UNIT -1

Introduction to Information Storage and Retrieval System

Topics

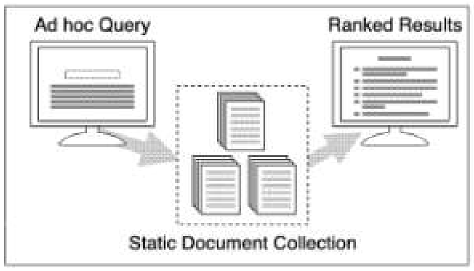
- Introduction
- Domain Analysis of IR Systems
- IR and Other types of Information Systems.
- IR System Evaluation
- Introduction to Data Structures and Algorithms related to Information Retrieval
- Basic Concepts
- Data Structures
- Algorithms

INTRODUCTION

The Systems which are used to store information gathered from different sources in such a way that it can be retrieved easily and effectively upon request are referred to as information storage and retrieval systems.

Collecting information from different resources and storing it in either storage room(maintaining paper records) or the storage devices such as hard disk, DVD, CD is called as information storage. This information may be in any of the form that is audio, video, text. Information Retrieval System is mainly focus electronic searching and retrieving old documents.

The process of searching, fetching and serving of information to the requested users is information retrieval. An IR System is capable of performing operations like methods for adding documents to the database, modifying or deleting them from the database, methods for searching and serving appropriate document to the users.
Information Retrieval is an activity of obtaining relevant documents based on user needs from collection of retrieved documents.



Basic information retrieval system

A static, or relatively static, document collection is indexed prior to any user query. A query is issued and a set of documents that are deemed relevant to the query are ranked based on their computed similarity to the query and presented to the user query.

Information Retrieval (IR) is devoted to finding relevant documents, not finding simple matches to patterns.

Automated information retrieval (IR) systems were originally developed to help manage the huge scientific literature that has developed since the 1940s.

Many university, corporate, and public libraries now use IR systems to provide access to books, journals, and other documents.

Commercial IR systems offer databases containing millions of documents in myriad subject areas.

Dictionary and encyclopedia databases are now widely available for PCs. IR has been found useful in such disparate areas as office automation and software engineering.

DOMAIN ANALYSIS OF IR SYSTEMS

Domain Analysis of IR System can be defined as the process of analyzing number of related systems offering a method to create a conceptual framework. This framework is capable of determining, understanding and using data structures, algorithms and techniques efficiently.

Prieto-Diaz and Arrango (1991) developed a framework as domain analysis which is nothing but a system analysis for multiple related systems. Via Domain Analysis, one attempts to discover and record the similarities and difference among related systems.

The primary steps of domain analysis involve the following activities:

i. Determining the essential concepts and vocabulary in the domain.
ii. Defining these essential concepts and vocabulary.

Arranging essential concepts and vocabulary with the help of facated classification.

The figure below is facated classification for IR systems:

Conceptual model	File Structure	Query Operations	Term Operations	Document Operations	Hardware
Boolean	Flat File	Feedback	Stem	Parse	VonNeuman
Extened Boolean	Inverted File	Parse	Weight	Display	Parallel
Probablistic	Signature	Boolean	Thesaurus	Cluster	IR Specific
String Search	Pat Trees	Cluster	Stoplist	Rank	Optical Disk
Vector Space	Graphs Hashing		Truncation	Sort, fieldmark, assign id	Magnetic Disk

The top row represents the facates(attributes used by IR System) which are the parts of the IR System and are common in all the systems but may vary in terms of their values or sub attributes .

The facet values are known as terms. Occurrence of one term does not restrict the other terms to occur within a facet. Moreover a facet can have one or more facet values for a single system.

Facets	Facet Values
Term Operations	Stem Stop List Truncation
File Structure	Inverted File
Conceptual Model	Boolean
Hardware	Von Neumann Magnetic Disk
Document Operations	Parse Display Sort field Mask Assign Id’s
Query Operations	Parse, Boolean

Table: Facets and Facet Values for CATALOG IR System

Conceptual Models:

The most common facet in the facated classification is conceptual model. An IR conceptual model is a general approach to IR systems. Conceptual model has been classified by two analysis :- Faloutsos categorization and Belkin and Croft categorization.

Faloutsos (1985) gives three basic approaches: text pattern search, inverted file search, and signature search.

Belkin and Croft (1987) categorize IR conceptual models differently. They divide retrieval techniques first into exact match and inexact match. The exact match category contains text pattern search and Boolean search techniques. The inexact match category contains such techniques as probabilistic, vector space, and clustering.The disadvantage of this classification is that an individual system may employ multiple categories of this classification.

Almost all of the IR systems fielded today are either Boolean IR systems for major document collections or text pattern search systems for handling small document collections(for ex: personal collections of files). Text pattern search queries are strings or regular expressions. The grep family of tools, described in Earhart (1986), in the UNIX environment is a well-known example of text pattern searchers.

In a Boolean IR system, documents are represented by sets of keywords, usually stored in an inverted file. An inverted file is a list of keywords and identifiers of the documents in which they occur. Boolean queries are keywords connected with Boolean logical operators (AND, OR, NOT). While Boolean systems have been criticized, improving their retrieval effectiveness has been difficult.

The conceptual model facet also focuses on the performance enhancements of IR systems with the information associated with statistical distribution of terms. The information maintains the term repeatitions with the documents , collections or subset of document collections. This information maintains the term repeatitions.

The statistical models such as vector space,probabilistic or clustering models do the statistical distribution of terms where every document in retrieved collection is allocated with probability of relevance.This helps in ordering of documents in accordance to the probability of relevance .

File Structures Facet of IR System:

This facet is also an important part of IR System because a decision must be made regarding the type of file structure that has to be adopted in design of IR system . There are different types of file structures that can be adopted by IR system. They are

- Inverted File
- Flat File
- Signature File
- Graphs
- PAT Trees

Inverted File: This file structure is a type of indexed file having the following fields,

Keyword: Indexing term defining the document

Document Id: uses a unique identifiers to represent a document.

Field Id: uses a unique name to represent the path of keyword within document.

In addition to these fields, few systems maintains addresses of sentences , paragraphs .
Searching is done by looking up query terms in inverted file.

Flat Files: The file structure creates a file with one or more documents usually as ASCII or EBC DIC text. The search is usually done via pattern matching.

For Ex: UNIX carryout this type of storage using UNIX directories within which one entire document collection can be placed. Searching is carried out through pattern searching tools like grep or awk.

Signature Files: This file structure deals with the signature (bit patterns) for representing the document. The signature can be created within different methods. One common method for creating the signatures is to split the document into logical blocks each containing a fixed numbers of distinct significant(i.e non stoplist, words). Each word in the block is hashed to give a signature. Through the generated signature, a block signature is produced by performing ‘OR’ operation on them. These block singnatures are then processessed by performing concatenation to generate the document signature. Searching is done by comparing the signature of queries with document signatures.

Graphs: Collection of nodes connected by arc. It is also known as networks. A document can be represented as a kind of graph called a semantic net which can be used to represent the semantic relationships in text. These graph based techniques are impractical at present because of high manual effort.

PAT Trees(Patricia trees): They are specially used for sistrings in a text. Consider that the set of document is arranged as an orderly numbered set of characters , then the sistring can be defined as the subset of characters from the set starting at a particular point and expanding itself arbitrarily towards the right side of the set.

Query Operations:
Queries the statement generated by users demanding certain amount of information. The queries are feeded to the IR System through specific means. Serving a query involves the operation that are considered as the function of query type and IR system capabilities.
Parsing is the common query operation where the query is divided into constituent parts. For instance, Boolean queries are divided into terms and operations upon parsing, so that it can easily retrieve the collection of document identifiers that are linked with each query term.
Feedback is a special type of query which makes use of historical data associated with search and modify the queries.
Term Operations:
There are various operations that are performed on terms in an IR System. They are:

- Stemming
- Weighting
- Thesaurus
- Stoplist
- Truncation

a) Stemming: This operation involve interconnecting of relavant word in an automated way. The interconnected of words is typically reduces the words that resembles like a common root.
b) Weighting: This operation allocates numbering to the indexing or query terms taking into consideration the information regarding the statistical distribution of terms.
c) Thesaurus: This operation combines the words that are equal (or) similar meanings are related to each other.
d) Stop List: This operation deals with the words that may not have indexing value. It simply removes the potential indexing terms by finding their presence in the stoplist.
e) Truncation: This operation manually combines the terms with help of wildcard characters in the word where the truncated term is used for matching multiple words.

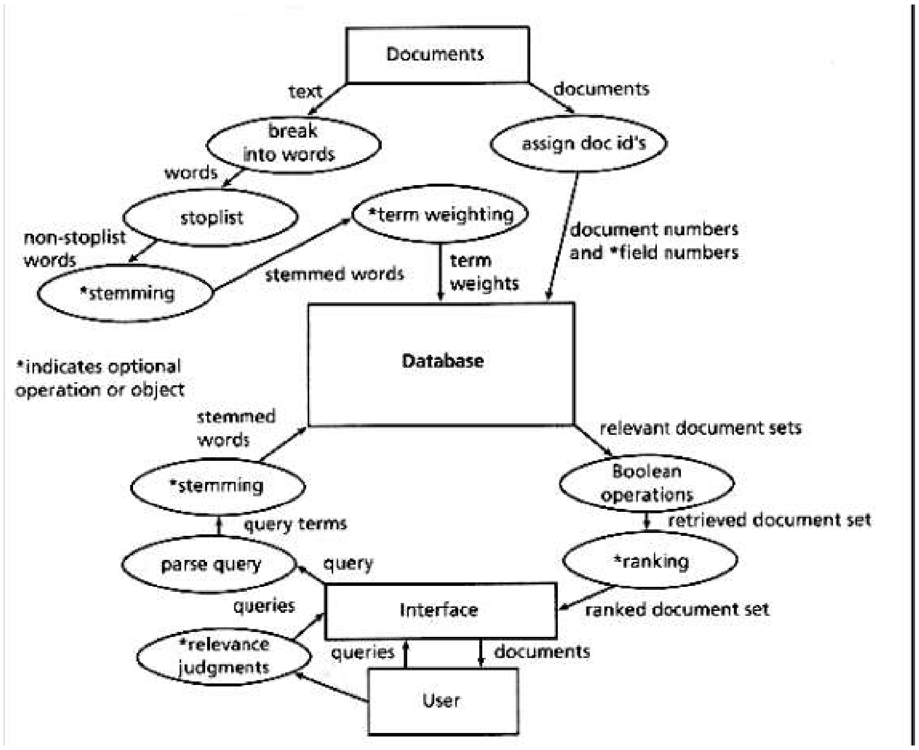
Document Operations:
Documents are the primary objects in IR systems and there are many operations for them. Some of the operations are:

- Addition of documents to database
- Masking of document fields
- Sorting of document fields
- Displaying documents
- Arranging the documents with some priority
- Clustering of documents

Hardware:
The IR System get affected by the hardware employed in them. This is because the operating speed which is a crucial element in interactive information system and the quantity and type of information to be stored in IR system are entirely dependent on hardware.
Most IR systems in use today are implemented on von Neumann machines--general purpose computers with a single processor. Most of the discussion of IR techniques assumes a von Neumann machine as an implementation platform. The computing speeds of these machines have improved enormously over the years, yet there are still IR applications for which they may be too slow. In response to this problem, some researchers have examined alternative hardware for implementing IR systems. There are two approaches—parallel computers and IR specific hardware.

Functional View of Paradigm IR System:

Figure shows the activities associated with a common type of Boolean IR system, chosen because it represents the operational standard for IR systems.



When building the database, documents are taken one by one, and their text is broken into words.

The words from the documents are compared against a stoplist--a list of words thought to have no indexing value.

Words from the document not found in the stoplist may next be stemmed.

Words may then also be counted, since the frequency of words in documents and in the database as a whole are often used for ranking retrieved documents.

Finally, the words and associated information such as the documents, fields within the documents, and counts are put into the database.

The database then might consist of pairs of document identifiers and keywords as follows.

keyword1 - document1-Field_2

keyword2 - document1-Field_2, 5
keyword2 - document3-Field_1, 2
keyword3 - document3-Field_3, 4

.
.
.
.

keyword-n - document-n-Field_i, j

Such a structure is called an inverted file.

In an IR system, each document must have a unique identifier, and its fields, if field operations are supported, must have unique field names.

To search the database, a user enters a query consisting of a set of keywords connected by Boolean operators (AND, OR, NOT).

The query is parsed into its constituent terms and Boolean operators.
These terms are then looked up in the inverted file and the list of document identifiers corresponding to them are combined according to the specified Boolean operators.
If frequency information has been kept, the retrieved set may be ranked in order of probable relevance. The result of the search is then presented to the user.

IR AND OTHER TYPES OF INFORMATION SYSTEMS

	Data Object	Primary Operation	Database Size
IR	document	Retrieval	small to very large
DBMS	table	Retrieval	small to very large
AI	logical statements	inference	Usually small

Consider DBMS and Artificial Intelligence systems for illustrating relationship between IR system and other information systems.

The IR system make use of document as data objects where as DBMS and AI uses table and logical statements respectively. Here the usable structure of data object in IR system is less when compared with DBMS and AI system. IN DBMS and AI there is a possibility of manual assessment of document and also data storage about syntax and semantics. Hence, it is practically complex in case of hugeset of documents.
The IR systems compared to DBMS contains very large databases .
The IR system has a typical feature that its retrieval is uncertain, that is there is no assurance about the exact match in required documents to the retrieved documents.
IR System has following features

- Enabling user to add , change and delete in databse.
- Facilitating the user with the method for feeding query
- The IR system must support the database to handle megabyte or gigabyte range.
- The IR system must retrieve relavat documents quickly upon request.

IR SYSTEM EVALUATION

IR systems can be evaluated in terms of many criteria including execution efficiency, storage efficiency, retrieval effectiveness.
Execution efficiency is measured by the time it takes a system, or part of a system, to perform a computation
Storage efficiency is measured by the number of bytes needed to store data. It is typically measured through space overhead which is given by following formula,

$$\text{Space overhead} = \frac{\text{Size of the index files} + \text{Size of the document files}}{\text{Size of the document files}}$$

Retrieval efficiency is the factor that is dependent on the decisions associated with relevancy of the documents. The retrieval can be calculated as
A) Recall: Recall is the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in the database.
B) Precision: Precision is the ratio of the number of relevant documents retrieved over the total number of documents retrieved.

Both recall and precision have values between 0 and 1.

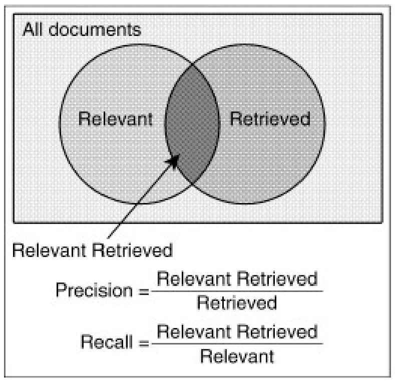
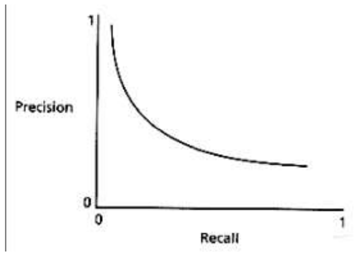

All documents
Relevant
Retrieved
Relevant Retrieved
Precision = Relevant Retrieved / Retrieved
Recall = Relevant Retrieved / Relevant

Fig: PRECISION AND RECALL

Recall-precision plots show that recall and precision are inversely related.



A combined measure of recall and precision, E, has been developed by van Rijsbergen (1979). The evaluation measure E is defined as:

$$\boxed{E = 1 - \frac{(1 + b^2) P R}{b^2 P + R}}$$

where P = precision, R = recall, and b is a measure of the relative importance

INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS IN IR

BASIC CONCEPTS

The basic concepts related to text include

Strings

Regular Expressions

Finite Automata

Strings: A string is a collection of symbols choosen from some alphabets. The set of strings over an alphabet **ε={a,b}** is denoted by $\boldsymbol{\varepsilon}^*$ which is a set containing empty and all combinations of a and b.

$\boldsymbol{\varepsilon}^*$ = {Empty,a,b,ab,ba,....}

Operations on strings are concatenation, reverse, string exponentiation, kleen closure, positive closure.

Similarity between strings: It can be determined as

$d(s_1, s_1)=0$

$d(s_1, s_2)>=0$

$d(s_1, s_3)<= d(s_1, s_2) + d(s_1, s_3)$

where, d= distance

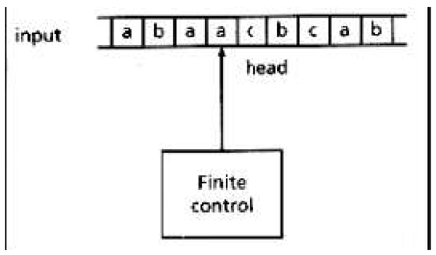
s_1 , s_2 , s_3 = different strings

Regular expressions: A regular expression is a concise notation for denoting regular sets. These describe the language accepted by the finite automata.

A language over an alphabet **Σ** is a set of strings over . Let L1 and L2 be two languages. The operations which can be performed on these languages are

Union
Concatenation
Closure on kleen star

Finite automata: A finite automaton is a mathematical model of a system. The automaton can be in any one of a finite number of states and is driven from state to state by a sequence of discrete inputs.



A finite automaton

Formally, a finite automaton (FA) is defined by a 5-tuple (Q, , , q0, F) where
Q is a finite set of states,
Σ is a finite input alphabet,
q_0 is the initial state,
F is the set of final states, and
δ is the (partial) transition function mapping .

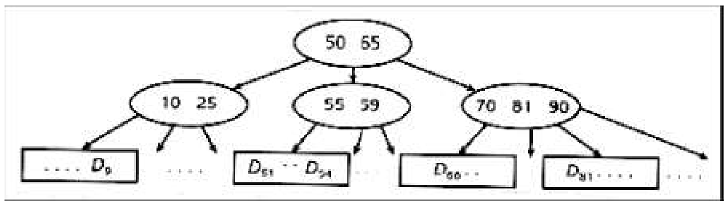
DATA STRUCTURES

There are three basic data structures that are used to organize data:

Search trees
Digital trees
Hashing

Search trees:
The most well-known search tree is the binary search tree. Each internal node contains a key, and the left subtree stores all keys smaller that the parent key, while the right subtree stores all keys larger than the parent key. Binary search trees are adequate for main memory. However, for secondary memory, multiway search trees are better, because internal nodes are bigger. In particular, we describe a special class of balanced multiway search trees called B-tree.
A B-tree of order m is defined as follows:
The root has between 2 and 2m keys, while all other internal nodes have between m and 2m keys.
If k_i is the i-th key of a given internal node, then all keys in the i - 1 th child are smaller than k_i, while all the keys in the i-th child are bigger.
All leaves are at the same depth.
Usually, a B-tree is used as an index, and all the associated data are stored in the leaves or buckets. This structure is called B^+-tree.

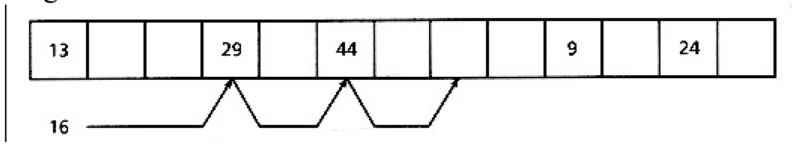
Example of B$^+$



A B+ -tree example (Di denotes the primary key i, plus its associated data).

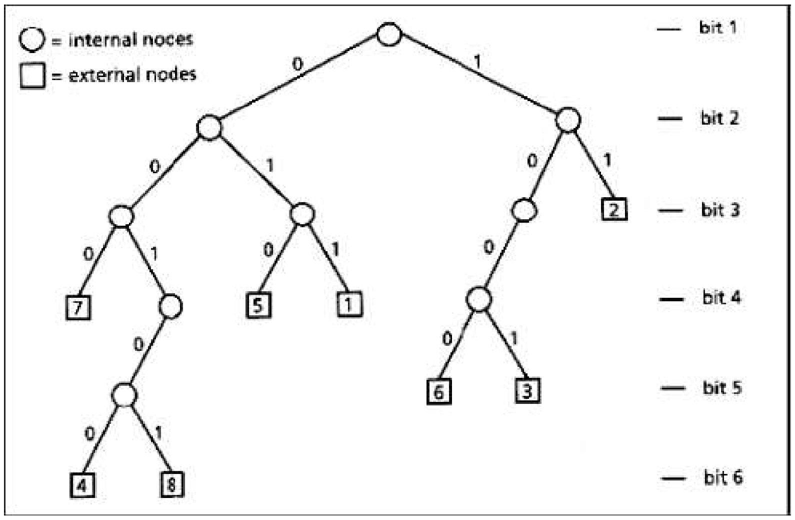
Hashing:
A hashing function h (x) maps a key x to an integer in a given range.
The hashing value is also called a signature.
A hashing function is used to map a set of keys to slots in a hashing table.
If the hashing function gives the same slot for two different keys, we say that we have a collision. Hashing techniques mainly differ in how collisions are handled. There are two classes of collision resolution schemas: open addressing and overflow addressing.

In open addressing , the collided key is "rehashed" into the table, by computing a new index value. The most used technique in this class is double hashing, which uses a second hashing function . The main limitation of this technique is that when the table becomes full, some kind of reorganization must be done.



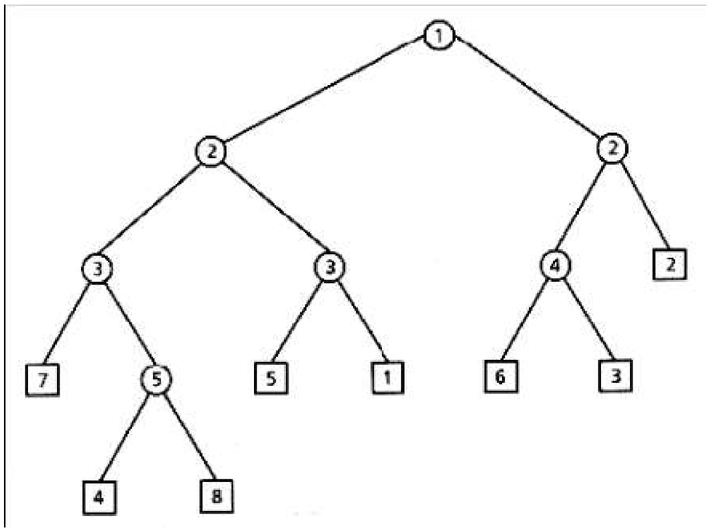
Insertion of a new key using double hashing.
A hashing table of size 13, and the insertion of a key using the hashing function **h (x) = x mod 13.**
In overflow addressing , the collided key is stored in an overflow area, such that all key values with the same hashing value are linked together. The main problem of this schema is that a search may degenerate to a linear search.

Digital Trees:
Efficient prefix searching can be done using indices. One of the best indices for prefix searching is a binary digital tree or binary trie constructed from a set of substrings of the text. This data structure is used in several algorithms.
Tries are recursive tree structures that use the digital decomposition of strings to represent a set of strings and to direct the searching.
If the alphabet is ordered, we have a lexicographically ordered tree. The root of the trie uses the first character, the children of the root use the second character, and so on. If the remaining subtrie contains only one string, that string's identity is stored in an external node.



Binary trie (external node label indicates position in the text) for the first eight suffixes in "01100100010111 . . .".

A Patricia tree is a trie with the additional constraint that single descendant nodes are eliminated. This name is an acronym for "Practical Algorithm To Retrieve Information Coded In Alphanumerical." A counter is kept in each node to indicate which is the next bit to inspect.



Patricia tree (internal node label indicates bit number).

For n strings, such an index has n external nodes (the n positions of the text) and n -1 internal nodes. Each internal node consists of a pair of pointers plus some counters. Thus, the space required is O(n).

ALGORITHMS

We can identify three main types of algorithms – Retrieval algorithms, Filtering Algorithms & Indexing Algorithms.

Retrieval algorithms:
The retrieval algorithms fetch the data from the textual databases. These are primary category of the algorithms for information retrieval systems. These algorithms have been divided into types depending on requirement of additional memory.

Sequential scanning of text
Indexed text

Sequential Scanning of text: This algorithm requires additional memory based on query size rather than database size. It also requires the running time to be least proportional to the text size.
Indexed Text: This algorithm makes use of index of the text for searching purpose. It is capable of minimizing the search time. However, size of index is based on database size (proportional) due to which search time cannot be compared with amount of text.
However, retrieval algorithms usually perform searching as:
Consider given string be t (in text) ,regular expression q (a query) and information i that acquired after preprocessing of the pattern or the text. Then the expression is given as

$$t \in \sum * q \sum * ?$$

This search provides the following information

The place where the occurrence of q is present i.e., t $\in \sum * q \sum *$, determine the location m > =0 such that t $\in \sum^m q \sum *$. For ex the first occurrence of q will be least on satisfying the criteria.
The frequency of occurrences of pattern in the text i.e., the frequency of all possible values of m in the earlist category.
All the places of pattern occurrence i.e, set of all the likely values of m.

Commonly the following difficulty arises during this search.
The result will be of less importance if the € is member od L(q) ,where q is string.
The retrieval algorithms need to efficient as they deal with online queries which requires a shorter response time.

Filtering Algorithms:
This category of algorithms filters the given text and sends back result. Filtering is performed in IR system doe to the reasons like reduction of text size , standardization of text .
Most common operations on filtering algorithms are:

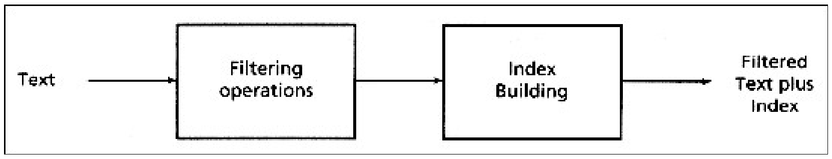
Common words removed using a list of stopwords.
Uppercase letters transformed to lowercase letters.
Special symbols removed and sequences of multiple spaces reduced to one space.
Numbers and dates transformed to a standard format .
Spelling variants transformed using Soundex -like methods.
Word stemming (removing suffixes and/or prefixes).
Automatic keyword extraction.
Word ranking.

Disadvantages of filtering algorithms:

The common words , special symbols or upper case letters cannot be searched.
The text pieces that have been mapped to same internal form can be differentiated.

Indexing algorithms:
The main purpose of indexing algorithms is speed up the textual searching as they employ indices. The indices are of multiple categories depending upon the retrieval methods.
Examples include indices trees, signature files, inverted files.

The clustered data structures and Direct Acyclic Word Graph (DAWG) are not based on hashing or tree rather it is based on automata theory.



Text Processing in indexing algorithm

The preprocessing time incurred for developing the index can be minimized by employing the index searching.
For ex: id O(n logn) time is needed for developing index then the query can be searched in database atleast O(n) times reducing the preprocessing cost obtained in index creation.
Thus, the O(logn) time can be extended for the complete query time where the O(logn) is the preprocessing time.

UNIT-2
INVERTED FILES

INTRODUCTION:

Three of the most commonly used file structures for information retrieval can be classified as lexicographical indices (indices that are sorted), clustered file structures, and indices based on hashing. One type of lexicographical index, the inverted file, is presented in this chapter, with a second type of lexicographical index, the Patricia (PAT) tree.

The concept of the inverted file type of index is as follows.

- Assume a set of documents
- Each document is assigned a list of keywords or attributes, with optional relevance weights associated with each keyword (attribute).
- An inverted file is then the sorted list (or index) of keywords (attributes), with each keyword having links to the documents containing that keyword in below figure.

This is the kind of index found in most commercial library systems. The use of an inverted file improves search efficiency by several orders of magnitude, a necessity for very large text files.

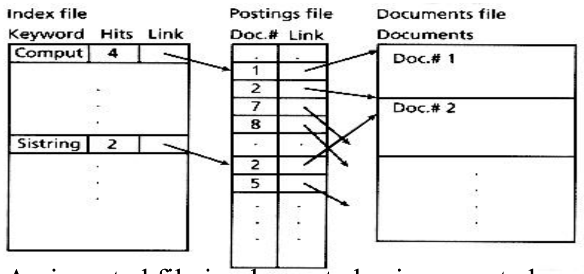


Fig: An inverted file implemented using a sorted array

Data Structures used in inverted file:-

- Sorted array
- B-Trees
- Tries

These are sorted indices and can efficiently support range queries such as documents having keywords that start with “comput”.

Range Query: It is a common DB operation that retrieves all records where some value is between an upper and lower boundary.

1) The Sorted Array:-

- It stores the lilst of keywords in a sorted array
- Including the no.of documents associated with easch keyword.
- And a link to the docments containing that keyword.

This array is commonly searched using a binary searach.

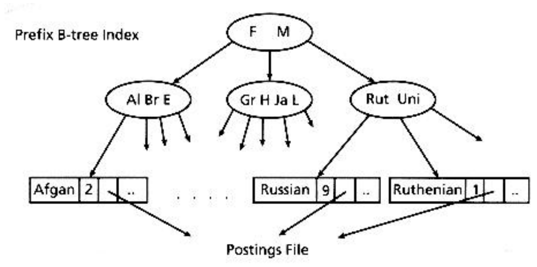
DisAdvantage: updating index is expensive.

Advantage: easy to implement and a reasonably fast.

2) B-Trees:- Another implementation structure for an inverted file is a B-tree.

- It is efficient for dynamic data(heavily updated).
- A special case of the B-tree, the prefix B-tree, uses prefixes of words as primary keys in a B -tree index (Bayer and Unterauer 1977) and is particularly suitable for storage of textual indices.
- Each internal node has a variable number of keys.
- Each key is the shortest word (in length) that distinguishes the keys stored in the next level. The key does not need to be a prefix of an actual term in the index.
- The last level or leaf level stores the keywords themselves.

- Because the internal node keys and their lengths depend on the set of keywords, the order (size) of each node of the prefix B -tree is variable.
- Updates are done similarly to those for a B-tree to maintain a balanced tree.
- **<u>DisAdvantage:</u>** Use more space when compare to sorted array.
- **<u>Advantage:</u>** Updates are much easier and search time
- The prefix B tree method breaks down if there are many words with the same (long) prefix.
- In this case, common prefixes should be further divided to avoid wasting space.



<u>Prefix B-tree:</u>

- Compared with sorted arrays, B-trees use more space.
- However, updates are much easier and the search time is generally faster, especially if secondary storage is used for the inverted file (instead of memory).
- The implementation of inverted files using B-trees is more complex than using sorted arrays, and therefore readers are referred to Knuth (1973) and Cutting and Pedersen (1990) for details of implementation of B -trees, and to Bayer and Unterauer (1977) for details of implementation of prefix B -trees.

3) <u>Tries:</u>

- Inverted files can also be implemented using a trie structure .
- This structure uses the digital decomposition of the set of keywords to represent those keywords.
- A special trie structure, the Patricia (PAT) tree, is especially useful in information retrieval . An additional source for tested and optimized code for B-trees and tries is Gonnet and Baeza-Yates (1991).

<u>Buliding an inverted file using a sorted array:</u>

1) The i/p text must be parsed into a list of words along with their location in the text.
2) The list must then be inverted from a list of terms in location order to a list of terms ordered for use in searching.
3) It is post processing of those inverted files, such as adding term weights or for reorganising or compressing the files.

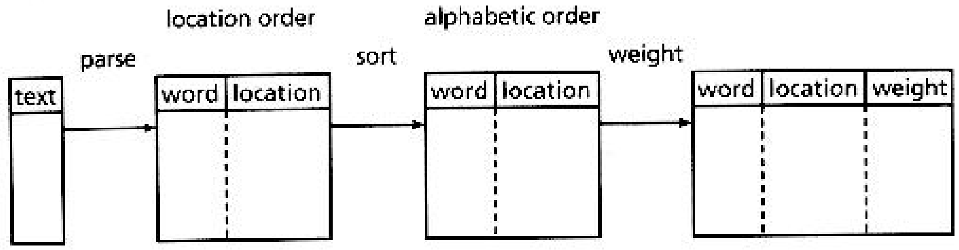
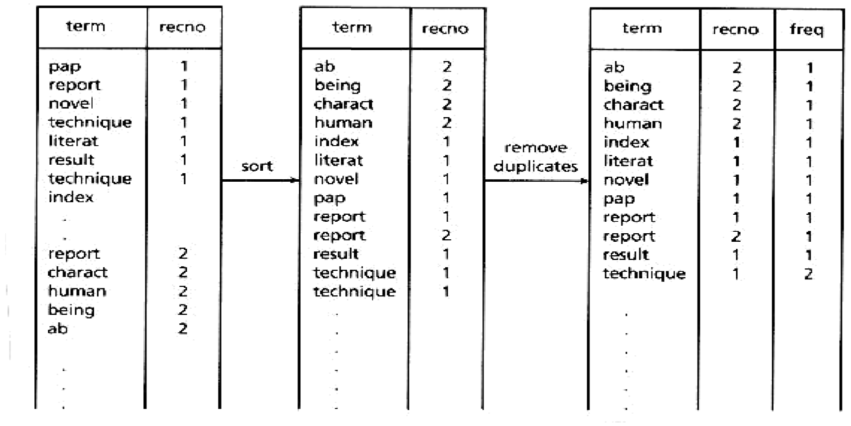


Fig: Overall schematic of sorted array inverted file creation

Creating the initial word list requires several different operations:-

- The individual words must be recognized from the text.
- Each word is then checked against a stoplist of common words.
- If it can be considered a non-common word, may be passed through a stemming algorithm. The resultant stem is then recorded in the word -within-location list.
- The word list resulting from the parsing operation (typically stored as a disk file) is then inverted.
- This is usually done by sorting on the word (or stem), with duplicates retained.
- Even with the use of high-speed sorting utilities, however, this sort can be time consuming for large data sets (on the order of n log n).
- One way to handle this problem is to break the data sets into smaller pieces, process each piece, and then correctly merge the results.
- After sorting, the duplicates are merged to produce within -document frequency statistics. (A system not using within -document frequencies can just sort with duplicates removed.)
- Note that although only record numbers are shown as locations in Figure 3.4, typically inverted files store field locations and possibly even word location.
- These additional locations are needed for field and proximity searching in Boolean operations and cause higher inverted file storage overhead than if only record location was needed. Inverted files for ranking retrieval systems (see Chapter 14) usually store only record locations and term weights or frequencies.


sort
remove duplicates

term	recno
pap	1
report	1
novel	1
technique	1
literat	1
result	1
technique	1
index	
.	
.	
report	2
charact	2
human	2
being	2
ab	2
.	
.	
.	

term	recno
ab	2
being	2
charact	2
human	2
index	1
literat	1
novel	1
pap	1
report	1
report	2
result	1
technique	1
technique	1
.	
.	
.	

term	recno	freq
ab	2	1
being	2	1
charact	2	1
human	2	1
index	1	1
literat	1	1
novel	1	1
pap	1	1
report	1	1
report	2	1
result	1	1
technique	1	2
.		
.		
.		

Inversion of word List:

- Although an inverted file could be used directly by the search routine, it is usually processed into an improved final format.
- This format is based on the search methods and the (optional) weighting methods used.
- A common search technique is to use a binary search routine on the file to locate the query words.
- This implies that the file to be searched should be as short as possible, and for this reason the single file shown containing the terms, locations, and (possibly) frequencies is usually split into two pieces.
- The first piece is the dictionary containing the term, statistics about that term such as number of postings, and a pointer to the location of the postings file for that term.
- The second piece is the postings file itself, which contains the record numbers (plus other necessary location information) and the (optional) weights for all occurrences of the term.
- In this manner, the dictionary used in the binary search has only one "line" per unique term. Below Figure illustrates the conceptual form of the necessary files; the actual form depends on the details of the search routine and on the hardware being used.

- Work using large data sets (Harman and Candela 1990) showed that for a file of 2,653 records, there were 5,123 unique terms with an average of 14 postings/term and a maximum of over 2,000 postings for a term.
- A larger data set of 38,304 records had dictionaries on the order of 250,000 lines (250,000 unique terms, including some numbers) and an average of 88 postings per record.
- From these numbers it is clear that efficient storage structures for both the binary search and the reading of the postings are critical.

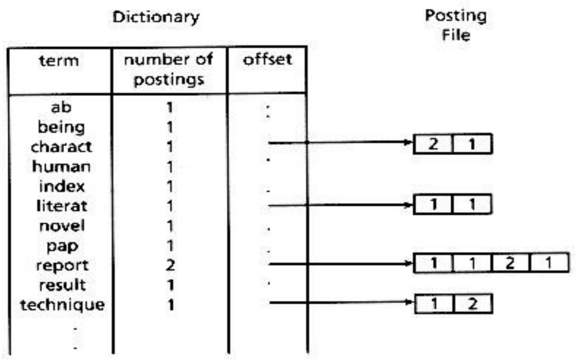


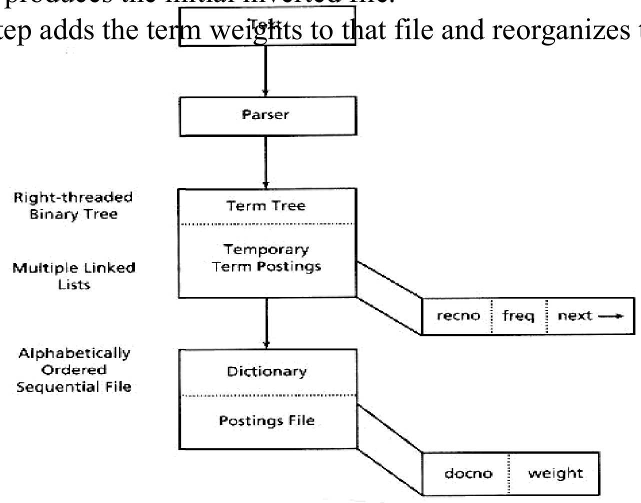
Fig: Dictionary and postings file from the last example

MODIFICATIONS TO THE BASIC TECHNIQUE:-

- Two diferent techniques are presented as improvements on the basic inverted file creation.
- The first technique is for working with very large data sets using secondary storage.
- The second technique uses multiple memory loads for inverting files.

1) **Producing an Inverted File for Large Data Sets without Sorting:**

- The new indexing method (Harman & Candela 1990) is a two-step process that does not need the middle sorting step.
- The first step produces the initial inverted file.
- The second step adds the term weights to that file and reorganizes the file for maximum efficiency .



- The creation of the initial inverted file avoids the use of an explicit sort by using a right-threaded binary tree (Knuth 1973).
  - The data contained in each binary tree node is the current number of term postings and the storage location of the postings list for that term.
- Each term is identified by the text parsing program
  - It is looked up in the binary tree, and either is added to the tree, along with related data, or causes tree data to be updated.

- The postings are stored as multiple linked lists, one variable length linked list for each term, with the lists stored in one large file.
- Each element in the linked postings file consists of a record number (the location of a given term), the term frequency in that record, and a pointer to the next element in the linked list for that given term.
- By storing the postings in a single file, no storage is wasted, and the files are easily accessed by following the links.

➢ As the location of both the head and tail of each linked list is stored in the binary tree, the entire list does not need to be read for each addition, but only once for use in creating the final postings file (step two).

➢ The binary tree and linked postings lists are saved for use by the term weighting routine (step two). This routine walks the binary tree and the linked postings list to create an alphabetical term list (dictionary) and a sequentially stored postings file.

➢ To do this, each term is consecutively read from the binary tree (this automatically puts the list in alphabetical order), along with its related data.
  - A new sequentially stored postings file is allocated, with two elements per posting.
  - The linked postings list is then traversed, with the frequencies being used to calculate the term weights (if desired).
  - The last step writes the record numbers and corresponding term weights to the newly created sequential postings file.
  - These sequentially stored postings files could not be created in step one because the number of postings is unknown at that point in processing, and input order is text order, not inverted file order. The final index files therefore consist of the same dictionary and sequential postings file as for the basic inverted file.

<u>Fast Inversion Algorithm(FAST-INV):</u>

The second technique to produce a sorted array inverted file is a fast inversion algorithm called FAST –INV.

This technique takes advantage of two principles:

➢ The large primary memories available on today's computers.

➢ The inherent order of the input data.

- The first principle is important since personal computers with more than 1 meg abyte of primary memory are common, and mainframes may have more than 100 megabytes of memory. Even if databases are on the order of 1 gigabyte, if they can be split into memory loads that can be rapidly processed and then combined, the overall cost will be minimized.

- The second principle is crucial since with large files it is very expensive to use polynomial or even n log n sorting algorithms. These costs are further compounded if memory is not used, since then the cost is for disk operations.

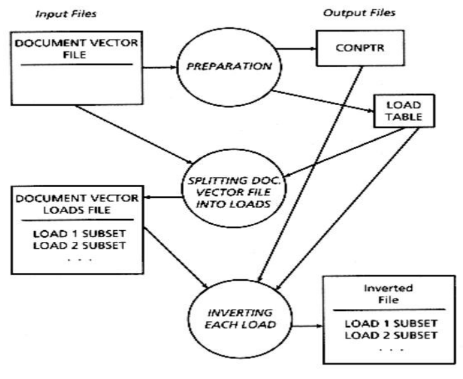


Fig: Overall scheme of FAST -INV

- The input to FAST -INV is a document vector file containing the concept vectors for each document in the collection to be indexed.
    - ✓ The document numbers appear in the left -hand column and the concept numbers of the words in each document appear in the right- hand column.
    - ✓ This is similar to the initial word list for the basic method, except that the words are represented by concept numbers, one concept number for each unique word in the collection (i.e., 250,000 unique words implies 250,000 unique concept numbers).
    - ✓ Note however that the document vector file is in sorted order, so that concept numbers are sorted within document numbers, and document numbers are sorted within collection. This is necessary for FAST-INV to work correctly.

Preparation:

HCN = highest concept number in dictionary
L = number of document/concept (or concept/document) pairs in the collection
M = available primary memory size, in bytes

- In the first pass, the entire document vector file can be read and two new files produced:
    - ✓ a file of concept postings/pointers (CONPTR)
    - ✓ and a load table.
- It is assumed that M >> HCN, so these two files can be built in primary memory. However, it is assumed that M < L, so several primary memory loads will be needed to process the document data. Because entries in the document vector file will already be grouped by concept number, with those concepts in ascending order, it is appropriate to see if this data can be somehow transformed beforehand into j parts such that:

- L / j < M, so that each part will fit into primary memory

- HCN / j concepts, approximately, are associated with each part

This allows each of the j parts to be read into primary memory, inverted there, and the output to be simply appended to the (partially built) final inverted file.

Specifically, preparation involves the following:

1. Allocate an array, con_entries_cnt, of size HCN, initialized to zero.
2. For each <doc#,con#> entry in the document vector file: increment con_entries_cnt[con#] .
3. Use the just constructed con_entries_cnt to create a disk version of CONPTR.
4. Initialize the load table.
5. For each <con#,count> pair obtained from con_entries_cnt: if there is no room for documents with this concept to fit in the current load, then create an entry in the load table and initialize the next load entry; otherwise update information for the current load table entry.

- After one pass through the input, the CONPTR file has been built and the load table needed in later steps of the algorithm has been constructed. Note that the test for room in a given load enforces the constraint that data for a load will fit into available memory. Specifically:
    - ▪ Let LL = length of current lo ad (i.e., number of concept/weight pairs)
    - ▪ S = spread of concept numbers in the current load (i.e., end concept - start concept + 1)
    - ▪ 8 bytes = space needed for each concept/weight pair
    - ▪ 4 bytes = space needed for each concept to store count of postings for it
- Then the constraint that must be met for another concept to be added to the current load is

$$\boxed{8 * LL + 4 * S < M}$$

Splitting document vector file:-

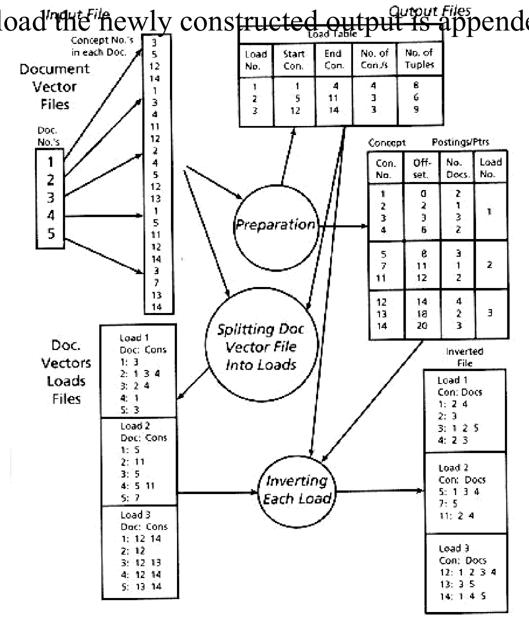
The load table indicates the range of concepts that should be processed for each primary memory load. There are two approaches to handling the multiplicity of loads.

- One approach, which is currently used, is to make a pass through the document vector file to obtain the input for each load. This has the advantage of not requiring additional storage space (though that can be obviated through the use of magnetic tapes), but has the disadvantage of requiring expensive disk I/O.
- The second approach is to build a new copy of the document vector collection, with the desired separation into loads. This can easily be done using the load table, since sizes of each load are known, in one pass through the input. As each document vector is read, it is separated into parts for each range of concepts in the load table, and those parts are appended to the end of the corresponding section of the output document collection file. With I/O buffering, the expense of this operation is proportional to the size of the files, and essentially costs the same as copying the file.

Inverting each load:-

- When a load is to be processed, the appropriate section of the CONPTR file is needed.
- An output array of size equal to the input document vector file subset is needed.
- As each document vector is processed, the offset (previously recorded in CONPTR) for a given concept is used to place the corresponding document/weight entry, and then that offset is incremented.
- Thus, the CONPTR data allows the input to be directly mapped to the output, without any sorting.
- At the end of the input load the newly constructed output is appended to the inverted file.

An example:


Input File
Output Files
Document Vector Files
Doc. No.'s
1
2
3
4
5
Concept No.'s in each Doc.
3 5 12 14 1 3 4 11 12 2 4 5 12 13 1 5 11 12 14 3 7 13 14
Preparation
Splitting Doc Vector File Into Loads
Inverting Each Load
Doc. Vectors Loads Files
Load 1 Doc: Cons 1: 3 2: 1 3 4 3: 2 4 4: 1 5: 3
Load 2 Doc: Cons 1: 5 2: 11 3: 5 4: 5 11 5: 7
Load 3 Doc: Cons 1: 12 14 2: 12 3: 12 13 4: 12 14 5: 13 14
Inverted File
Load 1 Con: Docs 1: 2 4 2: 3 3: 1 2 5 4: 2 3
Load 2 Con: Docs 5: 1 3 4 7: 5 11: 2 4
Load 3 Con: Docs 12: 1 2 3 4 13: 3 5 14: 1 4 5
Concept
Postings/Ptrs

Load Table

Load No.	Start Con.	End Con.	No. of Con.'s	No. of Tuples
1	1	4	4	8
2	5	11	3	6
3	12	14	3	9

Con. No.	Off-set.	No. Docs.	Load No.
1	0	2	1
2	2	1	
3	3	3	
4	6	2	
5	8	3	2
7	11	1	
11	12	2	
12	14	4	3
13	18	2	
14	20	3	

- The second phase of processing uses the load table to split the input document vector files and create the document vector loads files. There are three parts, corresponding to the three loads. It can be seen that the document vectors in each load are shortened since only those concepts in the allowable range for the load are included.
- The final phase of processing involves inverting each part of the document vectors loads file, using primary memory, and appending the result to the inverted file. The appropriate section of the CONPTR file is used so that inversion is simply a copying of data to the correct place, rather than a sort.

CHAPTER 3: SIGNATURE FILES

Abstract
This chapter presents a survey and discussion on signature-based text retrieval methods. It describes the main idea behind the signature approach and its advantages over other text retrieval methods, it provides a classification of the signature methods that have appeared in the literature, it describes the main representatives of each class, together with the relative advantages and drawbacks, and it gives a list of applications as well as commercial or university prototypes that use the signature approach.

3.1 INTRODUCTION

Text retrieval methods have attracted much interest recently. There are numerous applications involving storage and retrieval of
textual data:

- Electronic office filing.
- Computerized libraries.
- Library of Medicine.
- Automated law and patent offices. The U.S. Patent and Trademark Office has been examining electronic storage and retrieval of the recent patents on a system of 200 optical disks.
- Electronic storage and retrieval of articles from newspapers and magazines.
- Consumers' databases, which contain descriptions of products in natural language.
- Electronic encyclopedias.
- Indexing of software components to enhance reusability.
- Searching databases with descriptions of DNA molecules.
- Searching image databases, where the images are manually annotated. A similar approach could be used to search a database with animations, I scripted animation is used

The main operational characteristics of all the above applications are the following two:
1.Text databases are traditionally large.
2.Text databases have archival nature: there are insertions in them, but almost never deletions and updates.
A brief, qualitative comparison of the signature-based methods versus their competitors is as follows:
The signature-based methods are much faster than full text scanning (1 or 2 orders of magnitude faster, depending on the individual method). Compared to inversion, they require a modest space overhead (typically 10-15%, as opposed to 50-300% that inversion requires); moreover, they can
handle insertions more easily than inversion, because they need "append -only" operations -- no reorganization or rewriting of any portion of the signatures. Methods requiring "append only"
insertions have the following advantages: (a) increased concurrency during insertions (the readers may continue consulting the old portion of index structure, while an insertion takes place) (b) these methods work well on Write-Once-Read -Many (WORM) optical disks, which constitute an excellent archival medium.
On the other hand, signature files may be slow for large databases, precisely because their response time is linear on the number of items *N* in the database. Thus, signature files have been used in the following environments:

1. PC-based, medium size db
2. WORMs
3. parallel machines
4. distributed text db

3.2 BASIC CONCEPTS

Signature files typically use superimposed coding to create the signature of a document. A brief description of the method follows.
For performance reasons, which will be explained later, each document is divided into "logical blocks," that is, pieces of text that contain a constant number *D* of distinct, noncommon words. (To improve the space overhead, a stoplist of common words is maintained.) Each such word yields a "word signature," which is a bit pattern of size *F*, with *m* bits set to "1", while the rest are "0" (see Figure 3.1). *F* and *m* are design parameters. The word signatures are OR'ed together to form the block signature. Block signatures are concatenated, to form the document signature. The *m* bit positions to be set to "1" by each word are decided by hash functions. Searching for a word is handled by creating the signature of the word and by examining each block signature for "1" 's in those bit positions that the signature of the search word has a "1".

```
Word Signature
-----------------------------------
free 001 000 110 010
text 000 010 101 001
-----------------------------------
block signature 001 010 111 011
```

Figure 4.1: Illustration of the superimposed coding method. It is assumed that each logical block consists of D=2 words only. The signature size F is 12 bits, m=4 bits per word.
In order to allow searching for parts of words, the following method has been suggested: Each word is divided into successive, overlapping
triplets (e.g., "fr", "fre", "ree", "ee" for the word "free"). Each such triplet is hashed to a bit position by applying a hashing function on a numerical encoding of the triplet, for example, considering the triplet as a base-26 number. In the case of a word that has *l* triplets, with $l > m$, the word is allowed to set *l* (nondistinct) bits. If $l < m$, the additional bits are set using a random number generator, initialized with a numerical encoding of the word.
An important concept in signature files is the false drop probability *Fd*. Intuitively, it gives the probability that the signature test will fail, creating a "false alarm" (or "false hit" or "false drop"). Notice that the signature test never gives a false dismissal.
DEFINITION: False drop probability, *Fd*, is the probability that a block signature seems to qualify, *given that the block does not actually qualify*. Expressed mathematically:
Fd = Prob{signature qualifies/block does not}
The signature file is an *F N* binary matrix. Previous analysis showed that, for a given value of *F*, the value of *m* that minimizes the false drop probability is such that each row of the matrix contains "1" 's with probability 50 percent. Under such an optimal design, we have
Fd = 2-*m*
*F*1n2 = *mD*
This is the reason that documents have to be divided into logical blocks: Without logical blocks, a long document would have a signature full of "l" 's, and it would always create a false drop. To avoid unnecessary complications, for the rest of the discussion we assume that all the *documents span exactly one logical block*.
Table 3.1: Symbols and definitions

Symbol	Definition
F	signature size in bits
m	number of bits per word
D	number of distinct noncommon words per document
Fd	false drop probability

The most straightforward way to store the signature matrix is to store the rows sequentially. For the rest of this work, the above method will be called **SSF**, for Sequential Signature File.

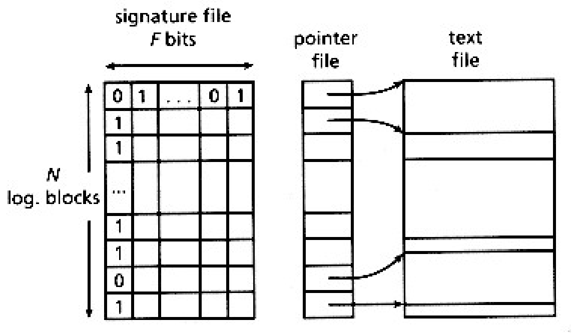


Figure 3.2: File structure for SSF

Although SSF has been used as is, it may be slow for large databases. Many methods have been suggested, trying to improve the response time of SSF, trading off space or insertion simplicity for speed. The main ideas behind all these methods are the following:

1. Compression: If the signature matrix is deliberately sparse, it can be compressed.
2. Vertical partitioning: Storing the signature matrix column wise improves the response time on the expense of insertion time.
3. Horizontal partitioning: Grouping similar signatures together and/or providing an index on the signature matrix may result in better-than-linear search.

Classification of the signature-based methods:

Sequential storage of the signature matrix
without compression
- sequential signature files (SSF)

with compression
- bit-block compression (BC)
- variable bit-block compression (VBC)

Vertical partitioning
without compression
- bit-sliced signature files (BSSF, B'SSF))
- frame sliced (FSSF)
- generalized frame- sliced (GFSSF)

with compression
- compressed bit slices (CBS)
- doubly compressed bit slices (DCBS)
- no-false-drop method (NFD)

Horizontal partitioning
data independent partitioning
- Gustafson's method
- partitioned signature files

data dependent partitioning
- 2-level signature files

S-trees

3.3 COMPRESSION

In this we create sparse document signatures on purpose, and then compress them before storing them sequentially. Analysis in that paper showed that, whenever compression is applied, the best value for *m* is 1. Also, it was shown that the resulting methods achieve better false drop probability than SSF for the same space overhead.
The idea in these methods is that we use a (large) bit vector of *B* bits and we hash each word into one (or perhaps more, say *n*) bit position(s), which are set to "1" (see Figure 3.4).
The resulting bit vector will be sparse and therefore it can be compressed.

```
data 0000 0000 0000 0010 0000
base 0000 0001 0000 0000 0000
management 0000 1000 0000 0000 0000
system 0000 0000 0000 0000 1000
----------------------------------------
block
signature 0000 1001 0000 0010 1000
```

Figure 3.4: Illustration of the compression-based methods. With B = 20 and n = 1 bit per word, the resulting bit vector is sparse and can be compressed.

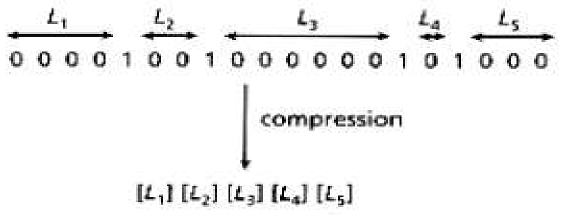


Figure 3.5: Compression using run-length encoding. The notation [x] stands for the encoded value of number x

3.3.1 Bit-block Compression (BC)

This method accelerates the search by sacrificing some space, compared to the run-length encoding technique. The compression method is based on bit-blocks, and was called BC (for bit-Block Compression). To speed up the searching, the sparse vector is divided into groups of consecutive bits (bit-blocks); each bit-block is encoded individual. For each bit-block we create a signature, which is of variable length and consists of at most three parts (see Figure 3.6):
Part I. It is one bit long and it indicates whether there are any "l"s in the bit-block (1) or the bit-block is empty (0). In the latter case, the bit-block signature stops here.
Part II. It indicates the number *s* of "1"s in the bit-block. It consists of *s* - 1 "1"s and a terminating zero. This is not the optimal way to record the number of "1"s. However this representation is simple and it seems to give results close to the optimal.
Part III. It contains the offsets of the "1"s from the beginning of the bit-block (1 *gb* bits for each "1", where *b* is the bit-block size).
b
<-->

```
sparse vector 0000 1001 0000 0010 1000
------------------------------------------
Part I 0 1 0 1 1
Part II 10 0 0
Part III 0011 10 00
```

Figure 3.6: Illustration of the BC method with bit-block size b = 4.

3.3.2 Variable Bit-block Compression (VBC)

The BC method was slightly modified to become insensitive to changes in the number of words *D* per block. This is desirable because the need to split messages in logical blocks is eliminated, thus simplifying the resolution of complex queries: There is no need to "remember" whether some of the terms of the query have appeared in one of the previous logical blocks of the message under inspection.

The idea is to use a different value for the bit-block size *bopt* for each message, according to the number *W* of bits set to "1" in the sparse vector. The size of the sparse vector *B* is the same for all messages. Figure 3.8 illustrates an example layout of the signatures in the VBC method. The upper row corresponds to a small message with small *W*, while the lower row to a message with large *W*. Thus, the upper row has a larger value of *bopt*, fewer bit-blocks, shorter Part I (the size of Part I is the number of bit-blocks), shorter Part II (its size is *W*) and fewer but larger offsets in Part III (the size of each offset is log *bopt* bits).

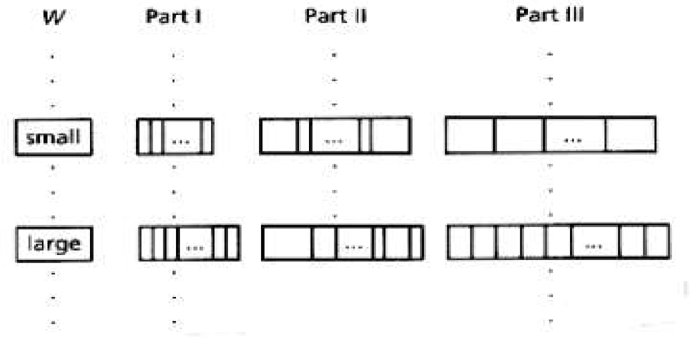


Figure 3.8: An example layout of the message signatures in the VBC method.

3.3.3 Performance

With respect to space overhead, the two methods (BC and VBC) require less space than SSF for the same false drop probability. Their response time is slightly less than SSF, due to the decreased I/0 requirements. The required main-memory operations are more complicated (decompression, etc.), but they are probably not the bottleneck. VBC achieves significant savings even on main -memory operations. With respect to insertions, the two methods are almost as easy as the SSF; they require a few additional CPU cycles to do the compression.

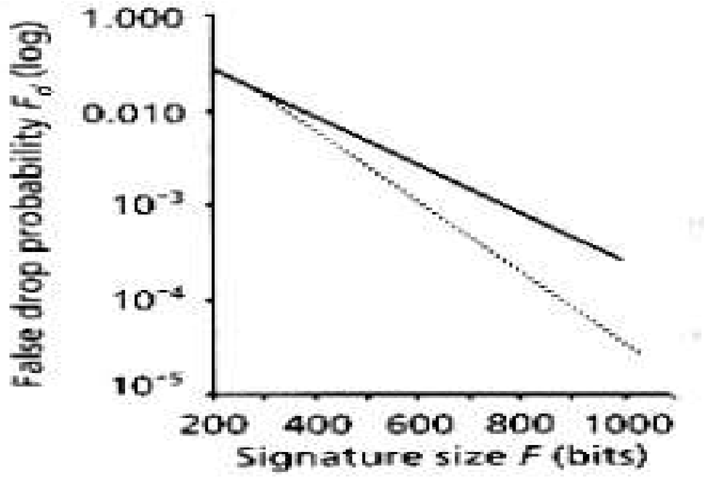


Figure 3.9: Comparison of Fd of BC (dotted line) against SSF (solid line), as a function of the space overhead Ov. Analytical results, from Faloutsos and Christodoulakis

(1987).

3.4 VERTICAL PARTITIONING

The idea behind the vertical partitioning is to avoid bringing useless portions of the document signature in main memory; this can be achieved by storing the signature file in a bit-sliced form or in a "frame-sliced" form.

3.4.1 Bit-Sliced Signature Files (BSSF)

The bit-sliced design is illustrated in Figure 3.10.

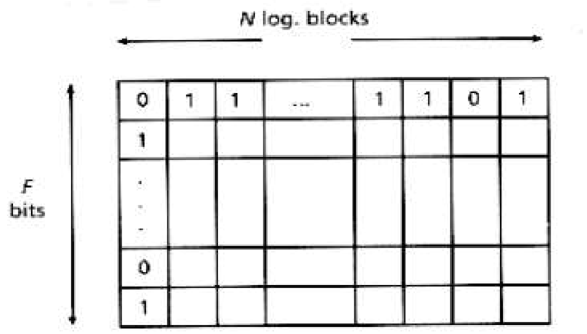


Figure 3.10: Transposed bit matrix

To allow insertions, we propose using *F* different files, one per each bit position, which will be referred to as "bit-files." The method will be called **BSSF**, for "Bit-Sliced Signature Files." Figure 3.11 illustrates the proposed file structure.
Searching for a single word requires the retrieval of *m* bit vectors (instead of all of the *F* bit vectors) which are subsequently ANDed together. The resulting bit vector has *N* bits, with "1" 's at the positions of the qualifying logical blocks. An insertion of a new logical block requires *F* disk accesses, one for each bit-file, but **no rewriting!**

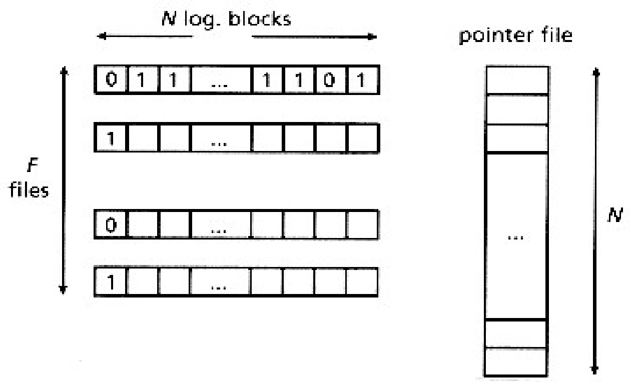


Figure 4.11: File structure for Bit-Sliced Signature Files. The text file is omitted.

3.4.2 B'SSF, a Faster Version of BSSF

The traditional design of BSSF suggests choosing the optimal value for *m* to be such that the document signatures contain "l"s by 50 percent. A typical value of *m* is of the order of 10. This implies 10 random disk accesses on a single word query. It is suggested to use a smaller than optimal value of *m*; thus, the number of random disk accesses decreases. The drawback is that, in order to maintain the same false drop probability, the document signatures have to be longer.

3.4.3 Frame-Sliced Signature File

The idea behind this method is to force each word to hash into bit positions that are close to each other in the document signature. Then, these bit files are stored together and can be retrieved with few random disk accesses. The main motivation for this organization is that random disk accesses are more expensive than sequential ones, since they involve movement of the disk arm.
More specifically, the method works as follows: The document signature (*F* bits long) is divided into *k* frames of *s* consecutive bits each. For each word in the document, one of the *k* frames will be chosen by a hash function; using another hash function, the word sets *m* bits (not necessarily distinct) in that frame. *F, k, s, m* are design parameters. Figure 3.12 gives an example for this method.

```
Word Signature
free 000000 110010
text 010110 000000
doc. signature 010110 110010
```

Figure 3.12: D = 2 words. F = 12, s = 6, k = 2, m = 3. The word free is hashed into the second frame and sets 3 bits there. The word text is hashed into the first frame and also sets 3 bits there.
The signature matrix is stored framewise. Each frame will be kept in consecutive disk blocks. Only one frame has to be retrieved for a single word query, that is, only one random disk access is required. At most, *n* frames have to be scanned for an *n* word query. Insertion will be much faster than BSSF since we need only access *k* frames instead of *F* bit-slices. This method will be referred to as Frame-Sliced Signature file **(FSSF)**.

3.4.4 The Generalized Frame-Sliced Signature File(GFSSF)

In FSSF, each word selects only one frame and sets *m* bit positions in that frame. A more general approach is to select *n* distinct frames and set *m* bits (not necessarily distinct) in each frame to generate the word signature. The document signature is the OR-ing of all the word signatures of all the words in that document. This method is called Generalized Frame-Sliced Signature File.
Notice that BSSF, B'SSF, FSSF, and SSF are actually special cases of GFSSF:
*When $k = F$, $n = m$, it reduces to the BSSF or B'SSF method.
*When $n = 1$, it reduces to the FSSF method.
*When $k = 1$, $n = 1$, it becomes the SSF method (the document signature is broken down to one frame only).

3.4.5 Performance

Since GFSSF is a generalized model, we expect that a careful choice of the parameters will give a method that is better (whatever the criterion is) than any of its special cases. Analysis in the above paper gives formulas for the false drop probability and the expected response time for GFSSF and the rest of the methods. Figure 3.13 plots the theoretically expected performance of GFSSF, BSSF, B'SSF, and FSSF. Notice that GFSSF is faster than BSSF, B'SSF, and FSSF, which are all its special cases. It is assumed that the transfer time for a page *Ttrans* = 1 msec
and the combined seek and latency time *Tseek* is *Tseek* = 40 msec

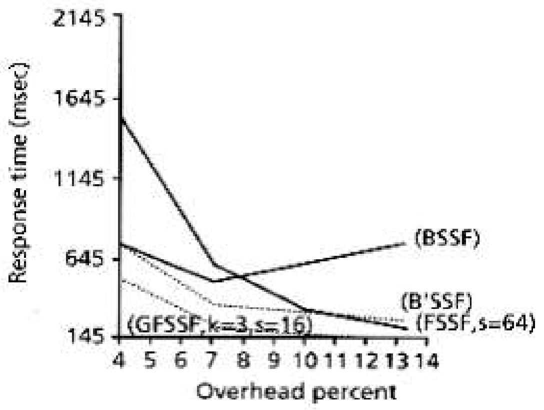


Figure 3.13: Response time vs. space overhead: a comparison between BSSF, B'SSF, FSSF and GFSSF. Analytical results on a 2.8Mb database.

3.5 VERTICAL PARTITIONING AND COMPRESSION

The idea in all the methods in this class is to create a very sparse signature matrix, to store it in a bit-sliced form, and compress each bit slice by storing the position of the "1"s in the slice. The methods in this class are closely related to inversion with a hash table.

3.5.1 Compressed Bit Slices (CBS)

Although the bit-sliced method is much faster than SSF on retrieval, there may be room for two improvements:

1. On searching, each search word requires the retrieval of *m* bit files, exactly because each word signature has *m* bits set to "1". The search time could be improved if *m* was forced to be "1".

2. The insertion of a logical block requires too many disk accesses (namely, *F*, which is typically, 600-1,000).

If we force *m* = l, then *F* has to be increased, in order to maintain the same false drop probability. For the next three methods, we shall use *S* to denote the size of a signature, to highlight the similarity of these methods to inversion using hash tables. The corresponding bit matrix and bit files will be sparse and they can be compressed.

Figure 3.14 illustrate a sparse bit matrix. The easiest way to compress each bit file is to store the positions of the "1" 's. However, the size of each bit file is unpredictable now, subject to statistical variations. Therefore, we store them in buckets of size *Bp*, which is a design parameter. As a bit file grows, more buckets are allocated to it on demand. These buckets are linked together with pointers. Obviously, we also need a directory (hash table) with *S* pointers, one for each bit slice. Notice the following:

1. There is no need to split documents into logical blocks any more.

2. The pointer file can be eliminated. Instead of storing the position of each "1" in a (compressed) bit file, we can store a pointer to the document in the text file.

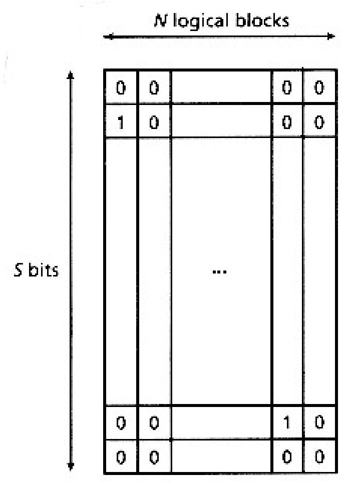


Figure 3.14: Sparse bit matrix

Thus, the compressed bit files will contain pointers to the appropriate documents (or logical blocks). The set of all the compressed bit files will be called "leve1 1" or "postings file," to agree with the terminology of inverted files.
The postings file consists of postings buckets, of size *B*p bytes (*B*p is a design parameter). Each such bucket contains pointers to the documents in the text file, as well as an extra pointer, to point to an overflow postings bucket, if necessary.
Figure 3.15 illustrates the proposed file structure.

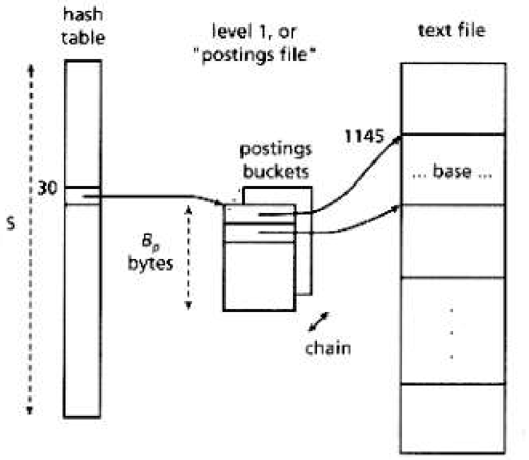


Figure 3.15: Illustration of CBS
Searching is done by hashing a word to obtain the postings bucket address. This bucket, as well as its overflow buckets, will be retrieved, to obtain the pointers to the relevant documents. To reduce the false drops, the hash table should be sparse. The method is similar to hashing. The differences are the following:
a. The directory (hash table) is sparse; Traditional hashing schemes require loads of 80-90 percent.
b. The actual word is stored nowhere. Since the hash table is sparse, there will be few collisions. Thus, we save space and maintain a simple file structure.

3.5.2 Doubly Compressed Bit Slices (DCBS)

The motivation behind this method is to try to compress the sparse directory of CBS. The file structure we propose consists of a hash table, an intermediate file, a postings file and the text file as in Figure 3.16. The method is similar to CBS. It uses a hashing function *h*1(), which returns values in the range (O,(*S*-1)) and determines the slot in the directory. The difference is that DCBS makes an effort to distinguish among synonyms, by using a second hashing function *h*2(), which returns bit strings that are *h* bits long. These hash codes are stored in the "intermediate file," which consists of buckets of B*i* bytes (design parameter). Each such bucket contains records of the form (*hashcode, ptr*). The pointer *ptr* is the head of a linked list of postings buckets.

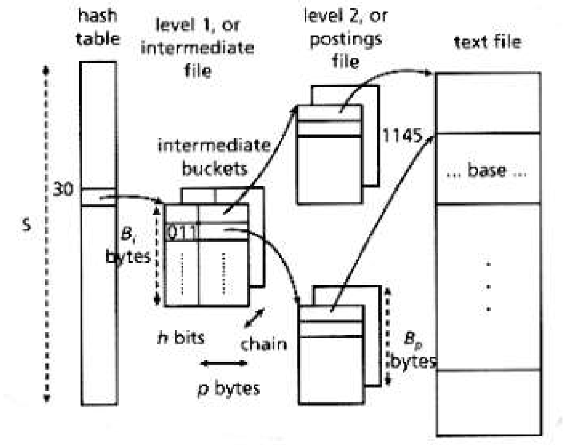


Figure 3.16: Illustration of DCBS

Figure 3.16 illustrates an example, where the word "base" appears in the document that starts at the 1145-th byte of the text file. The example also assumes that *h* = 3 bits, *h*l("base") = 30, and *h*2("base") = (011)2.

Searching for the word "base" is handled as follows:

Step 1 *h*1("base") = 30: The 30-th pointer of the directory will be followed. The corresponding chain of intermediate buckets will be examined.

Step 2 *h*2("base") = (011)2: The records in the above intermediate buckets will be examined. If a matching hash code is found (at most one will exist!), the corresponding pointer is followed, to retrieve the chain of postings buckets.

Step 3 The pointers of the above postings buckets will be followed, to retrieve the qualifying (actually or falsely) documents.

Insertion is omitted for brevity.

3.5.3 No False Drops Method (NFD)

This method avoids false drops completely, without storing the actual words in the index. The idea is to modify the intermediate file of the DCBS, and store a pointer to the word in the text file. Specifically, each record of the intermediate file will have the format (*hashcode, ptr, ptr-to -word*), where *ptr -to -word* is a pointer to the word in the text file. See Figure 3.17 for an illustration.

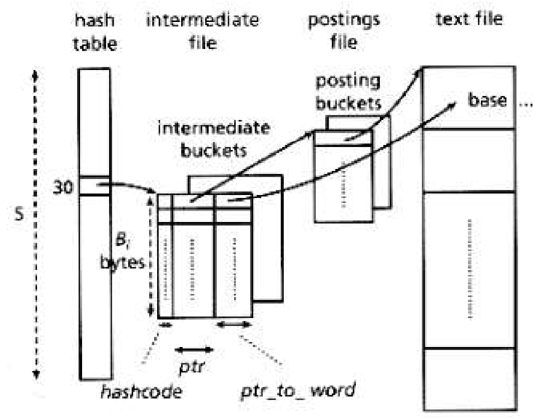


Figure 3.17: Illustration of NFD

This way each word can be completely distinguished from its synonyms, using only *h* bits for the hash code and *p* (=4 bytes, usually) for the *ptr-to -word* . The advantages of storing *ptr-to -word* instead of storing the actual word are two: (1) space is saved (a word from the dictionary is 8 characters long (Peterson 1980), and (2) the records of the intermediate file have fixed length. Thus, there is no need for a word delimiter and there is no danger for a word to cross bucket boundaries.
Searching is done in a similar way with DCBS. The only difference is that, whenever a matching hash code is found in Step 2, the corresponding *ptr-to -word* is followed, to avoid synonyms completely.

3.5.4 Performance

In Faloutsos and Chan (1988), an analytical model is developed for the performance of each of the above methods. Experiments on the 2.8Mb database showed that the model is accurate. Figure 3.18 plots the theoretical performance of the methods (search time as a function of the overhead).
The final conclusion is that these methods are fast, requiring few disk accesses, they introduce 20-25 percent space overhead, and they still require append-only operations on insertion.

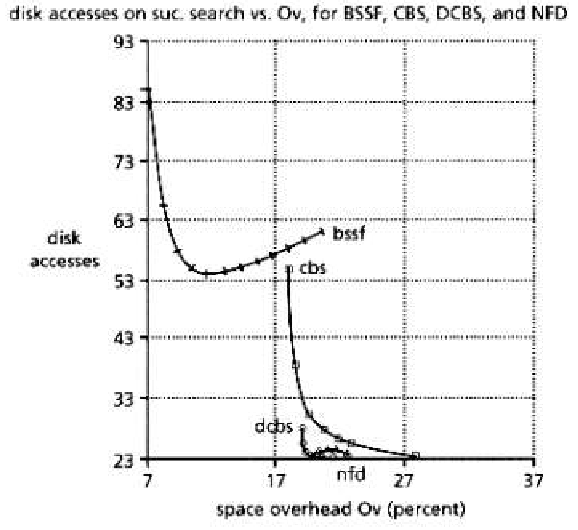


Figure 3.18: Total disk accesses on successful search versus space overhead.

Analytical results for the 2.8 Mb database, with p = 3 bytes per pointer. Squares correspond to the CBS method, circles to DCBS, and triangles to NFD.

3.6 HORIZONTAL PARTITIONING

The motivation behind all these methods is to avoid the sequential scanning of the signature file (or its bit-slices), in order to achieve better than $O(N)$ search time. Thus, they group the signatures into sets, partitioning the signature matrix horizontally. The grouping criterion can be decided beforehand, in the form of a hashing function $h(S)$, where S is a document signature (data independent case). Alternatively, the groups can be determined on the fly, using a hierarchical structure (e.g. a B-tree--data dependent case).

3.6.1 Data Independent Case

Gustafson's method
The earliest approach was proposed by Gustafson (1971). Suppose that we have records with, say six attributes each. For example, records can be documents and attributes can be keywords describing the document. Consider a hashing function h that hashes a keyword w to a number $h(w)$ in the range 0-15. The signature of a keyword is a string of 16 bits, all of which are zero except for the bit at position $h(w)$. The record signature is created by superimposing the corresponding keyword signatures. If $k < 6$ bits are set in a record signature, additional $6 - k$ bits are set by some random method. Thus, there are *comb* (16,6) = 8,008 possible distinct record signatures (where $C(m,n)$ denotes the combinations of m choose n items). Using a hash table with 8,008 slots, we can map each record signature to one such slot as follows: Let $p1 < p2 < \ldots < p6$ the positions where the "1"s occur in the record signature. Then the function $C(p1, 1) + C(p2, 2) + \ldots + C(p6, 6)$ maps each distinct record signature to a number in the range 0-8,007. The interesting point of the method is that the extent of the search decreases quickly (almost exponentially) with the number of terms in the (conjunctive) query. Single word queries touch $C(15,5) = 3,003$ slots of the hash table, two-word queries touch $C(14, 4) = 1,001$ slots, and so on.
Although elegant, Gustafson's method suffers from some practical problems:
1. Its performance deteriorates as the file grows.
2. If the number of keywords per document is large, then either we must have a huge hash table or usual queries (involving 3-4 keywords) will touch a large portion of the database.
3. Queries other than conjunctive ones are handled with difficulty.
Partitioned signature files

3.6.2 Data Dependent Case

Two-level signature files
Sacks-Davis and his colleagues (1983, 1987) suggested using two levels of signatures. Their documents are bibliographic records of variable length. The first level of signatures consists of document signatures that are stored sequentially, as in the SSF method. The second level consists of "block signatures"; each such signature corresponds to one block (group) of bibliographic records, and is created by superimposing the signatures of all the words in this block, ignoring the record boundaries. The second level is stored in a bit sliced form. Each level has its own hashing functions that map words to bit positions.
Searching is performed by scanning the block signatures first, and then concentrating on these portions of the first-level signature file that seem promising.

S-tree

Deppisch (1986) proposed a B-tree like structure to facilitate fast access to the records (which are signatures) in a signature file. The leaf of an S-tree consists of *k* "similar" (i.e. ,with small Hamming distance) document signatures along with the document identifiers. The OR-ing or these *k* document signatures forms the "key" of an entry in an upper level node, which serves as a directory for the leaves. Recursively we construct directories on lower level directories until we reach the root. The S-tree is kept balanced in a similar manner as a B-trees: when a leaf node overflows it is split in two groups of "similar" signatures; the father node is changed appropriately to reflect the new situation. Splits may propagate upward until reaching the root.

The method requires small space overhead; the response time on queries is difficult to estimate analytically. The insertion requires a few disk accesses (proportional to the height of the tree at worst), but the append-only property is lost. Another problem is that higher level nodes may contain keys that have many 1's and thus become useless.