Inverted indexes: Types and techniques

Ajit Kumar Mahapatra[1], Sitanath Biswas[2]

[1] Information Technology, ITER, Siksha 'O' Anusandhan University, Bhubaneswar, Orissa, 751030, India

[2] Information Technologies, ITER, Siksha 'O' Anusandhan University, Bhubaneswar, Orissa, 751030, India

Abstract

There has been a substantial amount of research on high performance inverted index because most web and search engines use an inverted index to execute queries. Documents are normally stored as lists of words, but inverted indexes invert this by storing for each word the list of documents that the word appears in, hence the name "inverted index". This paper presents the crucial research findings on inverted indexes, their types and techniques.

1. Introduction

Most web and intranet search engines uses an inverted text index to execute text queries. Because inverted indexes are expensive to update, search engines typically reconstruct their index from scratch on a periodic basis. The more frequently an index can be reconstructed, the faster update will be reflected in search results, which in turns improves search quality. There has been a substantial amount of research on high performance inverted index because most web and search engines use an inverted index to execute queries. Documents are normally stored as lists of words, but inverted indexes invert this by storing for each word the list of documents that the word appears in, hence the name "inverted index".

2. Inverted indexes

Documents are normally stored as lists of words, but inverted indexes invert this by storing for each word the list of documents that the word appears in, hence the name "Inverted index". There are several variations on inverted indexes. At a minimum, you need to store for each word the list of documents that the word appears in. If you want to support phrase and proximity queries you need to store word positions for each document, i.e. the positions that the word appears in. The granularity of a position can range from byte offset to word to paragraph to section, but usually it is stored at word position granularity. You can also store just the word frequency for each document instead of word positions.

Storing the total frequency for each word can be useful in optimizing query execution plans. Some implementations store two inverted lists, one storing just the document lists (and usually the word frequencies) and one storing the full word position lists. Simple queries can then be answered consulting just the much shorter document lists. Some implementations go even further and store meta-information about each "hit", i.e. word position. They typically use a byte or two for each hit that has bits for things like font size, text type (title, header, anchor (HTML), plain text, etc.) This information can then be used for better ranking of search results as words that have special formatting are usually more important.

Another possible variation is whether the lexicon is stored separately or not. The lexicon stores all the tokens indexed for the whole collection. Usually it also stores statistical information for each token like the number of documents it appears in. The lexicon can be helpful in various ways that we refer to later on.

The space used by the inverted index varies somewhere in the range of 5-100% of the total size of the documents indexed. This enormous range exists because inverted index implementations come in so many different variations. Some store word positions, some do not, some do aggressive document preprocessing to cut down the size of the index, some do not, some support dynamic updates (they cause fragmentation and usually one must reserve extra space for future updates), some do not, some use more powerful (and slower) compression methods than others, and so on.ble 3 contains three examples of inverted indexes for the document collection from table 2. No stop words or stemming are used in this example. The indexes are:

Table 1: Sample document collection

Id	Contents
1	The only way not to think about money is to have a great deal of it.
2	When I was young I thought that money was the most important thing in life; now that I am ld I know that it is.
3	A man is usually more careful of his money than he is of his principles.

List 1 Just the document lists. The format is (d1, d2 . . .), where dn is the document id number.

List 2 Document lists with word frequencies. The format is (d1:f1, d2:f2 . . .), where dn is the document id number and fn is the word frequency.

List 3 Document lists and word positions with word granularity. The format is (d1 :(w1, w2 . . .), (d2 :(w1, w2 . . .) . . .) , where dn is the document id number and wn are the word positions.

Table 3 is also a good example of how time-consuming manual construction of inverted indexes is. It took over half an hour to create the lists by hand, but that pales when compared to Mary Cowden Clarke, who in 1845 pu blished a concordance (an archaic term for an inverted index) of Shakespeare's works that had taken her 16 years to create

2.1 Compression

Storing inverted lists totally uncompressed wastes huge amounts of space. Using the word "is" in table 3 as an example, if we stored the numbers as fixed-width 32-bit integers, list 1 would take 12 bytes, list 2 would take 26 bytes (using a special marker byte to mark ends of word position lists), and list 3 would take 30 bytes.

Table 2: Inverted list example

Word	List1	List2	List3
a	1,3	1:1.3:1	1: (12) ,3: (1)
About	1	1:1	1: (7)
am	2	2:1	2: (19)
Careful	3	3:1	3: (6)
deal	1	1:1	1: (16)
great	1	1:1	1: (13)
have	1	1:1	1: (11)
he	3	3:1	3: (11)
his	3	3:2	3: (8,14)
i	2	2:4	2: (2,5,18,21)
important	2	2:1	2: (12)
in	2	2:1	2: (14)
is	1,2,3	1:1,2:1,3:2	1:(: (9),2: (25),3: (3,12)
it	1,2	1:1,2:1	1:(16) ,2: (25)
know	2	2:1	2: (22)
life	2	2:1	2: (25)
man	3	3:1	3: (2)
money	1,2,3	1:1,2:1,3:1	1: (8), 2:(8),3:(9)
more	3	3:1	3: (5)
most	2	2:1	2: (11)
not	1	1:1	1: (4)
now	2	2:1	2: (16)
of	1,3	1:1 , 3:2	1: (15),3: (7,13)
old	2	2:1	2: (20)
only	1	1:1	1: (2)
principles	3	3:1	3: (15)
than	3	3:1	3: (10)
that	2	2:2	2: (7,23)
the	1,2	1:1,2:1	1: (1),2: (10)
thing	2	2:1	2: (13)
think	1	1:1	1: (6)
thought	2	2:1	2: (6)
to	1	1:2	1: (5,10)
usually	3	3:1	3:4
was	2	2:1	2: (9)
way	1	1:1	1: (3)
when	2	2:1	2: (1)

There are many ways to store the lists in a more compact form. They can be divided into two categories depending on whether the number of bits they use for coding a single value is always a multiple of 8 or not. The non-byte-aligned methods are slightly more compact, but more complex, harder to handle if dynamic updates are needed, and much slower to encode/decode. In practice, simple byte-aligned methods are the preferred choice in most cases.

Variable length integers Instead of using 32 bits to store every value, we can use variable length integers that only use as many bytes as needed. There are many variations on these, but a simple and often used variation marks the final byte of the value by setting the high bit (0x80) to 1. The lower 7 bits of each byte are concatenated to form the value.

Elias gamma Elias gamma coding [Eli75] consists of the number written in binary, prefixed by N zeros, where N = number of bits in the binary representation − 1. This is efficient for small values, for example 1 is coded as 1, 2 as 010, and 3 as 011, but inefficient for bigger values, for example 64396 is coded as 00000000000000001111101110001100, which is 31 bits. Elias delta Elias delta coding [Eli75] consists of

separating the number into the highest power of 2 it contains (2N) and the remaining N binary digits of the number, encoding N+1 with Elias gamma coding, and appending the remaining N binary digits. This is slightly more inefficient than Elias gamma coding for very small values, but much more efficient for large numbers. For example, 1 is coded as 1, 2 as 010|0, 3 as 010|1, and 64396 as 000010000|111101110001100, which is 24 bits. The character | in the examples is used to mark the boundary between the two parts of the coded value.

Golomb-Rice Golomb coding [Gol66] differs from Elias codes in that it is a parameterized one. The parameter b changes how the values are coded, and must be chosen according to the distribution of values to be coded, either real or expected. If b is a power of two, the coding is known as Golomb-Rice coding, and is the one usually used, since shift operations can then be used instead of divides and multiplies.The number to be coded is divided into two parts: the result of a division by b, and the remainder. The quotient is stored first, in unary coding, followed by the remainder, in truncated binary encoding. Using a value of 4 for b, 1 is coded as 101, 2 as 110, 3 as 111, and 4 as 0100. Coding the number 64396 with b = 4 would take over 16100 bits, so using a roughly correct value for b is of critical importance.

Delta coding We can increase our compression ratios for the lists of numbers significantly if we store them delta coded. This means that instead of storing absolute values, we store the difference to the previous value in the list. Since we can sort the lists before storing them, and there are no duplicate values, the difference between consecutive values is always , 1.The smaller the values are that we store, the more efficient the compression methods described above are. It takes less space to store (1, 12, 5, 2, 3, 15, and 4) than (1, 13, 18, 20, 23, 38, and 42). In table 3, the word “is” has the following for list 3: “1 :(9), 2 : (25), 3 : (3, 12)”. Applying delta coding would produce the following list: “1 :(9), 1 :(25), 1 :(3, 9)”.

```
def memoryInvert(documents):
    index = {}
      for d in documents:
        for t in tokenize(d):
          if t.word not in index:
          index[t.word] = CompressedList()
          index[t.word].add(t)
return index
```

Figure 1: In-memory inversion

2.1.2 Construction

Constructing an inverted index is easy. Doing it without using obscene amounts of memory, disk space or CPU time is a much harder task. Advances in computer hardware do not help much as the size of the collections being indexed is growing at an even faster rate. If the collection is small enough; doing the inversion process completely in memory is the fastest and easiest way. The basic mechanism is expressed in Python pseudo code in Figure 1.

In-memory inversion is not feasible for large collections so in those cases we have to store temporary results to disk. Since disk seeks are expensive, the best way to do that is to construct in-memory inversions of limited size, store them to disk, and then merge them to produce the final inverted index.

Moffat and Bell describe such a method .To avoid using twice the disk space of the final result they use an in-place multi-way merge sort to merge the temporary blocks. After the sort is complete the index file is still not quite finished, since due to the in-place aspect of the sort the blocks are not in their final order, and need to be permuted to their correct order. Heinz and Zobel present a modified version of the above algorithm that is slightly more efficient, mainly because it does not require keeping the lexicon in memory permanently during the inversion process and also due to their careful choice of data structures used in the implementation. Keeping the lexicon in memory permanently during the inversion process is a problem for very large collections, because as the size of the lexicon grows, the memory available for storing the position lists decreases.

3. Inverted index techniques

In this section we describe the techniques needed to implement a search engine using an inverted index. At the end of the section we describe some of the remaining unsolved problems.

3.1 Document preprocessing

Documents are normally not indexed as-is, but are preprocessed first. They are converted to tokens in the lexing phase, the tokens are possibly transformed into more generic ones in the stemming phase, and finally some tokens may be dropped entirely in the stop word removal phase. The following sections describe these operations.

3.1.1 Lexing

Lexing refers to the process of converting a document from a list of characters to a list of tokens, each of which is a single alphanumeric word. Usually there is a maximum length for a single token, typically something like 32 characters, to avoid unbounded index size growth in atypical cases. To generate these tokens from the input character stream, first case-folding is done, i.e. the input is converted to lowercase. Then, each collection of alphanumeric characters separated by non-alphanumeric characters (whitespace, punctuation, etc.) is added to the list tokens. Tokens containing too many numerical characters are usually pruned from the list since they increase the size of the index without offering much in return. The above only works for alphabetic languages. Ideographic languages (Chinese, Japanese, and Korean) do not have words composed of characters and need specialized search technologies.

3.1.2 Stemming

Stemming means not indexing each word as it appears after lexing, but transforming it to its morphological root (stem) and indexing that instead. For example, the words “compute”, “computer”, “computation”, “computers”, “computed” and “computing” might all be indexed as “compute”.

The most common stemming algorithm used for the English language is Porter’s. All stemming algorithms are complex, full of exceptions and exceptions to the exceptions, and still do a lot of mistakes, i.e., they fail to unite words that should be united or unite words that should not be united. They also reduce the accuracy of queries, especially phrase queries. In the old days, stemming was possibly useful since it decreased the size of the index and increased the result set for queries, but today the biggest problems search engines have are too many results returned by queries and ranking the results so that the most relevant ones are shown first, both of which are hindered by stemming. For this reason, many search engines (Google, for example) do not do stemming at all. This trend will probably increase in the future, as stemming can be emulated quite easily by wildcard queries or by query expansion.

3.1.3 Stop words

Stop words are words like “a”, “the”, “of”, and “to”, which are so common that nearly every document contains them. A stop word list contains the list of words to ignore when indexing the document collection. For normal queries, this usually does not worsen the results, and it saves some space in the index, but in some special cases like searching for “The Who” or “to be or not to be” using stop words can completely disable the ability to find the desired information. Since stop words are so common the differences between consecutive values in both document number and word position lists for them are smaller than for normal words, and thus the lists compress better. Because of this, the overhead for indexing all words is not as big as one might think. Like with stemming, modern search engines like Google do not seem to use stop words, since doing so would put them at a competitive disadvantage. A slightly bigger index is a small price to pay for being able to search for any possible combination of words.

3.2 Query types

There are many different ways of searching for information. Here we describe the most prominent ones and how they can be implemented using an inverted index as the base structure. Sample queries are formatted in bold type.

3.2.1 Normal

A normal query is any query that is not explicitly indicated by the user to be a specialized query of one of the types described later in this section. For queries containing only a single term, the desired semantics are clear: match all documents that contain the term. For multi-word queries, however, the desired semantics are not so clear. Some implementations treat it as an implicit Boolean query (see the next section for details on Boolean queries) by inserting hidden AND operators between each search term. This has the problem that if a user enters many search terms, for example 10, then a document that only contains 9 of them will not be included in the result set even though the probability of it being relevant is high. For this reason, some implementations choose another strategy: instead of requiring all search terms to appear in a document, they allow some of the terms to be missing, and then rank the results by how many of the search terms were found in each document. This works quite well, since a user can specify as many search terms as he wants without fear of eliminating relevant matches. Of course it is also much more expensive to evaluate than the AND version, which is probably the reason Most Internet search engines do not seem to use it.

3.2.2 Boolean

Boolean queries are queries where the search terms are connected to each other using the various operators available in Boolean logic [Boo54], most common ones being AND, OR and NOT. Usually parentheses can be used to group search terms. A simple example is madonna AND discography, and a more complex one is bruce AND mclaren AND NOT ("formula one" OR "formula 1" OR f1). These are implemented using an inverted index as follows:

NOT A pure NOT is usually not supported in Full-Text Search (FTS) implementations since it can match almost all of the documents. Instead it must be combined with other search terms using the AND operator, and after those are processed and the preliminary result set is available, that set is then further pruned by eliminating all documents from it that contain the NOT term. This is done by retrieving the document list for the NOT term and removing all document ids in it from the result set.

OR The query term1 OR term2 OR . . . term n is processed by retrieving the document lists for all of the terms and combining them by a union operation, i.e., a document id is in the final result set if it is found in at least one of the lists.

AND The query term1 AND term2 AND . . . term n is processed by retrieving the document lists for all of the terms and combining them by an intersection operation, i.e., a document id is in the final result set if it is found in all of the lists.

Unlike the OR operation which potentially expands the result set for each additional term, the AND operation shrinks the result set for each additional term. This allows AND operations to be implemented more efficiently. If we know or can guess which search term is the least common one, retrieving the document list for that term first saves memory and time since we are not storing in memory longer lists than are needed. The second document list to retrieve should be the one for the second least common term, etc.

If we have a lexicon available, a good strategy is to sort the search terms by each term's document count found in the lexicon, with the term with the smallest document count being first, and then doing the document list retrievals in that order. As an example, consider the query cat AND toxoplasmosis done on a well known Internet search engine. If we processed cat first, we would have to store a temporary list containing 136 million document ids. If we process toxoplasmosis first, we only have to store a temporary list containing 2 million document ids. In both cases the temporary list is then pruned to contain only 200,000 document ids when the lists for the terms are combined. Another way to optimize AND operations is by not constructing any temporary lists. Instead of retrieving the document lists for each term sequentially, they are all retrieved in parallel, and instead of retrieving the whole lists, they are read from the disk in relatively small pieces. These pieces are then processed in parallel from each list and the final result set is constructed. Which one of the above optimizations is used depends on other implementation decisions in the FTS system. Usually the latter one is faster, however.

3.2.3 Phrase

Phrase queries are used to find documents that contain the given words in the given order. Usually phrase search is indicated by surrounding the sentence fragment in quotes in the query string. They are most useful for finding documents with common words used in a very specific way. For example, if you do not remember the author of some quotation, searching for it on the Internet as a phrase query will in all likelihood find it for you. An example would be "there are few sorrows however poignant in which a good income is of no avail".

The implementation of phrase queries is an extension of Boolean AND queries, with most of the same optimizations applying, e.g., it is best to start with the least common word. Phrase queries are more expensive though, because in addition to the document lists they also have to keep track of the word positions in each document that could possibly be the start position of the search phrase. For example, consider the query "big deal". The lexicon is consulted and it is determined that "deal" is the rarer word of the two, so it is retrieved first. It occurs in document 5 at positions 1, 46 and 182, and in document 6 at position 74. We transform these so that the word positions point to the first search term, giving us 5(0, 45, 181) and 6(73). Since position 0 is before the start of the document, we can drop that one as it cannot exist.

Next we retrieve the document lists for the word "big" and prune our result set so it only contains words where "big" occurs in the right place. If "big" occurs in document 5 at positions 33 and 45 and in document 53 at position 943, the final result set is "document 5, word position 45". Since the above is more expensive than normal searches, there have

been efforts to investigate the use of auxiliary indexes for phrase searches. For example, Bahle, Williams and Zobel propose using a "next word index", which indexes selected two-word sentence fragments. They claim that it achieves significant speedups with only a modest disk space overhead.

3.2.4 Proximity

Proximity queries are of the form term1 NEAR(n) term2, and should match documents where term1 occurs within n words of term2. They are useful in many cases, for example when searching for a person's name you never know whether a name is listed as "Osku Salerma" or "Salerma, Osku", so you might use the search osku NEAR(1) salerma to find both cases. Queries where n is 1 could also be done as a combination of Boolean and phrase queries ("osku salerma" OR "salerma osku"), but for larger n, proximity queries cannot be emulated with other query types. An example of such a query is apache NEAR (5) "performance tuning".

Proximity queries are implemented in the same way as phrase queries, the only difference being that instead of checking for exact relative word positions of the search terms, the positions can differ by a maximum of n.

3.2.5 Wildcard

Wildcard queries are a form of fuzzy, or inexact, matching. There are two main variants:
• Whole-word wildcards, where whole words are left unspecified. For example, searching for Paris is the * capital of the world matches documents that contain phrases "Paris is the romance capital of the world", "Paris is the fashion capital of the world", "Paris is the culinary capital of the world", and so on. This can be implemented efficiently as a variant of a phrase query with the wildcard word allowed to match any word.
• In-word wildcards, where part of a single word is left unspecified. It can be the end of a word (Helsin*), the start of the word (*sinki), the middle of the word (Hel*ki) or some combination of these (*el*nki).

These can be handled by first expanding the wildcard word to all the words it matches and then running the modified query normally with the search term replaced by (word1 OR word2 OR . . . wordn). To be able to expand the word, the inverted index needs a lexicon available. If it does not have a lexicon, there is no way to do this query.

If the lexicon is implemented as a tree of some kind, or some other structure that stores the words in sorted order, expanding suffix wildcards (Helsin*) can be done efficiently by finding all the words in the given range ([Helsin, Helsio]).If the lexicon is implemented as a hash table this cannot be done. Expansion of non-suffix wildcards is done by a complete traversal of the lexicon, and is potentially quite expensive. Since in-word wildcard queries need an explicit lexicon and are much more expensive in terms of time – and possibly space – needed than other kinds of queries, many implementations choose not to support them.

3.3 Result ranking

There are certain applications that do not care about the order in which the results of a query are returned, such as when the query is done by a computer and all the matching documents are processed identically. Usually, however, the query is done by a human being who is not interested in all the documents that match the query, but only in the few that best do so. It is for the latter case that ranking the search results is so important. With the size of the collections available today, reasonable queries can match millions of documents. If the search engine is to be of any practical use, it must be able to somehow sort the results so that the most relevant are displayed first.

Traditionally, the information retrieval field has used a similarity measure between the query and a document as the basis for ranking the results. The theory is that the more similar the query and the document are to each other, the better the document is as an answer to the query. Most methods of calculating this measure are fairly similar to each other and use the factors listed below in various ways.

Some of the factors to consider are: the number of documents the query term is found in (ft), the number of times the term is found in the document (fd,t), the total number of documents in the collection (N), the length of the document (Wd) and the length of the query (Wq).

If a document contains a few instances of a rare term, that document is in all probability a better answer to the query than a document with many instances of a common term, so we want to weigh terms by their inverse document frequency (IDF, or 1 ft). Combining this with the term frequency (TF, or fd,t) within a document gives us the famous TF × IDF equation.

The cosine measure is the most common similarity measure. It is an implementation of the TF × IDF equation with many variants existing, with a fairly typical one shown below:

$$cosine(Q, D_d) = \frac{1}{W_d W_q} \sum_{t \in Q \cap D_d} (1 + \log_e f_{d,t}) \cdot \log_e \left(1 + \frac{N}{f_t}\right)$$

The details of how Wd and Wq are calculated are not important in this context. Typically they are not literal byte lengths, or even term counts, but something more abstract like the square root of the unique term count in a document. They are not even necessarily stored at full precision, but perhaps with as few bits as five.

The above similarity measures work reasonably well when the queries are hundreds of words long, which is the case for example in the TREC (Text Retrieval Conference) competitions [tre], whose results are often used to judge whether a given ranking method is good or not.

Modern search engine users do not use such long queries, however. The average length of a query for web search engines is under three words, and the similarity measures do not work well for such queries.

There are several reasons for this. With short queries, documents with several instances of the rarest query term tend to be ranked first, even if they do not contain any of the other query terms, while users expect documents that contain all of the query terms to be ranked first.

Another reason is that the collections used in official competitions like TREC are from trusted sources and contain reasonable documents of fairly similar lengths, while the collections indexed in the real world contain documents of wildly varying lengths and the documents can contain anything at all. People will spend a lot of time tuning their documents so that they will appear on the first page of search results for popular queries on the major web search engines.

Thus, any naive implementation that tries to maximize the similarity between a query and a document is bound to do badly, as the makers of Google discovered when evaluating existing search engines. They tried a search for “Bill Clinton” and got as a top result a page containing just the text “Bill Clinton sucks”, which is clearly not the wanted result when the web is full of pages with relevant information about the topic.

3.4 Query evaluation optimization

Much research over the last 20 years has been conducted on optimizing query evaluation. The main things to optimize are the quality of the results returned and the time taken to process the query.

There are surprising gaps in the published research, however. The only query type supported by the best Internet search engines today is a hybrid mode that supports most of the extended query types discussed in Section 3.2 but also ranks the query results.

The query evaluation optimization research literature, however, ignores the existence of this hybrid query type almost completely and discusses just plain ranked queries, i.e., queries with no particular syntax which are supposed to return the k most relevant documents as the first k results. This is unfortunate since normal ranked queries are almost useless on huge collections like the Internet, because almost any query besides the most trivial one needs to use extended query methods like disallowing some words (Boolean AND NOT) and matching entire phrases (phrase query) to successfully find the relevant documents from the vast amounts in the collection.

The reason for this lack of material is obvious: the big commercial search engines power multi-billion dollar businesses and have had countless very expensive man years of effort from highly capable people invested in them. Of course they are not going to give away the results of all that effort for everyone, including their competitors, to use against them. Some day the details will leak out or an academic researcher will come up with them on his own, but the bar is continuously being raised, so I would not expect this to happen any time soon. That said, we now briefly mention some of the research done, but do not discuss the details of any of the work. Most of the optimization strategies work by doing some kind of dynamic pruning during the evaluation process, by which we mean that they either do not read all the inverted lists for the query terms (either skipping a list entirely or not reading it through to the end) or read them all, but do not process all the data in them if, it is unnecessary. The strategies can be divided into safe and unsafe groups, depending on whether or not they produce the exact same results as un-optimized queries. Buckley and Lewit were one of the first to describe such a heuristic.

Turtle and Flood give a good overview of several strategies. Anh and Moffat describe yet another pruning method. Persin et al. describe a method where they store the inverted lists not in document

order as is usually done, but in frequency-order, realizing significant gains in processing time.

There are two basic methods of evaluating queries: term-at-a-time and document at- a-time. In term-at-a-time systems, each query term's inverted list is read in turn and processed completely before proceeding to the next term. In document-at-a time systems, each term's inverted lists are processed in parallel. Kaszkiel and Zobel investigate which of these is more efficient, and end up with a different conclusion than Broder et al. who claim that document at- a-time is the more efficient one. To be fair, one is talking about context-free queries, i.e. queries that can be evaluated term by term without keeping extra data around, while the other one is talking about context-sensitive queries, e.g. phrase queries, where the relationships between query terms are important. Context sensitive queries are easier to handle in document-at-a-time systems since all the needed data is available at the same time.

Anh and Moffat also have another paper, this time on impact transformation, which is their term for a method they use to enhance the retrieval effectiveness of short queries on large collections. They also describe a dynamic pruning method based on the same idea.

Anh and Moffat make a t hird appearance with a paper titled "Simplified similarity scoring using term ranks", in which they describe a simpler system for scoring documents than what has traditionally been used.

Strohman et al. describe an optimization to document-at-time query evaluation they call term bounded max_score, which has the interesting property of returning exactly the same results as a un optimized query evaluation while being 61% faster on their test data. Carmel et al. describe a static index pruning method that removes entries from the inverted index based on whether or not the removals affect the top k documents returned from queries. Their method completely removes the ability to do more complex searches like Boolean and phrase searches, so it is usable only in special circumstances.

J´onsson et al. tackle an issue left alone in information retrieval research so far, buffer management strategies. They introduce two new methods: 1) a modification to a query evaluation algorithm that takes into account the current buffer contents, and 2) a new buffer-replacement algorithm that incorporates knowledge of the query processing strategy. The applicability of their methods to generic Full-Text Search (FTS) systems is not especially straightforward, since they use a very simplistic Full –Text Search (FTS) system with only a single query running at one time and other restrictions not found in real systems, but they do have some intriguing ideas.

4. Conclusion

To construct inverted index is an important issue in web search engines. In this paper we showed how to build inverted index for web search engine. We have discussed about the different strategies, and techniques. We have also discussed about various research findings on how to efficiently build inverted index.

References

[1] Vo Ngoc Anh and Alistair Moffat, " Compressed inverted files with reduced decoding overheads". In SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval, 1998, pp.290–297, New York, NY, USA, ACM Press.

[2] Vo Ngoc Anh and Alistair Moffat,"Impact transformation: effective and efficient web retrieval", In SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval", 2002, pp.3–10, New York, NY, USA, ACM Press.

[3] Vo Ngoc Anh and Alistair Moffat, "Simplified similarity scoring using term ranks", In SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval,2005, pp.226–233, New York, NY, USA, ACM Press.

[4] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien, " Efficient query evaluation using a two-level retrieval process", In CIKM '03: Proceedings of the 12th international conference on I nformation and knowledge management, 2003, pp.426–434, New York, NY, USA, ACM Press.

[5] Chris Buckley and Alan F. Lewit, " Optimization of inverted vector searches", In SIGIR '85: Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval, 1985, pp. 97–110, New York, NY, USA, ACM Press.

[6] Sergey Brin and Lawrence Page, "The anatomy of a large-scale hyper textual Web search engine", Computer Networks and ISDN Systems, Vol.30, No.17, 1998, pp.107–117.

[7] Dirk Bahle, Hugh E. Williams, and Justin Zobel, "Efficient phrase querying with an auxiliary index", In SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval, 2002, pp.215–221, New York, NY, USA, ACM Press.

[8] Charles L. A. Clarke, Gordon V. Cormack, and Elizabeth A. Tudhope, " Relevance ranking for one to three term queries", Information Processing and Management, Vol.36, No.2 ,2000, pp.291–311.

[9] Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane, " Dynamic dictionary matching and compressed suffix trees", In SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, 2005, pp.13–22, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics.

Mr. Ajit Kumar Mahapatra is presently working as an Asst. Professor in Information Technology Dept., ITER, Bhubaneswar under Siksha 'O' Anusandhan University (SOAU). Also,He is continuing his M.Tech in IT Dept. in Institute of Technical Education & Research (ITER), Bhubaneswar under Siksha 'O' Anusandhan University. She is having 6 y ears of teaching experience and keen interest in the areas such as Web Search Engine Optimization and building.

Mr. Sitanath Biswas is presently working as an A sst. Professor in Information Technology Dept., Institute of Technical Education & Research (ITER), Bhubaneswar under Siksha O Anusandhan University. He is having 3 y ears of teaching experience. H e got his M.E. Degree from Utkal University, Bhubaneswar. He published 15 r esearch papers in International and National journals and delivered invited lectures at various conferences, seminars and workshops. He guided a large number of students for their B.Tech and M .Tech degrees in Computer Science and E ngineering and I nformation Technology. His current research interests are Artificial Intelligence, Semantic Web, and Ontology Engineering