os.

5.7 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 orders is large. However, if there are relatively few orders and they are short (e.g., say less than 50,000 orders, 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 independent order model, the Counting Method performs well among the methods without rank information with respect to the total space required and the number of I/Os per document, but is bad in terms of the number of normalized accesses per document. However, we note that its performance with respect to the last metric is very sensitive to the normalizing factor.

Under the basic order 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.

With independent orders, 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 orders, 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 orders may or may not have such similarity and we plan to investigate this issue further.

6 Extensions

6.1 Negation

In all methods except counting, negation can easily be added. In the Key and Tree methods, when an order is entered, we first pick a non-negated word (randomly or with the lowest rank) and place the order in the word's inverted set. (We may need a special list that includes all the orders that contain negated words only and check these orders against every incoming document.) The arrangement within the inverted set is then just as before. For ranking, we assign to a negated word $\neg w$ a rank of $T - w$.

During matching, a negated word is taken to be "in" a document if and only if the word itself is not in the document. We need a bit per every word identifier to indicate if the word is negated.

In the Counting Method, the inverted list of a word v is still made up of records of all orders that contain it, regardless of whether it is negated in the orders or not. Every record now has, in addition to an order identifier field, a negation flag. If word w is negated in an order O, then we set the negation flag in the O record in the w list. The TOTAL array entries are set to the numbers of non-negated words in the orders. When a document arrives, the COUNT array entries are initialized to all 0's as before. The lists of the words in the document are looked up. For a record in which the negation flag is not set, we increment the COUNT entry for that order as before. For a record with the flag set, we decrement the COUNT entry for that order. After all lists are processed, orders whose COUNT entries become equal to the TOTAL entries match the document. Note that we cannot conclude that an order matches a document as soon as the COUNT entry becomes equal to the TOTAL entry but have to wait until all lists are processed.

With the addition of negation, the space requirements for all the methods will be slightly higher. As for the number of disk I/Os and the number of normalized accesses, the effect is unclear as it depends on the frequency of the NOT operator in orders. However, we believe that the NOT operator is not frequently used in orders, and thus our results are still applicable.

6.2 Disjunction

As discussed in Section 2, an order with logical ORs can be transformed into disjunctive normal form and treated as a collection of orders. Thus, an order for $a \wedge (b \vee c)$ becomes two orders, one for $a \wedge b$ and another for $a \wedge c$. Hence, the algorithms we have presented can still be used.

However, this strategy (splitting orders) may not be the most efficient. For instance, consider the

Random Brute Force Method where a document without word a arrives. If we process two separate orders, $a \wedge b$ and $a \wedge c$, we need to probe the occurrence table twice to discover that neither order matches. If we process the combined order $a \wedge (b \vee c)$, a single probe is sufficient to see that the document does not match.

To fully understand OR orders, two issues need to be addressed:

- Are the losses for processing orders as sets of conjunctive queries really significant?
- How can the indexing algorithms be modified to handle OR queries 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 order, using thesaurus expansion or truncation, can be handled as a special case (see Section 6.4). Under this scenario, AND is arguably the dominant operator used in queries, and thus it is reasonable to optimize the index structures and the algorithms for the AND operation.

6.3 Proximity

Traditional information retrieval systems allow proximity searches. The user can specify certain words in the retrieved documents to be within some proximity criteria, e.g., in the same phrase,

$$\texttt{PHRASE (relational database)} \wedge \texttt{performance}.$$

To process proximity searches, the query is usually converted to a conjunction query; for example, the above query becomes

$$\texttt{relational} \wedge \texttt{database} \wedge \texttt{performance}.$$

The documents retrieved are then examined to see if the proximity criteria is satisfied. The same idea can be used to process orders with proximity operators. The order is matched against documents as if it had conjunctions only. If it matches, then postprocessing can be done to check if it satisfies the proximity criteria.

As the amount of postprocessing is the same for all methods and is insignificant compared with the work done in the initial matching process (since the number of matched orders is expected to be small compared with N), our performance results are still applicable.

6.4 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 order with truncated term `PSYCH*` matches documents containing words like psychology, psychologist, psychiatry, and others.

To process an order in which one or more of the words are to be expanded to a thesaurus class, we can either do the expansion when the order is first entered, and store it in expanded form, or we can do the expansion at order evaluation time. To process an order 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 order 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 an order 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, it seems more straightforward if words are represented as strings of characters in the index structures and the the occurrence table is implemented as some other data structure (e.g. binary tree) that allows range search. With this change, all the methods except Counting can be extended to handle truncation.

6.5 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 for each method what need to be done to handle batches, and then evaluate its performance.

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 orders 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 information retrieval, 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.

6.5.1 The Brute Force Methods

For the Brute Force Methods, to process a batch, we read in all the orders as before. For each order, we look up the first word in the combined occurrence table. If its bit vector is not all F's, we look up the second word and logically AND its bit vector with the first. If the result is not all F's, we look up the third word and so on. (This is equivalent to intersecting the inverted lists of the words in conventional information retrieval.) We stop as soon as the ANDed result becomes all F's. If all words in an order are looked up, then the remaining T bits correspond to documents that match the order. Again, the order in which the words are looked up can be random or ranked.

The number of blocks read is the same as in the no-batching case, but the number of look-ups is different. For the Random Brute Force Method, recall as defined in Section 4.4.2. that the probability that a randomly picked word is in a document is α. Consider i randomly picked words. First we compute the following probability:

$$\begin{aligned}
\beta_i &= \mathrm{Pr}(\text{at least one of the } b \text{ documents has all } i \text{ words}) \\
&= 1 - \mathrm{Pr}(\text{none of the } b \text{ documents has all } i \text{ words}) \\
&= 1 - (\mathrm{Pr}(\text{not all } i \text{ words are in a document}))^b \\
&= 1 - (1 - \mathrm{Pr}(\text{all } i \text{ words are in a document}))^b
\end{aligned}$$

$$= \quad 1 - (1 - \alpha^i)^b.$$

We can now compute the expected number of look-ups per order as

$$\begin{aligned}
K_{RandomBruteForce,Batching} &= \sum_{i=1}^{K} \Pr(\text{the } i\text{-th word in the order is looked up}) \\
&= \sum_{i=1}^{K} \Pr(\text{at least one of the } b \text{ documents has all first } i-1 \text{ words}) \\
&= \sum_{i=0}^{K-1} \beta_i \\
&= 1 + \beta_1 + \beta_2 + \ldots + \beta_{K-1}.
\end{aligned}$$

So the expected number of look-ups, or number of normalized accesses, per batch is

$$A_{RandomBruteForce,Batching} = N K_{RandomBruteForce,Batching}.$$

The computation for the Ranked Brute Force Method is similar. We have

$$K_{RankedBruteForce,Batching} = 1 + \gamma_1 + \gamma_2 + \ldots + \gamma_{K-1},$$

where

$$\gamma_i = 1 - (1 - \alpha_1 \alpha_2 \ldots \alpha_i)^b,$$

and the α_j's are as defined in Section 4.4.2. Hence,

$$A_{RankedBruteForce,Batching} = N K_{RankedBruteForce,Batching}.$$

6.5.2 The Counting Method

For the Counting Method, we first construct the set of distinct words in the batch and the combined occurrence table. To evaluate the orders against the batch, we need the array TOTAL of N entries and as before, in the entry for order j in the array, we keep the number of words in the order. We also need a $b \times N$ COUNT matrix, which is initialized to all 0's for each arriving document. For each distinct word appearing in the batch, we look at its combined occurrence table entry. Suppose it appears in documents $D_{i_1}, D_{i_2}, \ldots, D_{i_m}$. We look up its list of orders, $O_{j_1}, O_{j_2}, \ldots, O_{j_n}$ and increment

COUNT entry (D_{i_p}, O_{j_q}) for all $p, q, 1 \leq p \leq m, 1 \leq q \leq n$. Document i matches order j when COUNT entry (i, j) becomes equal to the TOTAL entry j in the array.

The expected number of distinct words is as if we had a document of size bW, since documents are generated independently:

$$W_{T,Batching} = \sum_{w=1}^{T}(1 - (1 - Z(w))^{bW}),$$

and the expected number of distinct words that are queried is

$$W_{S,Batching} = \sum_{w=1}^{S}(1 - (1 - Z(w))^{bW}).$$

For the expected number of blocks read per batch, we have

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

where $\mathcal{F}$ is defined in Section 4.4.1.

For the expected number of normalized accesses per batch, first we need Nb COUNT matrix accesses for the initialization. Then we get the distinct words from the distinct words set and look up the directory for the inverted lists as before. This takes $W_{T,Batching}$ array accesses and $W_{T,Batching}$ hash table look-ups. The expected number of orders in a list is $\frac{NK}{S}$. Each one of these orders increments a certain number of entries in the COUNT matrix. Consider an order in the list of word w. We increment m COUNT entries, and then compare them with the corresponding TOTAL entries, where m is the number of documents in the batch that contain the word w, which is known to be in the batch. To compute $E[m]$, we need the following: given that a word w randomly picked from U is in a batch of documents, what is the probability that w is in a particular document in the batch? This is equal to

$$\frac{\text{\# distinct words in a document that are from } U}{\text{\# distinct words in a batch that are from } U}$$

or

$$\frac{W_S}{W_{S,Batching}}.$$

So $E[m] = b\frac{W_S}{W_{S,Batching}}$. Thus, the expected number of TOTAL array and COUNT matrix accesses is

$$\begin{aligned}
& \#\text{ orders per list} \times E[m] \times \#\text{ lists examined} \\
= \; & \frac{NK}{S} \times E[m] \times W_{S,Batching} \\
= \; & \frac{NK}{S} \times \frac{bW_S}{W_{S,Batching}} \times W_{S,Batching} \\
= \; & \frac{NK}{S} bW_S.
\end{aligned}$$

The expected total number of normalized accesses is

$$A_{Counting,Batching} = W_{T,Batching} + \frac{W_{T,Batching} + bN + \frac{NK}{S}bW_S}{r}.$$

6.5.3 The Key Methods

For the Key Methods, we first construct the set of distinct words in the batch and the combined occurrence table. For each distinct word w in the batch, we retrieve its inverted list. To process an order 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 order are looked up correspond to documents that match the order.

The expressions for the number of blocks read are the same as those of Section 4.4.4, modified for the "larger document" size. For the Random Key Method, the expected number of blocks read is

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

For the Ranked Key Method, we have

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

The number of look-ups per order for the Random Key Method can be similarly computed as in the case for Brute Force:

$$K_{RandomKey,Batching} = 1 + \beta_1 + \beta_2 + \ldots + \beta_{K-2}.$$

So the expected number of normalized accesses, including the distinct words set accesses and directory look-ups, is

$$A_{RandomKey,Batching} = W_{T,Batching}(1 + \frac{1}{r}) + \frac{NK_{RandomKey,Batching}W_{S,Batching}}{S}.$$

For the number of normalized accesses for the Ranked Key Method, the analysis proved to be difficult and we resort to simulations.

6.5.4 The Tree Methods

For the Tree methods, to process a batch, we use the idea of breadth first search as before. We maintain a queue to keep track of the nodes that we have to visit, but in this case, the queue entry contains not only the pointer to a node, but also a bit vector, called the *current set*, which represents the documents in the batch that contain the words in the prefix corresponding to the node.

For each distinct word in the batch, we look up the combined inverted index to get its bit vector, which is the current set for the root of its subtree. Then we read in the subtree and start with the root. At a node, for each word in the children list, we look up its bit vector and AND it with the current set. If it is not all F's, we construct a queue entry consisting of the pointer corresponding to the word and the result of the AND operation and append it to the queue. We also check the postfix, AND the bit vectors of the words in the postfix with the current set, and stop if it becomes all F's. Bits that remain T after all words in the postfix are looked up correspond to documents that match the order(s) the node identifies. When a node is done, we take the entry at the head of the queue and find the node the pointer points to, and use the bit vector as the current set and repeat. We stop when the queue becomes empty. Again, we use simulations to evaluate the Tree methods.

6.5.5 Performance Evaluation Results

The performance of the indexing methods with batching depends a lot on the document size. With a large document size, we expect the indexing methods to become less impressive as the batch size grows, since a large portion of the orders need to be examined. With a small document size, the indexing methods should still be superior. Hence, we study batching for small documents (with W equal to 1000) and large documents (with W set to 12000).

First, we investigate how the methods perform under different batch sizes with respect to the number of I/Os and the number of normalized accesses required per batch. We can see that, with

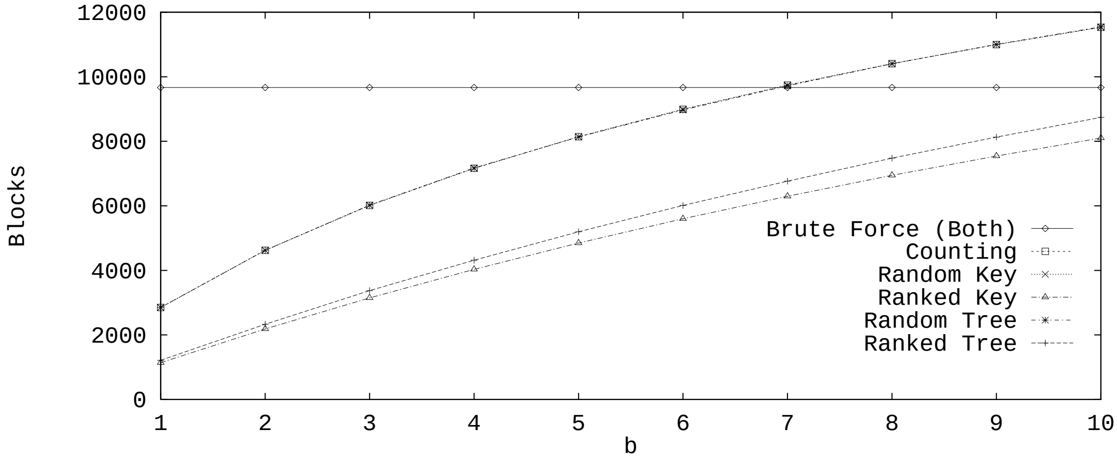

Figure 21: Number of Blocks Read Per Batch vs b, Large Document Case.

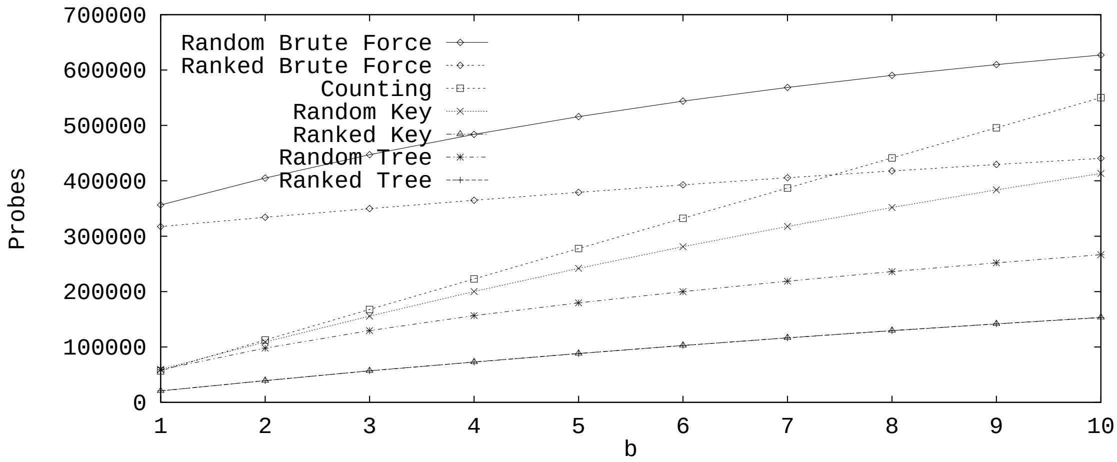

Figure 22: Number of Normalized Accesses Per Batch vs b, Large Document Case.

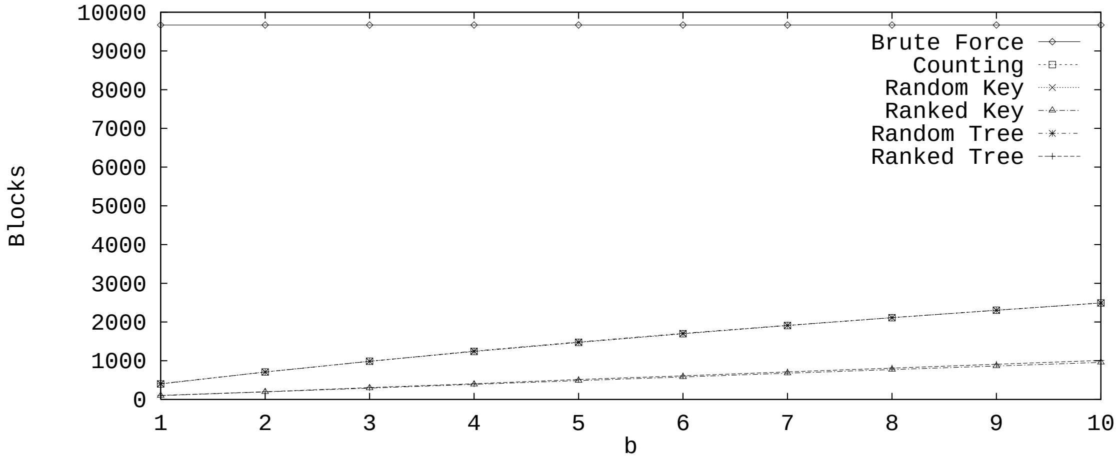

Figure 23: Number of Blocks Read Per Batch vs b, Small Document Case.

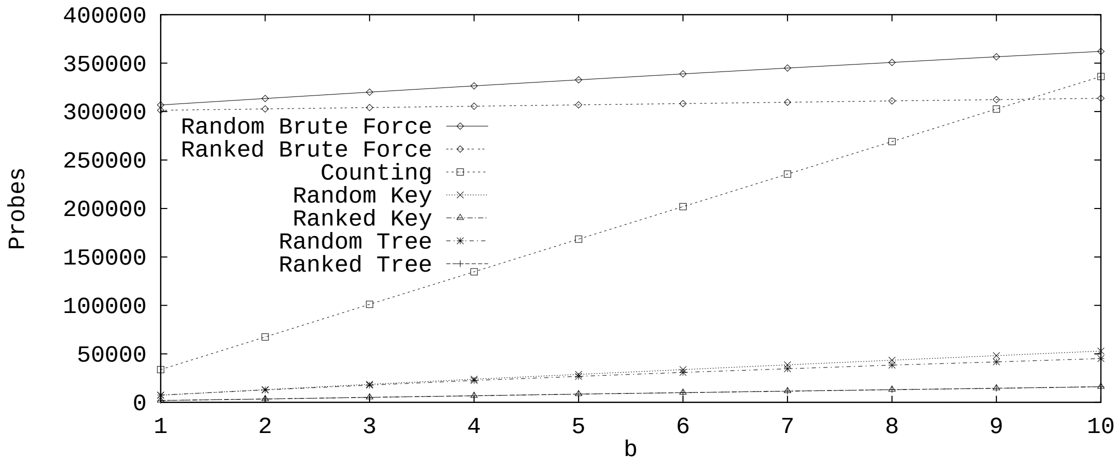

Figure 24: Number of Normalized Accesses Per Batch vs b, Small Document Case.

large documents, as the batch size increases, the performance of the indexing methods approach that of the Brute Force Methods. For the number of I/Os per batch, while the indexing methods that use rank information remain superior to the Brute Force Methods, the Brute Force Methods become better than the others at a batch size of 7. For the number of normalized accesses, all indexing methods perform better than the Brute Force Methods for the batch sizes shown.

With small documents, most of the indexing methods remain far better than the brute force methods for the values of b studied. However, the Counting Method scales badly with the batch size with respect to the number of normalized accesses per batch, mainly due to the initialization overhead.

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, most importantly, 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 Method.

7 Related Work

Related to the idea of selective dissemination is the idea of an active database [9] [10] [14] [15], one which automatically executes certain actions under certain conditions, according to some situation/action rules specified by the user. Work on active databases typically focuses on structured data in the realm of relational databases. Another recent work by Terry et. al. [17] proposes the notion of continuous queries in relational databases. Users issue continous queries, which are rewritten into incremental queries and run periodically. Their work concentrates on relational databases, while our work is concerned the dissemination of unstructured data (documents) using information retrieval techniques.

Wyle and Frei [18] give an overview of a WAN information server that extracts information from information sources (primarily Netnews) periodically and disseminates relevant information to passive users. User profiles, which consists of a number of example messages, are examined periodically (as new items are extracted) and all profiles are examined using statistical text comparsion methods. This falls into the brute force paradigm in our comparison.

Danzig et. al. [5] 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 retreival systems. They subscribe generator queries (similar to standing orders) that keep them informed of changes in these systems. However, these generator queries are not user-specified standing orders discussed in this work. Though segment trees are proposed for indexing Libary of Congress numbers, index structures for boolean generator queries are not addressed.

8 Conclusion

We present in this paper several index structures that can be used to index standing orders, together with algorithms to evaluate orders against documents. We compare their performances with the Brute Force Method that does not use any index structures, and find that while the Brute Force Method requires less disk space, it performs poorly in terms of number of disk I/Os and amount of CPU computation required to process an incoming document. We demonstrate that rank information of the words can be used to significantly improve the performances of the varioius methods. We also outline the scenarios under which processing documents in batches is beneficial.

References

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

[2] P.G. Anick, et. al., "A Direct Manipulation Interface for Boolean Information Retrieval via Natural Language Query," Proc. ACM SIGIR Conference, Brussels, Belgium, September 1990, pp.135-50.

[3] N.J. Belkin and W. B. Croft, "Information Filtering and Informatin Retrieval: Two Sides of the Same Coin?" Commun. ACM, Vol. 35, No. 12, Dec 1992, pp.29-38.

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

[5] P. Danzig, J. Ahn, J. Noll, K. Obraczka, "Distributed Indexing: A Scalable Mechanism for Distributed Information Retrieval," Proc. ACM SIGIR Conference, Chicago, October 1991, pp.220-229.

[6] M. Horton, "Standard for Interchange of USENET messages," UNIX Documentation.

[7] M. Horton, "How to Read the Network News," UNIX Documentation.

[8] D. Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading, Massachusetts, 1973.

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

[10] A. Rosenthal, S. Chakravarthy, B. Blaustein, J. Blakely, "Situation Monitoring for Active Databases," Proc. Very Large Data Bases, Amsterdam, Holland, 1989, pp.455-467.

[11] G. Salton, *Automatic Information Organization and Retrieval*, McGraw-Hill, New York, 1968, pp.381-383.

[12] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.

[13] G. Salton, "A Simple Blueprint for Automatic Boolean Query Processing," Information Processing & Management, Vol. 24, No. 3, 1988, pp.269-80.

[14] U. Schreier, H. Pirahesh, R. Agrawal, C. Mohan, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS," Proc. Very Large Data Bases, Barcelona, Spain, 1991, pp.469-478.

[15] M. Stonebraker, A. Jhingran, J. Goh, S. Potamianos, "On Rules, Procedures, Caching and Views in Database Systems," Proc. ACM SIGMOD Symposium on Management of Data, Atlantic City, May 1990, pp.281-290.

[16] A. Tomasic and H. Garcia-Molina, "Performance of Inverted Indices in Distributed Text Document Retrieval Systems," (to appear) Proc. Parallel and Distributed Information Systems Conference, San Diego, 1993.

[17] D. Terry, D. Goldberg, D. Nichols, B. Oki, "Continuous Queries over Append-Only Databases," Proc. ACM SIGMOD Symposium on Management of Data, San Diego, May 1992, pp.321-330.

[18] M.F. Wyle and H.P. Frei, "Retrieving Highly Dynamic, Widely Distributed Information," Proc. ACM SIGIR Conference, Cambridge, Massachusetts, June 1989, pp.108-115.

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