Assigning Identifiers to Documents to Enhance the Clustering Property of Fulltext Indexes

Fabrizio Silvestri
Dipartimento di Informatica
Università di Pisa
Italy
fasilves@di.unipi.it

Salvatore Orlando
Dipartimento di Informatica
Università di Venezia - Mestre
Italy
orlando@dsi.unive.it

Raffaele Perego
ISTI - CNR
Istituto di Scienza e Tecnologie
Informatiche (A. Faedo)
Pisa, Italy
r.perego@isti.cnr.it

ABSTRACT

Web Search Engines provide a large-scale text document retrieval service by processing huge *Inverted File* indexes. Inverted File indexes allow fast query resolution and good memory utilization since their d-gaps representation can be effectively and efficiently compressed by using variable length encoding methods. This paper proposes and evaluates some algorithms aimed to find an assignment of the document identifiers which minimizes the average values of d-gaps, thus enhancing the effectiveness of traditional compression methods. We ran several tests over the Google contest collection in order to validate the techniques proposed. The experiments demonstrated the scalability and effectiveness of our algorithms. Using the proposed algorithms, we were able to sensibly improve (up to 20.81%) the compression ratios of several encoding schemes.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software Performance evaluation—*efficiency and effectiveness*; E.4 [**Data**]: Coding and Information Theory Data compaction and compression

General Terms

Algorithms, Experimentation, Performance

Keywords

Clustering property, Index Compression, Web Search Engines, Document Identifier Assignment

1. INTRODUCTION

Compressing the huge index of a Web Search Engine (WSE) entails a better utilization of memory hierarchies and thus a lower query processing time [11]. During the last years several works addressed the problem of index compression. The majority of them focused on devising effective and efficient methods to encode the document identifiers (*DocIDs*) contained in the posting lists of Inverted File (IF) indexes [11, 9, 1, 2, 14]. Since posting lists are ordered sequences of integer DocID values, and are usually accessed by scanning them from the beginning, these lists are stored as sequences of d-gaps, i.e. differences between successive DocID values. d-gap lists are then compressed by using variable-length encodings, thus representing smaller integers in less space than larger ones. Variable-length encoding schemes can be **bitwise**, or **bytewise**:

- In bitwise schemes, the list of integers is stored as a sequence of variable-length codewords, each composed of a variable number of bits. Well-known bitwise schemes include: *Elias' gamma*, *Delta*, *Golomb-Rice* [14], and *Binary Interpolative* coding [9]. Bitwise codes, in general, achieve very good compression ratios. The main of these methods is the relatively high decoding time, which may negatively impact on the query processing performance of the system. To overcome this drawback, Anh and Moffat recently proposed a very effective bitwise encoding schema, which enhances considerably the decoding performance [2].

- In bytewise encoding, each integer is represented using a fixed and integral number of bytes. In its simplest form, the seven least significant bits of each byte are used to encode an integer, while the most significant bit is used as a sort of "continuation bit", to indicate the existence of following bytes in the representation of the integer. An effective bytewise method, where word-alignment is retained, even at the cost of some bits wasted within each word, has been recently proposed by Anh and Moffat [1]. Bytewise codes have low decoding time, but are, in general, less effective than bitwise ones.

Since small d-gaps are much more frequent than large ones within postings lists, such variable-length encoding schemes allow IF indexes to be represented concisely. This feature of posting lists is called *Clustering property*, and is passively exploited by compression algorithms. However, by permuting DocIDs in a way that increases the frequency of small d-gaps, we may likely enhance the effectiveness of any variable-length encoding schema. Only three works previously addressed this possibility [12, 3, 13].

Shieh *et al.* [12] proposed a DocID reassignment algorithm adopting a Travelling Salesman Problem (TSP) heuristic. A *similarity* graph is built by considering each document of the collection as a vertex, and by inserting an edge between any pair of vertexes whose associated documents share at least one term. Moreover, edges are weighted by the number of terms shared by the two documents. The TSP heuristic algorithm is then used to find a cycle in the *similarity* graph having maximal weight and traversing each vertex exactly once. The suboptimal cycle found is finally broken at some point, and the DocIDs are reassigned to the documents according to the ordering established. The rationale is that since the cycle preferably traverses edges connecting documents sharing a lot of terms, if we assign close DocIDs to these documents, we should expect a reduction in the average value of d-gaps, and thus in the size of the compressed IF index. The experiments conducted demonstrated a good improvement in the compression ratio achieved. Unfortunately, this technique requires to store the whole graph in the main memory, and is too expensive to be used for real Web collections: the authors reported that reordering a collection of approximately $132,000$ documents required about 23 hours and 2.17 GBytes of main memory.

Also Blelloch and Blandford [3] proposed an algorithm (hereinafter called *B&B*) that permutes the document identifiers in order to enhance the clustering property of posting lists. Starting from a previously built IF index, a similarity graph G is considered where the vertexes correspond to documents, and the edges are weighted with the *cosine similarity* [4] measure between each pair of documents. The *B&B* algorithm recursively splits G into smaller subgraphs $G_{l,i} = (V_{l,i}, E_{l,i})$ (where l is the level, and j is the position of the subgraph within the level), representing smaller subsets of the collection. Recursive splitting proceeds until all subgraphs become singleton. The DocIDs are then reassigned according to a *depth-first* visit of the resulting tree. The main drawback of this approach is its high cost both in time and space: similarly to [12] it requires to store the whole graph G in the main memory. Moreover, the graph splitting operation is expensive, although the authors proposed some effective sampling heuristics aimed to reduce its cost. In [3] the results of experiments conducted with the TREC-8 ad hoc track collection are reported. The enhancement of the compression ratio obtained is significant, but execution times reported refer to tests conducted on a sub-collection of only $32,000$ documents. The paper addresses relevant issues, but due to its cost, also the *B&B* algorithm seems unfeasible for real Web collections.

In our opinion, another drawback of the previous approaches is that they focus on *reassigning* DocIDs appearing in a previously built IF index. The innovative point of our work is a bunch of DocID assignment techniques according to which DocIDs are assigned on the fly, during (and not after) the inversion of the document collection. In order to compute efficiently and effectively a good assignment, a new model to represent the collection of documents is needed. We propose a model that allows the assignment algorithm to be placed into the typical spidering-indexing life cycle of a WSE. Our model, hereinafter called *Transactional Model*, is based on the popular *bag-of-words* model, but it does not consider the *within-doc* frequency of the terms. In a previous work [13], we presented preliminary results relative to one of the algorithms discussed in this paper. Here we extend and complete the work by proposing and comparing several scalable and space-effective algorithms that can be used to assign DocIDs while the spidered collection is being processed by the Indexer. This means that when the index is actually committed on the disk, the new DocID assignment has been already computed. Conversely, the other methods proposed so far require that the IF index has already been computed before.

The rest of the paper is organized as follows. Section 2 introduces the DocID assignment problem more formally, and defines the goals of our algorithms. Section 3 introduces the model of document collection on which we will base the proposed algorithms. Section 4 presents and discusses our novel DocID assignment algorithms. The complexity in time and space is also evaluated. In Sections 5 and 6 we discuss the experimental performance of the algorithms, measured on the publicly available Google Programming Contest collection. Finally, Section 7 reports some conclusions along with a description of the research directions we plan to investigate in the near future.

2. THE ASSIGNMENT PROBLEM

Let $D = \{d_1, d_2, \ldots, d_{|D|}\}$ be a set of $|D|$ textual documents. Moreover, let T be the set of distinct terms t_i, $i = 1, \ldots, |T|$, present in D. Let G be a bipartite graph $G = (V, E)$, where the set of vertexes $V = T \cup D$ and the set of edges E contains arcs of the form (t, d), $t \in T$ and $d \in D$. An arc (t, d) appears in E if and only if term t is contained in document d.

Definition 1. *A* document assignment *for a collection of documents D is defined as a bijective function π:*

$$\pi : D \rightarrow \{1, \ldots, |D|\}$$

that maps each document d_i into a distinct integer identifier $\pi(d_i)$.

Definition 2. *Let l_i^π be the* posting list *associated with a term t_i. This list refers to both the set of vertexes $d_j \in D$ making up the neighborhood of vertex $t_i \in T$, and a given assignment function π:*

$$l_i^\pi = \langle \pi(d_j) | (t_i, d_j) \in E \rangle \quad i = 1..|T|$$

The posting list is ordered. More formally, if $l_{i,u}^\pi$ and $l_{i,v}^\pi$ are respectively the u-th and v-th elements of l_i^π, then $l_{i,u}^\pi < l_{i,v}^\pi$ iff $u < v$.

The compression methods commonly used to encode the various posting lists l_i^π exploit a *dgap*-based, representation of lists. Before encoding l_i^π, the list is thus transformed into a list of dgaps of the form $(l_{i,k+1}^\pi - l_{i,k}^\pi)$, i.e., gaps between successive document identifiers. Let $\bar{l}_i^\pi$ be the dgap-based representation of l_i^π.

Definition 3. *Let L^π be the set of all $\bar{l}_i^\pi$, $i = 1..|T|$ making up an IF index. We can define the size of the IF index encoded with m as:*

$$PSize_{L^\pi}^m = \sum_{i=1,\ldots,|T|} Encode_m\left(\bar{l}_i^\pi\right)$$

where $Encode_m$ is a function that returns the number of bits required to encode the list $\bar{l}_i^\pi$.

Definition 4. *The* document assignment problem *is an optimization problem, which aims to find the assignment*

π that yields the most compressible IF index with a given method m:

$$\min_{\pi} PSize^{m}_{L^{\pi}}$$

The above definition is very informal, since the optimality of an assignment also depends on the specific encoding method. However, we can observe that a practical simple measure of compressibility is the value of the *average gap* appearing in the various lists. When we reduce the average gap, the resulting IF results smaller almost independently from the encoding method actually adopted.

3. COLLECTION MODEL

Differently from the approaches proposed so far, our algorithms adopt a collection model which does not assume the existence of a previously built IF index. All the algorithms take in input a *transactional* representation of the documents belonging to the collection. Each document $d_i \in \mathcal{D}$ is represented by the set $\widetilde{d_i} \subseteq T$ of all the terms that the document contains. We denote the collection of documents in its transactional form with $\widetilde{\mathcal{D}} = \left\{\widetilde{d}_1, \widetilde{d}_2, \ldots, \widetilde{d}_{|D|}\right\}$. Basically, the transactional representation is very similar to the more popular bag-of-words one, but it does not consider the frequency of occurrences of the word within the documents (i.e. the *within-doc* frequency)

The transactional representation is actually stored by using a sort of digest scheme for each term. That is, for each $\widetilde{d_i}$ we store a list of integers obtained from the digest of the terms contained within. In our case, for each term we proceed as follows: we compute the MD5 digest [10] and pick the first four bytes of the computed digest. Since the MD5 features a very low collision rate for arbitrary long texts, it is very likely that it remains true even considering just the first four bytes of the digest. To support our claim, we tested this digest scheme on the terms contained in the Google collection. We measured a collision ratio of 0.0006083, i.e. roughly a collision every thousand distinct terms.

Finally, we used the *Jaccard* measure [7] to evaluate document similarity. The Jaccard measure is a well known metric used in cluster analysis to estimate the similarity between two generic objects described by the presence or absence of attributes. It counts the number of attributes common to both objects, and divides this number by the number of attributes owned by at least one of them. More formally, the *Jaccard* measure used to measure the similarity between two distinct documents $\widetilde{d_i}$, and $\widetilde{d_j}$ of $\widetilde{\mathcal{D}}$ is given by:

$$jaccard\_measure(\widetilde{d_i}, \widetilde{d_j}) = \frac{|\widetilde{d_i} \cap \widetilde{d_j}|}{|\widetilde{d_i} \cup \widetilde{d_j}|}$$

4. OUR ALGORITHMS

From a first analysis of the problem we could devise two different assignment schemes:

- *top-down assignment*: we start from the collection as a whole, and we recursively partition it by assigning, at each level, similar documents to the same partition. At the end of this partitioning phase a merging phase is performed until a single and ordered group of documents is obtained. The assignment function π is then deduced by the ordering of this last single group. This is the approach also followed by *B&B*. Within this scheme we propose two different algorithms which will be discussed in the following: Transactional *B&B* and *Bisecting*;
- *bottom-up assignment*: we start from a flat set of documents and extract from this set disjoint sequences containing similar documents. Inside each sequence the documents are ordered, while we do not make any assumption on the precedence relation among documents belonging to different sequences. The assignment function π in this case is deduced by first considering an arbitrary ordering of the produced sequences and then the internal ordering of the sequences themselves. In our case to order the produced sequences we simply consider the same order in which the sequences are produced by the algorithms themselves. Within this approach we propose two different algorithms: single-pass *k-means* and *k*-scan.

4.1 Top-down assignment

In the top-down scheme we start from the set $\widetilde{D}$. The general scheme of our top-down algorithms is the following (see Algorithm 1):

1. *center selection* (steps 3-5 of algorithm 1): according to some heuristic H, we select two (groups of) documents from $\widetilde{D}$ which will be used as partition representatives during the next step;
2. *redistribution* (steps 6-18) : according to their similarity to the centers, we assign each unselected document to one of the two partitions $\widetilde{D}'$ and $\widetilde{D}''$. Actually, we adopt a simple heuristic which consists in assigning exactly $\frac{|\widetilde{D}|}{2}$ documents to each partition in order to equally split the computational workload among the two partitions;
3. *recursion* (steps 19-20): we recursively call the algorithm on the two resulting partitions until each partition becomes a singleton;
4. *merging* (steps 21-25): the two partitions built at each recursive call are merged (operator $\oplus$) bottom-up thus establishing an ordering ($\preceq$) between them. The precedence relation $\preceq$ is obtained by comparing the borders of the partitions to merge ($\widetilde{D}'$ and $\widetilde{D}''$) and, according to the distance measure adopted, we put $\widetilde{D}'$ before $\widetilde{D}''$ if the similarity between the last document(s) of $\widetilde{D}'$ and the first document(s) of $\widetilde{D}''$ is greater than the similarity computed by swapping the two partitions.

It is also possible to devise a general cost scheme for such top-down algorithms.

Claim 1. *Let $\widetilde{D}$ be a collection of documents. The cost of our top-down assignment algorithms is*

$$T\left(|\widetilde{D}|\right) = O\left(|\widetilde{D}| \log\left(|\widetilde{D}|\right)\right)$$

Proof. Let σ be the cost of computing the Jaccard distance between two documents, and τ the cost of comparing two Jaccard measures. Computing the Jaccard similarity mainly consists in performing the intersection of two sets,

Algorithm 1 *TDAssign($\widetilde{D}, H$)*: the generic *top-down* assignment algorithm.

```
1: Input:
     • The set $\widetilde{D}$.
     • The function $H$ used to select the initial documents to form the centers of mass of the partitions.
2: Output:
     • An ordered list representing an assignment function $\pi$ for $\widetilde{D}$.
3: $\left(\widetilde{D}', \widetilde{D}''\right) = H\left(\widetilde{D}\right)$;
4: $c_1 = \text{center\_of\_mass}\left(\widetilde{D}'\right)$;
5: $c_2 = \text{center\_of\_mass}\left(\widetilde{D}''\right)$;
6: for all not previously selected $d \in \widetilde{D} \setminus \left(\widetilde{D}' \cup \widetilde{D}''\right)$ do
7:   if $\left(\left|\widetilde{D}'\right| \geq \frac{|\widetilde{D}|}{2}\right) \vee \left(\left|\widetilde{D}''\right| \geq \frac{|\widetilde{D}|}{2}\right)$ then
8:     Assign $d$ to the smallest partition;
9:   else
10:    $dist_1 = \text{distance}(c_1, d)$;
11:    $dist_2 = \text{distance}(c_2, d)$;
12:    if $dist_1 < dist_2$ then
13:      $\widetilde{D}' = \widetilde{D}' \cup \{d\}$;
14:    else
15:      $\widetilde{D}'' = \widetilde{D}'' \cup \{d\}$;
16:    end if
17:  end if
18: end for
19: $\widetilde{D'}_{ord}$ = TDAssign($\widetilde{D'}, H$);
20: $\widetilde{D''}_{ord}$ = TDAssign($\widetilde{D''}, H$);
21: if $\widetilde{D'}_{ord} \preceq \widetilde{D''}_{ord}$ then
22:   $\widetilde{D}_{ord} = \widetilde{D'}_{ord} \oplus \widetilde{D''}_{ord}$
23: else
24:   $\widetilde{D}_{ord} = \widetilde{D''}_{ord} \oplus \widetilde{D'}_{ord}$
25: end if
26: return $\widetilde{D}_{ord}$;
```

while comparing the similarity of two document only requires to compare two floats. We thus have that $\sigma \gg \tau$. Furthermore, let C_H be the cost of the heuristic H used to select the initial centers, and C_S be the cost of the *merging* step.

At each iteration, the top-down algorithm computes the initial centers. Then it computes at most $|\widetilde{D}| - 2$ Jaccard distances in order to assign each document to the right partition. The total cost of this phase at each iteration is thus bounded by: $\sigma\left(|\widetilde{D}| - 2\right) + C_H$.

At the end of the *center selection* and *distribution* phases, the top-down algorithm proceeds by calling recursively itself on the two equally sized sub-partitions obtained so far (*recursion* step) and then proceeds to order and merge the two partitions obtained (*merging* step).

The total cost of the algorithm is thus given by the following recursive equation:

$$T\left(|\widetilde{D}|\right) = C_H + \sigma\left(|\widetilde{D}| - 2\right) + 2T\left(\frac{|\widetilde{D}|}{2}\right) + C_S \qquad (1)$$

This equation corresponds to the well known:

$$T\left(|\widetilde{D}|\right) = O\left(|\widetilde{D}| \log\left(|\widetilde{D}|\right)\right)$$

$\square$

Furthermore, we can compute the space occupied by the top-down assignment algorithm.

CLAIM 2. *The space occupied by our top-down assignment algorithm is given by*

$$S\left(|\widetilde{D}|\right) = O\left(|\widetilde{D}| \log\left(|\widetilde{D}|\right)\right)$$

PROOF. Since we need to keep, at each level, a bit indicating the assigned partition, we need in total $S_{rec}\left(|\widetilde{D}|\right) = |\widetilde{D}| + S_{rec}\left(\frac{|\widetilde{D}|}{2}\right) = O\left(|\widetilde{D}| \log |\widetilde{D}|\right)$ bits to store the partition assignment map. In practice $S_{rec}\left(|\widetilde{D}|\right)$ defines the total space occupied by the partitions D' and D'' at all levels.

Now, let $|\widetilde{S}|$ be the average length of a document. The total space of the algorithm is thus given by:

$$\begin{aligned} S\left(|\widetilde{D}|\right) = \quad & |\widetilde{S}||\widetilde{D}| + && \text{documents} \\ & +2S_{rec}\left(|\widetilde{D}|\right) && \text{space for } D' \text{ and } D'' \end{aligned}$$

We can get rid of the linear term thus obtaining:

$$S\left(|\widetilde{D}|\right) = O\left(|\widetilde{D}| \log\left(|\widetilde{D}|\right)\right) \qquad (2)$$

$\square$

As for the time, also the space complexity of the algorithm is super-linear in the number of documents processed. In practice, anyway, this is not a correct assertion. In fact the linear term dominates the $n \log n$ one until $4 \cdot \log n \leq 1000$. The last value for which the inequality holds is given by $\log n \leq 250 \Leftrightarrow n \leq 2^{250}$. Obviously the size of the whole Web is considerably smaller then 2^{250} documents!

We designed two different top-down algorithms: TRANSACTIONAL *B&B* and *Bisecting*.

4.1.1 TRANSACTIONAL B&B

The TRANSACTIONAL *B&B* algorithm is basically a porting under our model of the algorithm described in [3]. We briefly recall how the original *B&B* algorithm works. It starts by computing a sampled similarity graph: it chooses a document out of $|\widetilde{D}|^{\rho}$ (ρ is the document sampling factor $0 < \rho < 1$) only considering terms appearing in less than τ documents. After this reduced similarity graph has been built, it applies the *Metis* graph partitioning algorithm [8], which splits the graph in two equally sized partitions. The algorithm than proceeds with the *redistribution*, *recursion*, and *merging* steps of the generic top-down algorithm. However, since in our model we do not have an IF index previously built over the document collection, we cannot know which terms appear in less than τ documents, and thus we did not introduce sampling over the maximum term frequency as in the original implementation.

In TRANSACTIONAL *B&B* the cost C_H at each iteration is thus given by the cost of picking up a subset of documents with a sampling factor equal to ρ, plus the cost of building the distance graph over this subset and computing the *Metis* algorithm over this graph.

4.1.2 Bisecting

The second algorithm we propose is called *Bisecting*. In this algorithm we adopt a *center selection* step which simply consists of uniformly choosing *two* random documents as centers. The cost of the *centers selection* step is thus reduced considerably. The algorithm is based on the simple observation that, since in TRANSACTIONAL *B&B* the cost C_H may be high, the only way to reduce it is to choose a low sampling parameter ρ, thus selecting at each iteration a very small number of documents as centers of the partitions. Thus we thought to just get rid of the first three phases, i.e. sampling, graph building, and *Metis* steps.

4.2 Bottom-up assignment

These algorithms consider each document of the collection separately, and proceed by progressively grouping together similar documents. Our bottom-up algorithms thus produce a set of non-overlapping sequences of documents.

The two different assignment algorithms presented here are both inspired by the popular *k-means* clustering algorithm [5]:

- a single-pass *k-means* algorithm;
- k-scan which is based on a centroid search algorithm which adapts itself to the characteristics of the processed collection.

4.2.1 Single-pass k-means

k-means [5], is a popular iterative clustering techniques which defines a *Centroid Voronoi Tessellation* of the input space. The k-means algorithm works as follows. It initially chooses k documents as cluster representatives, and assigns the remaining $|\widetilde{D}| - k$ documents to one of these clusters according to a given similarity metric. New centroids for the k clusters are then recomputed, and all the documents are reassigned according to their similarity with the new k centroids. The algorithm iterates until the position of the k centroids become stable. The main strength of this algorithm is the $O\left(|\widetilde{D}|\right)$ space occupancy. On the other hand, computing the new centroids is expensive for large values of $|\widetilde{D}|$, and the number of iterations required to converge may be high. The single-pass k-means consists of just the first pass of this algorithm where the k centers are chosen using the technique described in [6]: *Buckshot*. We will not describe here the *Buckshot* technique, the only thing to keep into account is that the complexity of this step do not influence the theoretical linear performance of k-means which remains $O\left(k|\widetilde{D}|\right)$. Since th e$k$-means algorithm does not produce ordered sequences but just clusters, the internal order of each cluster is given by the insertion order of documents into each cluster.

4.2.2 k-scan

The other bottom-up algorithm developed is k-scan. It resembles to the k-means one. It is, indeed, a simplified version requiring only k steps. At each step i, the algorithm selects a document among those not yet assigned and uses it as centroid for the i-th cluster. Then, it chooses among the unassigned documents the $\frac{|\widetilde{D}|}{k} - 1$ ones most similar to the current centroid and assign them to the i-th cluster. The time and space complexity is the same as the single-pass k-means one and produces sets of ordered sequences of documents. Such ordering is exploited to assign consecutive DocIDs to consecutive documents belonging to the same sequence. The k-scan algorithm is outlined in Algorithm 2. It takes as input parameters the set $\widetilde{D}$, and the number k of sequences to create. It outputs the ordered list of all the members of the k clusters. This list univocally defines π, an assignment of $\widetilde{D}$ minimizing the average value of the d-gaps.

Algorithm 2 The k-scan assignment algorithm.

```
 1: Input:
      • The set $\widetilde{D}$.
      • The number $k$ of sequences to create.
 2: Output:
      • $k$ ordered sequences representing an assignment $\pi$ of $\widetilde{D}$.
 3: sort $\widetilde{D}$ by descending lengths of its members;
 4: $c_i = \emptyset \quad i = 1, \ldots, k$;
 5: for $i = 1, \ldots, k$ do
 6:     $current\_center = longest\_member\left(\widetilde{D}\right)$
 7:     $\widetilde{D} = \widetilde{D} \setminus current\_center$
 8:     for all $\widetilde{d_j} \in \widetilde{D}$ do
 9:         $sim[j] = compute\_jaccard\left(current\_center, \widetilde{d_j}\right)$
10:     end for
11:     $M = select\_members(sim)$
12:     $c_i = c_i \oplus M$
13:     $\widetilde{D} = \widetilde{D} \setminus M$
14:     $c_i = c_i \oplus current\_center$
15:     $dump(c_i)$
16: end for
17: return $\bigoplus_{i=1}^{k} c_i$;
```

The algorithm performs k scans of $\widetilde{D}$. At each scan i, it chooses the longest document not yet assigned to a cluster as current center of cluster c_i, and computes the distances between it and each of the remaining unassigned documents. Once all the similarities have been computed, the algorithm selects the $\left(\left|\widetilde{D}\right|/k\right) - 1$ documents most similar to the current center by means of the procedure reported in Algorithm 3, and put them in c_i. It is worth noting that when two documents result to have the same similarity, the longest one is selected. In fact, since the first DocID of each posting list has to be coded as it is, assigning smaller identifiers to documents containing a lot of distinct terms, maximizes the number of posting lists starting with small DocIDs.

The complexity of k-scan in terms of number of distance computation and in space occupied is given by the following two claims.

CLAIM 3. *The complexity of the k-scan algorithm is:*

$$T(|\widetilde{D}|, k) = O\left(|\widetilde{D}|k\right)$$

PROOF. Since we are focusing on the number of distance computations, the initial ordering step (at point 3) of Algorithm 2 should not be considered when computing the complexity of the algorithm. Let σ be the cost of computing the Jaccard similarity between two documents, and τ the cost of comparing two Jaccard measures. Computing the Jaccard similarity mainly requires to intersect two sets,

Algorithm 3 The *select_members* procedure.

```
1: Input:
     • An array sim: sim[j] contains the similarity between current_center and $S_j$.
2: Output:
     • The set of the $\left(\left|\widetilde{D}\right|/k\right)-1$ documents more similar to current_center.
3: Initialize a min_heap of size $\left(\left|\widetilde{D}\right|/k\right)-1$
4: for $i = 1, \ldots, |\widetilde{D}|$ do
5:   if (sim[i] > sim[heap_root()]) OR ((sim[i] = sim[heap_root()]) AND (length(i) > length(heap_root())))
     then
6:     heap_insert(i)
7:   end if
8: end for
9: $M = \emptyset$
10: for $i = 1, \ldots, \left(\left|\widetilde{D}\right|/k\right)-1$ do
11:   $M = M \oplus$ heap_extract()
12: end for
13: return M
```

while comparing two similarity measures only requires to compare two floats. We thus have that $\sigma \gg \tau$.

At each iteration, k-scan computes $|\widetilde{D}|-i\frac{|\widetilde{D}|}{k}$ Jaccard measures. The total cost of this phase at each iteration i is thus:

$$\sigma\left(|\widetilde{D}|-i\frac{|\widetilde{D}|}{k}\right)$$

Once oll the entries in the vector of similarities *sim* have been computed, k-scan calls the *select_members* procedure which performs $|\widetilde{D}|-i\frac{|\widetilde{D}|}{k}$ insertions into a heap of size $\frac{|\widetilde{D}|}{k}-1$. Since an insertion is actually performed only if the element in the root of the heap is smaller than the element to be inserted, we should scale down the cost by a factor $\tau' \ll 1$. The total time spent in *select_members* is thus:

$$\omega\left(|\widetilde{D}|-i\frac{|\widetilde{D}|}{k}\right)\cdot\log\left(\frac{|\widetilde{D}|}{k}-1\right) \qquad \omega=\tau\cdot\tau'.$$

The total time spent by the algorithm is thus given by:

$$T(|\widetilde{D}|,k)=\sum_{i=0}^{k-1}T'$$

where

$$T'=\sigma\left(|\widetilde{D}|-i\frac{|\widetilde{D}|}{k}\right)+\left(\omega\left(|\widetilde{D}|-i\frac{|\widetilde{D}|}{k}\right)\log\left(\frac{|\widetilde{D}|}{k}-1\right)\right).$$

Since $\sigma \gg \tau \gg \omega$ we have that:

$$T(|\widetilde{D}|,k)\approx\sum_{i=0}^{k-1}\sigma\left(|\widetilde{D}|-i\frac{|\widetilde{D}|}{k}\right)=\sigma|\widetilde{D}|\left(\frac{k+1}{2}\right)=O\left(|\widetilde{D}|k\right)$$

□

Claim 4. *The space occupied by the k-scan algorithm is:*

$$S\left(|\widetilde{D},k|\right)=O\left(|\widetilde{D}|\right)$$

Proof. Let $|\overline{S}|$ be the average length of a document. The k-scan algorithm thus uses $|\overline{S}||\widetilde{D}|$ words for storing the documents, $8|\widetilde{D}|$ for the array of similarities and $4\frac{|\widetilde{D}|}{k}$ for the heap data structure used by Algorithm 3.

The total space is thus given by

$S\left(\lvert\widetilde{D},k\rvert\right)=$	$\lvert\overline{S}\rvert\lvert\widetilde{D}\rvert+$	documents
	$+8\lvert\widetilde{D}\rvert+$	array of similarities
	$+4\frac{\lvert\widetilde{D}\rvert}{k}+$	the heap data structure
	$=O\left(\lvert\widetilde{D}\rvert\right)$	

□

5. EXPERIMENTAL SETUP

To assess the performance of our algorithms we tested them on a real collection of Web documents, the publicly available Google Programming Contest collection[1]. The main characteristics of this collection are:

- it contains about 916,000 documents coming from real web sites;
- it is monolingual;
- the number of distinct terms is about 1,435,000.

On the considered collection we performed a preprocessing step consisting in the transformation of the documents considered in the *transactional* model.

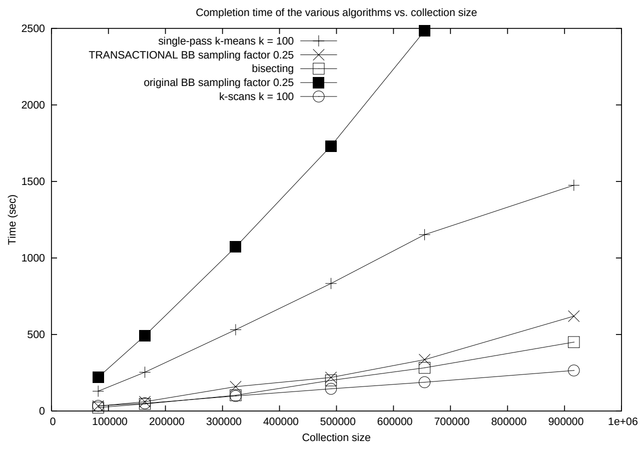
For each method proposed we evaluated: the completion time, the space occupied, and the compression ratios achieved after the assignment. The effectiveness gains resulting by adopting Binary Interpolative [9], Gamma [14] and Variable Byte [11] encoding methods were evaluated. We ran our tests on a Xeon 2GHz PC equipped with 1GB of main memory, and an Ultra-ATA 60GB disk. The operating system was Linux.

6. COMPARISONS

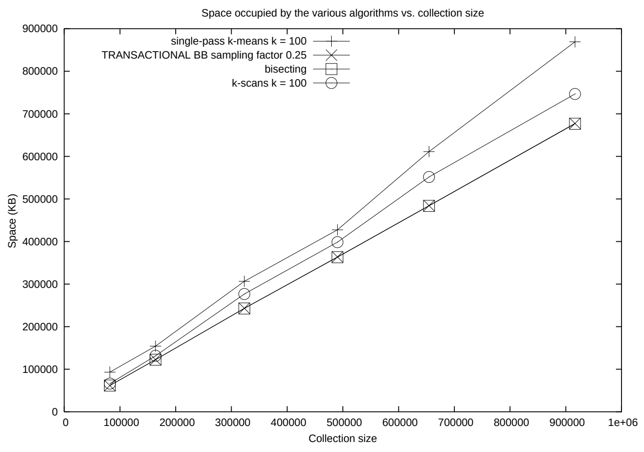
In Figure 1 the performance in terms of completion time (a), and space consumed (b) are shown. All the time reported are the actual times taken by all the algorithms to finish their operations and do not include I/O.

From Figure 1.(a) we can draw several important conclusions. First of all, the execution time of the original *B&B* algorithm is remarkably higher than the time spent by all our algorithms. In particular, on the whole Google contest collection (i.e. about 916,000 documents) the *B&B* algorithm ran out of memory before finishing its operations. Please note that the values plotted in the curve of the original *B&B* do not consider the time spent in preliminarly computing the input IF index. If we look at the curve related to the original implementation of the *B&B* algorithm we can observe an $n\log n$ behavior that is typical of top-down approaches described above. Looking at the curves of our algorithms, it is evident that the single-pass k-means algorithm is the one that takes the highest time to compute

[1]http://www.google.com/programming_contest



(a)



(b)

Figure 1: (a) Time (in seconds) and (b) Space (in KBytes) consumed by the proposed transactional assignment algorithms as a function of the collection size.

the assignment. On the other hand, the others transactional techniques have relatively low completion times. In particular the k-scan algorithm sensibly outperforms the others. Obviously, the linear behavior exhibited by this algorithm evidentiates that on large collections k-scan will remarkably outperforms the others which instead show a $n \log n$ trend.

Figure 1.(b) shows the space occupancy of our algorithms. Since the tests with the original *B&B* algorithm were performed by using the implementation kindly provided to us by the authors, we were not able to measure directly the space occupied by this algorithm. However the algorithm is memory consuming and, as said above, on the whole Google contest collection it ran out of memory. Obviously, also in our case we cannot fit completely in memory large collections. In those cases we can split the collections in several partitions and then proceed to reorder each partition separately. The curves plotted in Figure 1.(b) show that, as expected, TRANSACTIONAL *B&B* and *Bisecting* use the same amount of memory and exhibit a linear scale-up trend. This last fact follows directly from the observations made in Section 4 about the space complexity of the top-down approaches. Anyway, the space occupied by all our algorithms appears to grow linearly with respect to the collection size.

The DocIDs of four collections of different size, with up to $916,000$ documents, were assigned by exploiting the various algorithms. For each assignment we measured the compression performance. Table 1 reports the average number of bits required to represent each posting of the IF obtained after the DocID assighment with three popular encoding methods: *Interpolative, Gamma*, and *Variable Byte*. In all the cases we can observe a reduction in the average number of bits used with respect to a random DocID assignment, i.e., the baseline for comparisons reported in the first block of rows of the table. We can see that the original implementation of the *B&B* algorithm outperforms our algorithms. The gain in the compression performance of *B&B* is, in almost all the cases, about 10% which corresponds to ~ 0.5 bits saved for each posting. However, our methods spend remarkably less time than the *B&B* one. We can also observe that our methods are similar in terms of compression gain. For the largest collection the performance obtained is approximately the same for all the encoding methods and for all the assignment algorithms implemented. Moreover, we can note that the TRANSACTIONAL *B&B* algorithm obtains worse results than the original *B&B* algorithm. We think that this may depend on the term sampling which cannot be exploited by our TRANSACTIONAL *B&B*. (see Section 4). The higher gains in compression performance were obtained by using the Gamma algorithm. This is a very important result since a method which is very similar to Gamma was recently presented in [2]. This method is characterized by a very good compression performance and a relatively low decoding overhead, in some cases lower than those of the Variable Byte method. The results on the Gamma algorithm are thus very important to validate our approaches.

7. SUMMARY

In this paper we presented an analysis of several efficient algorithms for computing approximations of the optimal DocID assignment for a collection of textual documents. We have proved that our algorithms are a viable way to enhance the compressibility (up to 21%) of IF indexes.

The algorithms proposed operate following two opposite strategies: a top-down approach and a bottom-up approach. The first group includes the algorithms that recursively split the collection in a way that minimizes the distance of lexicographically closed documents. The second group contains algorithms which compute an effective reordering employing linear space and time complexities. Although our algorithms obtain gains in compression ratios which are slightly worse than those obtained by the *B&B* algorithm, their performance in terms of space and time are instead remarkably higher. Moreover, an appealing feature of our approach is the possibility of performing the assignment step on the fly, during the indexing process. As future work we plan to test the performance of our algorithms on some recently proposed encoding methods. In particular we would like to evaluate the method described in [2] for which we should be able to obtain good results. Furthermore, we want to investigate possible adaptations of the algorithms proposed to collections which change dynamically in the time.

8. ACKNOWLEDGMENTS

Assignment Algorithm	Collection Size	Bits per Postings		
		Interpolative	Gamma	Var Byte
Random assignment	81,875	6.52	8.96	9.67
	323,128	6.59	9.05	9.71
	654,535	6.61	9.08	9.73
	916,429	6.61	9.10	9.72
B&B	81,875	5.31	6.71	9.17
	323,128	5.18	6.58	9.20
	654,535	5.08	6.40	9.14
	916,429	N/A	N/A	N/A
TRANSACTIONAL *B&B*	81,875	5.56	7.20	9.29
	323,128	5.46	7.11	9.32
	654,535	5.54	7.19	9.35
	916,429	6.04	8.04	9.52
Bisecting	81,875	5.66	7.60	9.37
	323,128	5.62	7.53	9.33
	654,535	5.71	7.66	9.38
	916,429	6.10	8.23	9.52
single-pass k-means	81,875	5.60	7.27	9.26
	323,128	5.64	7.34	9.32
	654,535	5.67	7.44	9.32
	916,429	6.10	8.11	9.51
k-scan	81,875	5.53	7.25	9.20
	323,128	5.56	7.36	9.27
	654,535	5.66	7.54	9.33
	916,429	6.10	8.11	9.51

Table 1: Performance, as average number of bits used to represent each posting, as a function of the assignment algorithm used, of the collection size (no. of documents), and of the encoding algorithm adopted. The row labeled "Random assignment" reports the performance of the various encoding algorithms when DocIDs are randomly assigned.

We thank Guy E. Blelloch and Dan K. Blandford for kindly providing us with the source code of their reordering algorithm. We also thank Ronny Lempel, Paolo Ferragina, and Mirco Nanni for the many helpful discussions we had and for some helpful hints on the problem analysis. Finally, we would like to thank Alistair Moffat for its suggestions about how to improve this work. Moreover, we acknowledge the financial support of the *Italian Ministry of University and Research* which funded this work conducted within the ECD project.

9. REFERENCES

[1] V. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 2004. To appear.

[2] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. In K.-D. Schewe and H. Williams, editors, *Proc. 15th Australasian Database Conference*, Dunedin, New Zealand, Jan. 2004.

[3] D. Blandford and G. Blelloch. Index compression through document reordering. In IEEE, editor, *Proceedings of the Data Compression Conference (DCC'02)*. IEEE, 2002.

[4] C. Buckley. Implementation of the smart information retrieval system. Technical Report TR85–686, Cornell University, Computer Science Department, May 1985.

[5] S. Chakrabarti. *Mining the Web - Discovering Knowledge from Hypertext Data*. Morgan Kaufmann Publishers, San Francisco, 2003.

[6] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey. Scatter/gather: a cluster-based approach to browsing large document collections. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 318–329. ACM Press, 1992.

[7] W. B. Frakes and E. R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*, chapter Clustering Algorithms (E. Rasmussen). Prentice Hall, Englewood Cliffs, NJ, 1992.

[8] G. Karypis. Metis: Family of multilevel partitioning algorithms. http://www-users.cs.umn.edu/~karypis/metis/.

[9] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, July 2000.

[10] R. Rivest. Rfc 1321: The md5 algorithm.

[11] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted index for fast query evaluation. In *Proceedings of the 25rd annual international ACM SIGIR conference on Research and development in information retrieval*, 2002.

[12] W.-Y. Shieh, T.-F. Chen, J. J.-J. Shann, and C.-P. Chung. Inverted file compression through document identifier reassignment. *Information Processing and Management*, 39(1):117–131, January 2003.

[13] F. Silvestri, R. Perego, and S. Orlando. Assigning document identifiers to enhance compressibility of web search. In *Proceedings of the 19th Annual ACM Symposium on Applied Computing - Data Mining Track*, 2004.

[14] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes – Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco, second edition edition, 1999.