Index Compression through Document Reordering

Dan Blandford and Guy Blelloch
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{blandford,blelloch}@cs.cmu.edu
(412) 268-3074 *

January 11, 2002

1 Abstract

An important concern in the design of search engines is the construction of an inverted index. An inverted index, also called a concordance, contains a list of documents (or *posting list*) for every possible search term. These posting lists are usually compressed with difference coding. Difference coding yields the best compression when the lists to be coded have high locality. Coding methods have been designed to specifically take advantage of locality in inverted indices. Here, we describe an algorithm to permute the document numbers so as to create locality in an inverted index. This is done by clustering the documents. Our algorithm, when applied to the TREC ad hoc database (disks 4 and 5), improves the performance of the best difference coding algorithm we found by fourteen percent. The improvement increases as the size of the index increases, so we expect that greater improvements would be possible on larger datasets.

2 Introduction

Memory considerations are a serious concern in the design of search engines. Some web search engines index over a billion documents, and even this is only a fraction of the total number of pages on the Internet. Most of the space used by a search engine is in the representation of an *inverted index*, a data structure that maps search terms to lists of documents containing those terms. Each entry (or *posting list*) in an inverted index is a list of the document numbers of documents containing a specific

*This work was supported in part by the National Science Foundation under grants CCR-9706572, CCR-0085982, and CCR-0122581.

term. When a query on multiple terms is entered, the search engine retrieves the corresponding posting lists from memory, performs some set operations to combine them into a result, sorts the resulting hits based on some priority measure, and reports them to the user.

A naive posting list data structure would simply list all the document numbers corresponding to each term. This would require $\lceil \log(n) \rceil$ bits per document number, which would not be efficient. To save space, the document numbers are sorted and then compressed using *difference coding*. Instead of storing the document numbers themselves, a difference-coding algorithm stores the difference between each document number and its immediate successor. These differences are likely to be smaller than the original numbers, so the algorithm can save space by using a code which represents small numbers using a smaller number of bits.

In general, a difference-coding algorithm will get the best compression ratio if most of the differences are very small (but one or two of them are very large). Several authors [8, 2, 3] have noted that this is achieved when the document numbers in each posting list have high locality. These authors have designed methods to explicitly take advantage of this locality. These methods achieve significantly improved compression when the documents within each term have high locality. However, all compression methods thus far have been devoted to passive exploitation of locality that is already present in inverted indices.

Here, we will study how to improve the compression ratio of difference coding on an inverted index by permuting the document numbers to actively create locality in the individual posting lists. One way to accomplish this is to apply a hierarchical clustering technique to the document set as a whole, using the cosine measure as a basis for document similarity. Our algorithm can then traverse the hierarchical clustering tree, applying a numbering to the documents as it encounters them. Documents that share many term lists should be close together in the tree and therefore close together in the numbering.

We have implemented this idea and tested it on indexing data from the TREC-8 ad hoc track [9] (disks 4 and 5, excluding the Congressional Record). We tested a variety of codes in combination with difference coding. Our algorithm was able to improve the performance of the best compression technique we found by fourteen percent simply by reordering the document numbers. The improvement offered by our algorithm increases with the size of the index, so we believe the improvement on larger real-world indices would be greater.

Conceptually, our ORDER-INDEX algorithm is divided into three parts. The first part, BUILD-GRAPH, constructs a document-document similarity graph from an index. The second part, SPLIT-INDEX, makes calls to the Metis [7] graph partitioning package to recursively partition the graphs produced by BUILD-GRAPH. It uses these partitions to construct a hierarchical clustering tree for the index. The third part of our algorithm, ORDER-CLUSTERS, applies rotations to the clustering tree to optimize the ordering. It then numbers the documents with a simple depth-first traversal of the clustering tree. At all levels we apply optimizations and heuristics to ensure that the time and memory requirements of our algorithm will scale well.

In practice, constructing the full hierarchical clustering would be infeasible, so the three parts of our algorithm are combined into a single recursive procedure that makes only one pass through the clustering tree.

The remainder of our paper is organized as follows. Section 3 formalizes the problem. Section 4 describes our algorithm in detail. Section 5 demonstrates the performance of our algorithm when run on the TREC-8 database. Section 6 presents directions for future work.

3 Definitions

We describe an inverted index I as a set of terms $t_1 \ldots t_m$. For every term t_i there is an associated list of $|t_i|$ document numbers $d_{i,1} \ldots d_{i,|t_i|}$. The document numbers are in the range $1 \leq d_{i,j} \leq n$. We are interested in the cost of representing these documents using a difference code. Thus we define, first, $s_i(d_i) = s_{i,1} \ldots s_{i,|t_i|}$ to be the sequence of documents $d_{i,j}$, rearranged so that $s_{i,j} < s_{i,j+1}$ for all j. That is, s_i is the sorted version of the sequence of documents d_i. (For convenience we also define $s_{i,0} = 0$ for all i.) Then, if we have an encoding scheme c which requires $c(d)$ bits to store the positive integer d, we can write the cost C of encoding our index as follows:

$$C(I) = \sum_{i=1}^{n} \sum_{j=1}^{|t_i|} c(s_{i,j} - s_{i,j-1})$$

We wish to minimize $C(I)$ by creating a permutation σ which reorders the document numbers. Since c is convex for most useful encoding schemes, this means we need to cluster the documents to improve the locality of the index.

4 Our Algorithm

Document Similarity. Up to this point, we have viewed an inverted index as a set of terms, each of which contains some subset of the documents. Now it will be convenient to consider it as a set of documents, each of which contains some subset of the terms. Specifically, we can consider a document to be an element of $\{0,1\}^m$, where the i^{th} element of a document is 1 if and only if the document contains term t_i. Eventually our algorithm will need to compute centers of mass of groups of documents, and then it will be convenient to allow documents to contain fractional amounts of terms—that is, to represent a document as an element of $\Re^m$.

Our algorithm uses the *cosine measure* to determine the similarity between a pair of documents:

$$\cos(A,B) = \frac{A \cdot B}{((A \cdot A)(B \cdot B))^{\frac{1}{2}}}$$

Build-Graph. Using this similarity measure our BUILD-GRAPH algorithm can construct a document-document similarity graph. For large databases, creating a full graph with n^2 edges is not feasible. However, most of the documents contain only

```
SPLIT-INDEX($I$):
  $I' \leftarrow$ SUBSAMPLE($I$, $|I|^\rho$)
  $G \leftarrow$ BUILD-GRAPH($I'$)
  $(G_1, G_2) \leftarrow$ PARTITION($G$)
  $d_1 \leftarrow G_1.centerofmass$
  $d_2 \leftarrow G_2.centerofmass$
  $I_1, I_2 \leftarrow$ empty indices
  foreach $d$ in $I$
    if COS($d$, $d_1$) $>$ COS($d$, $d_2$) then
      add $d$ to $I_1$
    else
      add $d$ to $I_2$
  return $(I_1, I_2)$
```

```
BUILD-GRAPH($I$):
  $G \leftarrow$ new Graph
  foreach $d_1$ in $I$
    foreach $t$ in $d_1$
      if $|t| \leq \tau$ then
        foreach $d_2$ in DOCLIST($t$)
          $e \leftarrow$ new Edge($d_1$, $d_2$, COS($d_1$, $d_2$))
          add $e$ to $G$
  return $G$
```

Figure 1: Our BUILD-GRAPH and SPLIT-INDEX algorithms.

a small fraction of the total set of terms. It seems reasonable that the graph might besparse: many of the edges in the similarity graph might actually have a weight of zero. This is especially true if we remove common "stopwords" from consideration, as described below.

To save space, BUILD-GRAPH uses the following method to generate the graph. Consider the index to be a bipartite document-term graph from which BUILD-GRAPH needs to generate a document-document graph. For each term in the document-term graph, our algorithm eliminates that term and inserts edges (weighted with the cosine measure) to form a clique among that node's neighbors. After eliminating all of the terms in the document-term graph, BUILD-GRAPH has produced a document-document graph which contains an edge between every pair of documents that shares a common term.

If term t_i contains $|t_i|$ documents, then BUILD-GRAPH will compute $O(\sum |t_i|^2)$ cosine measures in computing the edge graph. Our algorithm can improve this bound slightly by being careful never to compute the same cosine measure more than once, but the worst-case number of cosine measures will still be $O(\sum |t_i|^2)$.

However, BUILD-GRAPH does not actually need all this information in order to represent the structure of the similarity graph. In particular, a lot of the documents in the index are likely to be "trivially" similar because they share terms such as "a", "and", or "the". The most frequently occurring terms in the index are also the least important to the similarity measure. Removing the edges corresponding to those terms should not have much of an impact on the quality of the ordering (as demonstrated in Section 5), while it should decrease the work required for BUILD-GRAPH considerably. Thus, when generating the graph, our algorithm creates cliques among the neighbors of only those terms with less than a threshold number of neighbors τ. All other terms are simply deleted from the graph. (This technique is similar to that used by Broder et al. [5] for identifying near-duplicate web pages.) Pseudocode for this part of our algorithm is shown in Figure 1.

Split-Index. Once build-graph has produced a similarity graph, the next step is to derive a hierarchical clustering of that graph. There are a large number of hierarchical clustering techniques that we could choose from (for example, those of [4, 10], and additional references from those papers), but most of those techniques are not designed for data sets as large as the ones we are dealing with. Many of them, in fact, require as input an $O(n^2)$ similarity matrix. We do not have enough space or time to even construct a matrix of that size, much less run a clustering algorithm on it. Furthermore, this problem has certain special features which are not captured by any general clustering algorithms. Therefore we have created our own hierarchical clustering algorithm based on graph partitioning.

A naive hierarchical clustering algorithm would work as follows. Given an index, compute a similarity graph from that index. Partition the graph into two pieces. Continue partitioning on the subgraphs until all pieces are of size one. Use the resulting partition tree as the clustering hierarchy.

Unfortunately, this algorithm uses too much memory. Our build-graph algorithm requires less than the full $O(n^2)$ memory, but it is still infeasible to apply it to the full index. Instead, split-index uses a sampling technique on the index: at each recursive step, it subsamples some fraction of the documents from the original index. It runs build-graph on this subindex and partitions the result. Once it has done this, split-index uses the subgraph partition to partition the original index. To do this, it computes the centers of mass of the two subgraph partitions. It then partitions the documents from the original index based on which of the centers of mass they are nearest to. Pseudocode for split-index is shown in Figure 1.

An interesting point to note is that split-index recreates the document similarity graph at each node of the recursion tree. This offers our build-graph algorithm significantly more flexibility when creating the similarity graph: build-graph only needs to create the graph in such a way that the *first* partition made on it will be a good one. This allows build-graph to use a very small value of τ: if a term occurs more than, say, 10 times at a given partition level, it is likely that any partition split-index computes will have documents containing this term on both sides anyway. Thus build-graph ignores that term until later iterations.

Order-Clusters. Once split-index has produced a hierarchical clustering, order-index uses that clustering to create a numbering of the leaves. To do this it performs an inorder traversal of the tree. At each step, however, it needs to decide which of the two available partitions to traverse first. In essence, our order-clusters algorithm looks at every node in the hierarchy and decides whether or not to swap its children.

Within any given subtree S, there are four variables to consider. We denote the children of S by I_1 and I_2. We also define I_L and I_R to be the documents that will appear to the immediate left and right of S in the final ordering. (At the first recursion we initialize I_L and I_R to place equal weight on each term. This causes infrequently-occurring terms to be pulled away from the middle of the ordering.) Since order-clusters operates with a depth-first traversal, we take I_L to be the left child of S's left ancestor, and I_R to be the right child of S's right ancestor.

```
ORDER-CLUSTERS($I_L$, $I_1$, $I_2$, $I_R$):
    $m_L \leftarrow I_L.centerofmass$
    $m_1 \leftarrow I_1.centerofmass$
    $m_2 \leftarrow I_2.centerofmass$
    $m_R \leftarrow I_R.centerofmass$
    $s_1 \leftarrow \cos(m_L, m_1) * \cos(m_R, m_2)$
    $s_2 \leftarrow \cos(m_L, m_2) * \cos(m_R, m_1)$
    if $s_2 > s1$ then
        return ($I_2$, $I_1$)
    else
        return ($I_1$, $I_2$)
```

```
\\ Assigns the numbers between $\ell$ and $h$
\\ to the documents of an index $I$,
\\ which must have exactly $(h - \ell + 1)$
\\ documents.
ORDER-INDEX($I$, $\ell$, $h$, $I_L$, $I_R$):
  if $\ell = h$ then
    $I.v[0].number \leftarrow \ell$
  else
    ($I_1$, $I_2$) $\leftarrow$ SPLIT-INDEX($I$)
    ($I_1$, $I_2$) $\leftarrow$ ORDER-CLUSTERS($I_L$, $I_1$, $I_2$, $I_R$)
    ORDER-INDEX($I_1$, $\ell$, $m - 1$, $I_L$, $d_2$)
    ORDER-INDEX($I_2$, $m$, $h$, $I_1$, $I_R$)
```

Figure 2: Our ORDER-CLUSTERS and ORDER-INDEX algorithms.

ORDER-CLUSTERS tracks the centers of mass of each of these clusters, and it rotates I_1 and I_2 so as to place similar clusters closer together.

Pseudocode for ORDER-CLUSTERS and for the main body of our algorithm is shown in Figure 2.

5 Experimentation

Compression Techniques. We tested several common difference codes to see how much improvement our algorithm could provide. The codes we tested include the delta code, Golomb code [6], and arithmetic code. These codes are described in more detail by Witten, Moffat, and Bell in [11]. We also tested the binary interpolative compression method of Moffat and Stuiver [8]. This code was explicitly designed to exploit locality in inverted indices, so it gained the most from our algorithm.

We did not count the cost of storing the sizes of each term since that cost would be invariant across all coding schemes. We did count the cost of storing an arithmetic table for arithmetic coding, but this cost was negligible compared to the cost of storing the bulk of the data.

Testing. To test our algorithm we used the ad-hoc TREC indexing data, disks 4 and 5 (excluding the Congressional Record). This data contained 527094 documents and 747990 distinct words, and occupied about one gigabyte of space when uncompressed. We tested three different orderings of the data in combination with the difference codes described above. First, we tested a random permutation of the document numbers as a baseline for comparison. Second, we tested the default ordering from the TREC database. We noted that this was already a significant improvement over a random ordering, indicating that there is considerable locality inherent in the TREC database. Third, we tested the ordering produced by our algorithm. Results are shown in Figure 3.

	Random	Identity	Ordered
Binary	20.0	20.0	20.0
Delta	7.52	6.46	5.45
Golomb	5.79	5.77	5.78
Arith	6.82	6.03	5.19
Interp	5.89	5.29	4.53

Figure 3: The improvement (in bits per edge) our algorithm offers for different coding schemes using disks 4 and 5 of the TREC database.

Index Size	Random	Identity	Ordered	Improvement overRandom	Improvement over Identity
32943	5.73	5.44	4.87	14.9%	10.4%
65886	5.75	5.43	4.78	16.9%	12.0%
131773	5.77	5.41	4.71	18.4%	13.0%
263547	5.78	5.36	4.63	19.9%	13.7%
527094	5.79	5.29	4.53	21.8%	14.4%

Figure 4: The improvement offered by our algorithm increases as the size of the index (measured in documents) increases.

Analysis. The Golomb code is near-optimal for the encoding of randomly distributed data, and in fact it was the best code for the Random ordering. However, the Golomb code is not convex, so it does not benefit from locality.

The locality inherent in the TREC database made the interpolative code the most efficient code for the identity ordering. Interpolative coding used 5.29 bits per edge, an improvement of about 8.6% over the best encoding with a random document ordering.

Using the ordering produced by our algorithm, however, the interpolative code needed an average of only 4.53 bits per edge to encode the data - a 21.8% improvement over the best coding of a random ordering, and a 14.4% improvement over the best coding of an identity ordering.

Index size. To measure the effect of index size on our algorithm, we tested our algorithm on various subsets of the full index. These subsets were formed by evenly subsampling documents from the full dataset. For each subset we evaluated the best compression using a random, identity, or ordered permutation of the documents. The random permutation was best coded with a Golomb code; the identity and ordered permutations were coded with interpolative codes. Figure 4 shows the results of our tests. Interestingly, the improvement offered by our algorithm increases as the size of the index increases.

		τ						
	Rand	1	2	5	10	15	20	40
Time(s)		51.81	81.62	120.7	163.7	196.4	225.0	304.8
Delta	7.44	6.56	6.04	5.95	5.94	5.90	5.89	5.88
Arith	6.73	6.11	5.69	5.62	5.62	5.58	5.57	5.56
Interp	5.81	5.29	4.97	4.89	4.87	4.86	4.86	4.85

Figure 5: The performance (in bits per edge) of different values of τ on one-sixteenth of the TREC indexing data.

		ρ				
	Rand	.75	.5	.25	.1	0
Time(s)		70.07	60.59	163.7	454.8	518.9
Delta	7.44	6.27	6.05	5.94	5.83	5.83
Arith	6.73	5.88	5.70	5.61	5.52	5.52
In terp	5.81	5.07	4.95	4.87	4.83	4.84

Figure 6: The performance (in bits per edge) of different values of ρ on one-sixteenth of the TREC indexing data. Note that our algorithm's running time is greater with $\rho = .75$ than with $\rho = .5$. This is because the aggressive subsampling results in unbalanced partitions, increasing the recursion depth of the algorithm.

Parameter Tuning. Our algorithm uses two parameters. The first parameter, τ, is a threshold which determines how sensitive our BUILD-GRAPH algorithm is to term size. If a term t_i has $|t_i| > \tau$, our algorithm will still consider it when calculating cosine measures, but will not add any edges to the similarity graph because of it.

T able5 shows the performance of our algorithm (on a subset of the full dataset containing one-sixteenth as many documents) with different v alues of τ. Choosing τ to be less than 5 causes too few edges to be included in the similarity graph, but increasing τ beyond that was not beneficial on the index we studied. We chose $\tau = 10$ to be safe.

The second parameter, ρ, determines how aggressively our algorithm subsamples the data. On an index of size n, the algorithm extracts one out of every $\lfloor n^\rho \rfloor$ elements to build a subindex. T able6 shows the performance of our algorithm with different v alues ofρ. Our algorithm does not perform too badly even with a very large ρ, but there is still a clear tradeoff between time, space and quality. We chose $\rho = .25$ in our experiments as a suitable balance between these concerns.

Graph Compression. Our algorithm can also be used to enhance the performance of difference coding in graph compression. In graph compression, for each vertex of the graph, an algorithm stores an adjacency list of the v ertices that share an edge with that vertex. The vertices are numbered, so it is only natural to apply a difference code to compress each list. If we view the vertices as terms and the adjacency lists as

Code	Random	Identity	Clustered	Ordered
Binary	18.0	18.0	18.0	18.0
Delta	17.5	4.92	4.58	4.52
Golomb	13.3	12.7	12.4	12.6
Arith	14.4	4.32	3.82	3.75
Interp	13.4	5.83	5.66	5.58

Figure 7: The performance of our algorithm on the TREC-8 WT2g web track. The "Clustered" column describes the performance of our algorithm without the final rotation step.

posting lists, we can apply our clustering technique to renumber the vertices of the graph.

To test our clustering technique on graph data we used another TREC dataset: the TREC-8 WT2g web data track. That track can be represented as a directed graph on 247428 web pages, where hyperlinks are edges. For best compression, we stored the in-edges (rather than the out-edges) of each vertex in our adjacency lists. The number of in-edges for each vertex was more variable than the number of out-edges, meaning that some adjacency lists were very dense and thus compressed very well. The performance of our algorithm on the in-link representation is shown in Table 7.

6 Future Work

One interesting avenue for future work would be to find a more advanced rotation scheme (using simulated annealing) for the cluster tree. This would improve the quality of the orderings produced by our algorithm.

Another avenue would be to investigate an alternative similarity measure to the cosine measure. When dealing with a pure document-term index, the cosine similarity measure is quite good. However, when we apply our clustering algorithm to graph reordering as described above, the cosine measure does not capture all of the details from the original graph. In particular, the cosine measure does not necessarily assign any similarity to two vertices that share an edge: in order to be similar, two vertices must each have an edge to a third vertex. One might imagine other similarity measures that took into account factors such as the edge distance between two vertices. For example, Adler and Mitzenmacher [1] take the distance between two documents to be the cost (in bits) of coding the difference between the two.

On graphs (such as meshes and planar graphs) with high locality, it might also be interesting to investigate alternative forms of clustering. An algorithm could apply a form of bottom-up clustering to the original graph in which it sequentially collapsed edges and merged the nodes of the graph into multinodes. If a way could be found to do this efficiently, this could be a more powerful technique than our top-down clustering.

References

[1] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Proceedings of IEEE Data Compression Conference (DCC)*, Mar. 2001.

[2] A. Bookstein, S. Klein, and T. Raita. Markov models for clusters in concordance compression. In *Proc. IEEE Data Compression Conference*, pages 116–125, Mar. 1994.

[3] A. Bookstein, S. Klein, and T. Raita. Modeling word occurrences for the compression of concordances. In *Proc. IEEE Data Compression Conference*, page 462, Mar. 1995.

[4] A. Borodin, R. Ostrovsky, and Y. Rabani. Subquadratic approximation algorithms for clustering problems in high dimensional spaces. In *ACM Symposium on Theory of Computing*, pages 435–444, July 1999.

[5] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Sixth Int'l. World Wide Web Conference*, pages 391–404, Cambridge, July 1997.

[6] S. Golomb. Run-length encodings. *IEEE Transactionson Information Theory, IT*, 12:399–401, July 1966.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, 1995.

[8] A. Moffat and L. Stuiver. Exploiting clustering in inverted file compression. In *Proc. Data Compression Conference*, pages 82–91, Mar. 1996.

[9] E. Voorhees and e. D.K. Harman. Overview of the eighth text retrieval conference (trec-8). In *Proceedingsof the Eighth TextREtrieval Conference (TREC-8)*, pages 1–24, 1999.

[10] D. Wishart. Efficient hierarchical cluster analysis for data mining and knowledge discovery .*Computer Science and Statistics*, 30:257–263, July 1998.

[11] I. H.Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images (Second Edition).* Morgan Kaufmann Publishing, San Francisco,1999.