Optimizations for Dynamic Inverted Index Maintenance

Doug Cutting and Jan Pedersen

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California

Abstract

For free-text search over rapidly evolving corpora, dynamic update of inverted indices is a basic requirement. B-trees are an effective tool in implementing such indices. The Zipfian distribution of postings suggests space and time optimizations unique to this task. In particular, we present two novel optimizations, *merge update*, which performs better than straight forward block update, and *pulsing* which significantly reduces space requirements without sacrificing performance.

Inverted Indices

Most standard free-text search methods in Information Retrieval (IR) can be implemented efficiently through the use of an inverted index. These include standard boolean, extended boolean, proximity, and relevance search algorithms. [7]

An inverted index is a data structure that maps a word, or atomic search item, to the set of documents, or set of indexed units, that contain that word — its *postings*. An individual posting may be a binary indication of the presence of that word in a document, or may contain additional information, such as its frequency in that document and an offset for each occurrence, required for various non-boolean search algorithms. In the following, we will simplify this situation by assuming that each word *occurrence* indexed has a corresponding posting. This approximation has the advantage of being amenable to analysis.

Since access to an inverted index is based on a single key (i.e. the word of interest) efficient access typically implies that the index is either sorted or organized as a hash table. In the following we will assume that keys are sorted. For hashing schemes, the interested reader is directed to [6].

As a part of an operational IR system, properties beyond formal description become important. This paper is concerned with the following performance criteria:

- Block Update Speed: The time required to index documents.

  It is presumed that a practical IR system will be manipulating indices too large to conveniently fit in main memory, and hence that the inverted index will be represented as a data structure on secondary storage. Since insertion in a sorted structure is at best a $\log n$ operation, where n is the number of previously indexed postings, we will measure performance by the number of references to secondary storage. This is further justified by noting that these sorts of computations are typically completely dominated by disk access time.

- Access Speed: The time required to access the postings for a given word.

  This parametrizes search performance, and, hence, is extremely user-visible. Again, access time is inherently $\log n$, but may require fewer than $\log n$ disk accesses.

- Index Size: The amount of storage required for the inverted index.

  Since some record must be made for each posting, the inverted index must be proportional to the number of postings. This proportionality constant is referred to as the *indexing overhead*, or the size of the index expressed as a percentage of the size of the entire corpus.

- Dynamics: The ease with which the inverted index is incrementally updated.

  This is particularly important for rapidly evolving corpora. Insertion is typically more common than deletion. Many indexing schemes presume a static corpus. [2, 7] These may be updated only by reconstructing the entire index. We will only discuss incrementally updatable indices in this paper.

- Scalability: The relation between the above and corpus size.

  Indexing algorithms should scale gracefully with increasing corpus size. In particular, main memory usage should be independent of corpus size.

We assume that each of the above are important, and seek methods which perform well on all these axes.

B-trees are a file-based data structure particularly appropriate for the implementation of dynamically modifiable inverted indices. The following will analyze the use of B-trees in this task and suggest several novel time and space optimizations.

B-trees

Conceptually, a B-tree maintains an ordered sequence as an n-ary branching balanced tree, where the tree resides on secondary storage rather than in main memory. [1, 5, 6] Nodes in the tree are represented as disk *pages*, and rebalancing algorithms insure insertion, examination and deletion of entries in $O(\log_b N)$ disk accesses where b is the *branching factor* of the B-tree and N is the number of entries contained in the B-tree. The quantity $\log_b N$ is referred to as the *depth* of the B-tree. Entries may also be enumerated in sorted order in time proportional to N, requiring roughly N/b disk accesses.

The branching factor b is related to both the page size and the average size of entries, but is typically fairly large, say around 100. This means that any entry in such a B-tree containing a million entries may be accessed in three disk accesses. This may be improved by retaining some of the B-tree nodes or pages in main store. Since each page contains b entries on average, holding one page in memory requires storage proportional to b. If the root page is kept in core then we can reduce by one the number of disk accesses required for any operation — at the cost of storing just b entries. To make another similar gain we must cache all the immediate children of the root. In general, the cost of access is

$$\log_b N - \log_b C, \text{ for } N \geq C \tag{1}$$

disk accesses, where C is the number of entries stored in core, or the *cache size*. Note, we are describing an upper-nodes caching strategy. Other strategies, such as an LRU (least recently used) cache, have similar performance characteristics. [5]

If $C = N$, then the entire B-tree is represented in core, and no disk accesses are required. If $C = N/b$, all the non-terminal nodes are cached in core, and any access operation may be performed with just one disk access. In our example, with $b = 100$, a B-tree of size $N = 1,000,000$ is of depth 3, and a cache of size $C = N/b = 10,000$ guarantees random access with just one disk operation.

Although the typical inverted index operation is random access to the postings for a given word, the sorted enumeration property of a B-tree may be exploited for prefix wild-carding. Suppose words are sorted in lexicographic order, then words with a common prefix are adjacent in the B-tree. If $b \log_b N$ storage is allocated to hold the path of pages between the root and the current point in an enumeration, a disk operation need only occur every b adjacent entries.

Naive B-tree Indexing

A B-tree inverted index clearly addresses the issues of access speed, dynamic update, and scalability. Access to any given entry requires no more than $\log_b N$ disk accesses, B-trees are intrinsically updatable, and access times may be reduced through the use of relatively small page caches. However, we have yet to discuss the time required for a block update or the space occupied by a B-tree index. In the following we will analyze block update time in terms of the number of disk *reads* required. An actual update will, of course, also requires disk *writes*, but each read will require no more than one write, hence the total cost of an update operation is no worse than proportional to the number of disk reads.

A simple approach to constructing a B-tree inverted index is to consider each entry to be a pair of the form

$$\langle \textit{word}, \textit{location} \rangle$$

ordered first by *word* and second by *location*. Thus, an entry is just a posting ordered so that the postings for the same word are adjacent. Random access to all the postings for a given word is simply B-tree enumeration, requiring a disk access for every b postings. Index update with this representation requires a B-tree insert for every word instance in the new text. Thus, by (1), indexing n words of new text requires

$$n(\log_b N - \log_b C) \tag{2}$$

disk reads.

Removal of references to a document can be accomplished at the same expense as insertion, providing the document is still available. If the tokens comprising the document are no longer available, then an exhaustive enumeration of the index is required to find and remove references to it.

A Speed Optimization

It has been shown that the number of disk accesses required to perform an update can be reduced by caching the upper nodes of the B-tree in core. If, instead, this core memory is used as a buffer for postings, which is merged with the B-tree when full, many more disk accesses can be eliminated. Equation (2) gives the expected number of disk reads to insert n new postings using a page cache containing room for C entries. If these same n postings are buffered and sorted by word in core, then, rather than having n instances, we have w words, each with an associated list of postings. (The time required to sort in core is ignorable, since it requires no disk accesses and is, in any event, much less than the time required to sort n postings via B-tree inserts.) At merge time, we must insert these w entries into the B-tree. Since the postings for a given entry typically fit on a single B-tree page, this is approximately equivalent, in disk access time, to w inserts in sort order. If we assume that these entries are uniformly distributed over the set of keys, each ordered insert will require an average advance of N/w entries through the B-tree.

Suppose $\log_b N$ pages are held in core to hold a path between the root and a leaf. Then the cost of each such advance can be estimated as follows. Consider the *sub*-B-tree defined by the span to be advanced over. This B-tree contains, by definition, N/w entries of which the leftmost path is in core. To access the last entry in the span we must bring the rightmost path into core. This requires $\log_b(N/w)$ disk transactions. Hence

$$w(\log_b(N/w)) = w(\log_b N - \log_b w) \tag{3}$$

disk accesses are required on average to index n postings with this technique.

If n is large the frequency distribution of unique words will approximately follow Zipf's law.[8] In other words

$$f(w)r(w) \approx z \tag{4}$$

for some constant z, where $f(w)$ is the frequency of word w in the set of n instances and $r(w)$ is the rank of $f(w)$ among the frequencies of all words in that set of postings. Note that in this approximation z is both the vocabulary size and the frequency of the most frequent word. It follows that

$$n \approx \sum_{r=1}^{r=z} z/r \approx z \int_{r=1}^{r=z} 1/r\,\mathrm{d}r = z \ln z. \tag{5}$$

In other words, the vocabulary size, z, grows much less rapidly than the number of word occurrences, n, and can, in fact, be estimated given n.

To demonstrate that memory is better utilized in a buffer than in a page cache, we must show that for constant C the cost in disk accesses, (2), denoted by X, is greater than (3), denoted by Y. This may be expressed by equating the number of postings n with C, since the same memory may either be allocated to a page cache or to store postings in a buffer. From these considerations and (5) we have

$$X = z \ln z(\log_b N - \log_b(z \ln z)), \tag{6}$$

and

$$Y = z(\log_b N - \log_b z). \tag{7}$$

Clearly,

$$Y = X/\ln z + z \log_b(\ln z). \tag{8}$$

Suppose, $X > Y$, then, by substitution of (8),

$$X > X/\ln z + z \log_b(\ln z),$$

or, by rearrangement,

$$\frac{X}{z \ln z}(\ln z - 1) > \log_b(\ln z).$$

After substituting in (6) and rearranging terms, this reduces to

$$(\ln z - 1)(\log_b N - \log_b(z \ln z)) > \log_b(\ln z)$$

or

$$\log_b N > \log_b z + \log_b(\ln z)\frac{\ln z}{\ln z - 1}. \tag{9}$$

Exponentiating both sides leads to

$$N > z(\ln z)^{\frac{\ln z}{\ln z - 1}} \tag{10}$$

In other words, $X > Y$ iff inequality (10) holds. Now, since $z \ln z = n = C$ and (2) is only valid if $N \geq C$, we have $N > z \ln z$. For substantial z, the exponent $\ln z/(\ln z - 1)$ will be close to unity. Hence, for the update case, we can expect $X > Y$.

For the example under consideration, $b = 100$, $N = 1,000,000$ and $z \ln z = 10,000$, and $z \approx 1,383$. Hence $X \approx 10,000$ and $Y \approx 1,977$. Indeed, from (6), we expect

$$X = \ln z(Y - z \log_b(\ln z)). \tag{11}$$

Hence, substantially fewer disk accesses are required if memory is used as buffer for sorting and subsequently merging postings into an existing B-tree inverted index than if it is allocated as a B-tree page cache and updates are preformed in occurrence order.

Table 1: Merge Experiment

case		predicted		observed		
		z	reads	z	reads	writes
100	cache	30	200	74	302	260
	merge		68		216	201
1,000	cache	191	1,500	420	2,587	2,038
	merge		355		931	867
10,000	cache	1,383	10,000	3,421	16,012	13,605
	merge		1,977		5,049	4,890
100,000	cache	10,800	50,000	14,105	70,721	63,530
	merge		10,620		7,795	9,100

Experimental Results

Table (1) contains the results of experiments which demonstrate the effectiveness of the merge optimization. Initially a B-tree was created containing the postings for a corpus with one million word instances. Its branching factor, b, averaged 100, and its depth was three. We ran cached and merged updates of 100, 1000, 10,000, and 100,000 new postings. The B-tree cache size, C, and the merge buffer size for each trial was set equal to the number of new postings.

The predictions were based on equations (5), (6), and (7). The observed results for the merge case are higher than expected due to the inaccuracies of Zipf's law in predicting vocabulary size, z. The observed results for both cases are also slightly higher than expected because equations (6) and (7) do not account for page reads due to B-tree rebalancing. Writes exceed reads in one case due to the creation of new pages.

Space Optimizations

There is obvious redundancy in the 'naive' indexing scheme presented above; each word instance requires a reference to the word itself. If, instead, the set of postings for a word is decomposed into the word and a sequence of locations, then much of this redundancy is eliminated. There is a small overhead in performing this grouping, since any representation for a sequence of locations requires some record of its length, or a termination. A prefix length indication is preferred since it can then serve as a record of the marginal, or corpus, frequency. In other words, in addition to requiring a 'cell' for each word and for each location of that word, one additional cell is required to note the total number of locations.

For example, in the naive case, words of frequency one occupy two cells, while the revised representation requires three cells; words of frequency two require four cells in both representations. In general, the naive representations requires $2N$ cells for N postings, while, by (5), grouping requires

$$\sum_w (2 + f(w)) \approx W(2 + \ln W)$$

cells, where W is the vocabulary size of the same N postings. Since $W \ln W \approx N$, this is smaller than $2N$ if $2W < N$, which is true if $\ln W > 2$ or $W > 7.39$. The ratio of cells required for the two strategies is

$$\frac{2W(1 + \ln W/2)}{2W \ln W} = 1/\ln W + 1/2, \qquad (12)$$

or slightly over 1/2. Hence, grouping postings reduces space requirements by almost 50%.

If words are represented as integers, as would be the case if a lexicon is built to provide a 'word to number' mapping, and locations are also integers, then all cells may be presumed to be roughly equivalent in size. In this case, the space analysis above refers to the real sizes of the respective indices.

Heap Update

A grouped index may be implemented by considering variable-length B-tree entries of the form

$$\langle word, F, \langle location\rangle^{\star}\rangle, \qquad (13)$$

where F is the marginal frequency of *word* and *location* refers to the corpus position of a single posting. As only one entry exists per unique word, the B-tree's ordering function need only examine *word*. A difficulty immediately arises, however, for words with a large number of postings. Recall that, by definition, the maximum size for any B-tree entry is one B-tree page; if the locations sequence overflows this limit no recourse is available. Indeed, if l is the number of locations that fit on one B-tree page, then from (5), we expect a corpus of size $l \ln l$

postings to overflow this limit for the highest frequency word.

This situation may be ameliorated by indexing tuples of the form

$$\langle word, F, pos \rangle \tag{14}$$

where *pos* indicates a position in an auxiliary data structure, known as a *heap file*, where the sequence of locations can be found. As is suggested by its name, a heap file is simply a binary file manipulated as if it were main memory. Continuous chunks of this memory are allocated as necessary for the storage of arbitrary data, in this case location sequences. Update is accomplished either in place, or if sizes change sufficiently, by allocating a new chunk to hold the updated sequence and freeing the old chunk. This implies that the chunks comprising a heap file are maintained by a dynamic storage allocation algorithm.

One such algorithm, the *buddy system*, allocates chunks in sizes that are powers of two. [4] This arrangement insures that chunks are on the average 75% full, which is comparable to the storage utilization of B-trees.

Access to an individual postings list for representation (14) requires no more than $\log_b W + 1$ disk reads; $\log_b W$ to read in the B-tree entry followed by one more to access the heap at *pos*. Block update proceeds by, for each new instance encountered, appending a new location to the end of the chunk at *pos*, computed by adding F to *pos*, and incrementing F. An additional cost is incurred when a chunk is filled, since its contents must be copied to a freshly allocated twice-larger chunk, and the old chunk deallocated. Since chunks are allocated in powers of two, a location sequence of length f must be copied no more than $\log_2 f$ times. This amortized per instance cost of $(\log_2 f)/f$ additional accesses is sufficiently small to be ignorable. Hence, the block update time is proportional to

$$n(\log_b W - \log_b C + 1) \tag{15}$$

for n postings. The 'merge' optimization mentioned above may also be applied in this case to reduce n at the expense of the B-tree page cache.

The space requirements for representation (14) are similar to (13) with one additional cell to hold the heap file pointer. Hence, N postings occupy

$$W(3 + \ln W)$$

cells. The ratio with respect to the naive indexing strategy is

$$3/(2 \ln W) + 1/2,$$

which is only slightly greater than (12).

Pulsing

Use of a heap file solves the B-tree page overflow problem at the cost of slightly increased access time and slightly larger overall index size. Yet, B-tree overflows will only occur for the relatively few high frequency words. This observations leads one to consider buffering postings directly in the B-tree, and overflowing to a heap file only when necessary, a technique known as *pulsing*. Essentially, a threshold t is chosen which determines the maximum number of locations which are to be stored immediately. Updates are made directly in the B-tree, and only after t new instances of a word are seen are their t locations *pulsed* to the heap. In other words, at most, the t newest posting for any given word appear directly in the B-tree; any additional postings are found in the heap file.

In this representation B-tree entries have the form

$$\langle word, F, l, \langle doc \rangle^\star, pos \rangle \tag{16}$$

where *word*, F and *pos* are as in (14), and l is the length of the locations $\langle doc \rangle^\star$. By convention, l and *pos* need not be provided when F is less than t as a space-saving measure.

For words, w, of frequency $f(w) < t$, all postings are directly accessible. Hence, the cost of access is no more than $\log_b E$, where E is the effective number of postings in the B-tree, $E \leq N$. We can estimate E by computing the number of occurrences of words whose frequencies are less than or equal to t. From (5) we have

$$f(w) \leq t \ \text{ iff } \ r(w) \geq W/t.$$

It follows that

$$E = \sum_{\{w:\ f(w) \leq t\}} f(w) \ \approx$$

$$\int_{W/t}^{W} \frac{W}{r}\ \mathrm{d}r = W \ln t \tag{17}$$

If $f(w) > t$, than the cost of access is no more than $(\log_b E + 1)$. Since $W \ln t$ postings reside directly in the B-tree, we will avoid heap file access with probability

$$p = \ln t / \ln W.$$

Hence, the expected access time is no worse than

$$p \log_b E + (1-p)(\log_b E + 1) = \log_b E + (1-p).$$

In other words, in comparison to (14), access is accelerated by the hit rate p, but potentially penalized by the larger B-tree size, E. In particular, the ratio of access times is

$$\frac{\log_b W + 1}{\log_b E + (1-p)} = \frac{\log_b W + 1}{\log_b W + 1 + (\log_b(\ln t) - p)}$$

This ratio is greater than one iff

$$\log_b(\ln t) < p \quad \text{or} \quad t < e^{b^p}.$$

However, $t < b$ since we cannot buffer more postings than the B-tree page size directly in the B-tree.

From the above, with the introduction of a page cache, block update has expected cost proportional to

$$n(\log_b E - \log_b C + (1-p)). \tag{18}$$

Representation (16) also occupies less space than (14) since heap file indirection can be avoided with probability p. For an entry with $f(w) \leq t$, we use $2+f(w)$ cells. An entry with indirection uses $4+f(w)$ cells. Hence the space occupied for N postings is

$$\sum_{\{w:f(w)\leq t\}} (2+f(w)) + \sum_{\{w:f(w)>t\}} (4+f(w)) \approx$$

$$\int_{W/t}^{W} (2+W/r))\ \mathrm{d}r + \int_{1}^{W/t} (4+W/r)\ \mathrm{d}r =$$

$$W \ln W + 2W(1+1/t) - 4 \tag{19}$$

The threshold, t, may be selected to generate a desired 'hit' probability, p. For our example, $N = 1,000,000$ and $W \approx 88,000$. To achieve a hit rate of 25% we require $t = \sqrt[4]{W}$, or $t \approx 17$. In other words, use of pulsing with a threshold of 17 reduces the frequency of access to the heap file by 1/4.

Delta Encoding

A typical inverted index operation randomly accesses the postings for a given word. However, these postings are typically processed linearly. In other words, an inverted index need not provide easy random access to individual postings. This suggests that the sequence of postings may be compressed in any fashion that requires no worse than linear time to decode.

A pulsing strategy arranges for locations to be stored in the order indexed. If each location is represented as an integer, and these integers are allocated in an increasing manner, then a particularly simple compression scheme we term *delta encoding* is possible. [5] Rather than storing an actual location the difference between it and the previous location, a delta, is stored instead. This yields a sequence of much smaller integers than the original sequence of locations.

In itself this is uninteresting unless integers are encoded in such a way that small integers occupy less space than large integers. A typical scheme for performing such an encoding employs the high order bit of each byte to indicate whether another byte need be read. Thus, with eight-bit bytes, the numbers 0-127 may be represented in one byte, 128-16384 in two bytes, and so on. [3] Thus, if a word occurs on average every 64 locations, each location will, on average, occupy only one byte.

In order to easily incrementally append new locations at the end of an existing chunk of locations (i.e. without having to decode the entire chunk) the size of each block and the last location in it must also be maintained. These can be maintained in the B-tree or at the beginning of each block. The former is preferred as it minimizes the amount of the block which must be touched. As with other information about the chunk, e.g. its position in the heap, these need not be stored at all for words whose marginal frequency is less than the threshold t.

There is one application in which random access to individual postings is desirable; that is deletion. A compression strategy, such as delta encoding will require us to read the entire postings list for a word to delete a single entry, and will require us to rewrite it since the sequence of postings will have changed. In other words, space and access time is optimized at the expense of this relatively rare operation.

Conclusion

For free-text search over rapidly evolving corpora, dynamic update of inverted indices is a basic requirement. B-trees are an effective tool in implementing such indices and may be optimized to reduce access and update time and to minimize size. A speed optimization, *merge update*, performs better than straight forward block update and two space optimizations, *pulsing* and *delta encoding* significantly reduce space requirements without sacrificing performance.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[2] Harman. D. and G. Candela. A very fast prototype retrieval system using statistical ranking. *SIGIR Forum*, 23(3,4):100–110, Summer 1989.

[3] H. S. Heaps. Storage analysis of a compression coding for a document database. *INFOR*, 10(1):47–61, February 1972.

[4] D. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, 1968.

[5] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[6] G. Salton. *Automatic Text Processing.* Addison-Wesley, 1989.

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

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