Almost-Constant-Time Clustering of Arbitrary Corpus Subsets

Craig Silverstein
Stanford University
Gates Hall, Stanford, CA 94305
(415)937-2853
csilvers@cs.stanford.edu

Jan O. Pedersen
Verity Inc.
894 Ross Dr., Sunnyvale, CA 94089
(408)542-2323
jpederse@verity.com

Almost-Constant-Time Clustering of Arbitrary Corpus Subsets

Abstract

Methods exist for constant-time clustering of corpus subsets selected via Scatter/Gather browsing [3]. In this paper we expand on those techniques, giving an algorithm for almost-constant-time clustering of arbitrary corpus subsets. This algorithm is never slower than clustering the document set from scratch, and for medium-sized and large sets it is significantly faster. This algorithm is useful for clustering arbitrary subsets of large corpora — obtained, for instance, by a boolean search — quickly enough to be useful in an interactive setting.

1 Introduction

Document clustering has emerged as an important tool for the presentation and navigation of document collections. For example, the Scatter/Gather browsing paradigm clusters documents into topic-coherent groups and presents descriptive textual summaries to the user [2]. Informed by the summaries, the user may select clusters, thereby forming a subcollection, for iterative examination. The clustering and reclustering is done on the fly, so that different topics are seen depending on the subcollection clustered. This is preferrable to navigating with respect to a static, global, topic hierarchy since the induced topics are specific to the document set under investigation [6]. However, since typical clustering algorithms are at least linear in the number of objects to be clustered (see e.g. [7]), the computational cost of these operations may be too great for interactive use if the document set is large.

The computational cost of on-the-fly, or *on-line*, document clustering may be greatly reduced by preprocessing the document collection. The general approach is to precompute, off-line, a global *cluster hierarchy*, a tree structure whose leaves are documents and whose internal nodes correspond to aggregates of similar documents. The cluster hierarchy is then used to accelerate on-line clustering. For example, Cutting, Karger, and Pedersen [3] develop a constant-time Scatter/Gather step, assuming that one starts with document subsets corresponding to nodes in a precomputed hierarchy.

Suppose one wants to cluster an entire collection into five groups. One could start at the root of a precomputed hierarchy and expand that node by replacing it with its children. These nodes could in turn be examined and the largest one expanded. The process could be iterated until the total number of nodes is equal to some predefined constant, say 50. These nodes correspond to either groups of documents or, for leaf nodes, individual documents. In either case, the node can be represented as a vector, namely the centroid of document vectors associated with the node. One could then apply a linear-time custering algorithm to these 50 vectors to produce a partition into five groups. This partition would be returned as the result of the operation. Since we would only cluster 50 objects, the clustering would take constant time regardless of the size of the collection.[1]

[1] Actually, for this to be formally true the objects clustered must have constant length. This is achieved by truncating centroids to a predefined size.

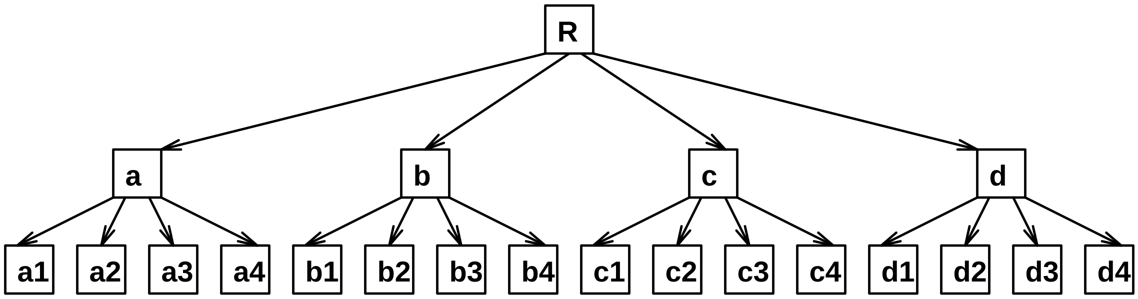

Figure 1: An example cluster hierarchy.

See Figure 1 for an example cluster hierarchy. R represents the entire corpus divided into 4 clusters a, b, c, and d which are presented to the user. Suppose the user selects a and d and desires a reclustering into four clusters. We expand clusters a and b to obtain 8 clusters, a_1, a_2, a_3, a_4, b_1, b_2, b_3, and b_4. We then run a linear-time clustering algorithm to cluster these 8 "pseudo-documents" into 4 new clusters. The algorithm might output a_1, $a_2 \cup a_4 \cup b_1$, $a_3 \cup b_2 \cup b_4$, b_3. These would be the 4 clusters presented to the user for the next interactive step. Note that clusters displayed to the user are always the union of nodes in the cluster hierarchy. The constant c determines how far down into the hierarchy we descend; the further we descend, the more complicated the unions become, and the less obvious the pre-computed clusters are in the final cluster set.

Though the process, as described, starts at the root of the precomputed hierarchy, it could just as easily start at an arbitrary node. Similarly, it could start with a set of nodes rather than a single node. This is the case, for example, if one selects one or more clusters, computed as above, for further expansion. Hence if one begins a Scatter/Gather process using this method, it can also apply to each subsequent iteration. By modifying the predefined constant (50 above) the user can trade off clustering time and the customization of the results to the input set, while keeping all calculations constant time.

Note that one could avoid the final clustering in each Scatter/Gather iteration by, instead of expanding to 50 nodes and then clustering back down to 5, merely expanding to 5 nodes and stopping. However, this would be equivalent to navigating a static, global, topic hierarchy. The extra expansion and clustering achieves a greater degree of customization to the input. This is especially desirable when the input set corresponds to more than one internal node of the precomputed hierarchy.

This constant-time clustering technique is constrained to document collections that correspond to sets of nodes in the precomputed hierarchy, such as arise when one performs Scatter/Gather on an entire collection. However, Scatter/Gather browsing is most useful as a tool for viewing search results, which are arbitrary corpus subsets [1]. Indeed, experiments indicate that searching followed by the presentation of clusters and selection of the most relevant one improves precision with respect to a baseline presentation of ranked documents [6]. The constant-time Scatter/Gather method does not directly apply; arbitrary corpus subsets cannot be trivially mapped to nodes in a precomputed hierarchy and hence cannot be easily "expanded".

One solution, presented in this paper, is to find a reasonable embedding of an arbitrary subcollection into an existing cluster hierarchy, run the constant-time clustering algorithm, and recover the relevant documents from the output clusters. We present a technique for doing so that is linear in theory but — since the time is dominated by the constant-time clustering — in practice effectively constant-time.

Section 2 presents the algorithm. Section 3 describes experimental results indicating that the near-constant-time method does not significantly degrade precision while Section 4 shows that it is in fact considerably faster than direct clustering methods.

2 Methodology

We begin with the with a precomputed cluster hierarchy, H, covering a document corpus C of size N. We also have an arbitrary set of documents $S \subseteq C$. The cluster hierarchy is a tree of cluster nodes with fan-out k and root r such that the union of the k children of a node has the same documents as the node itself. The cluster hierarchy is computed off-line, typically in $N \log N$ time by recursively applying a linear-time partitioning algorithm (see, for example, [3]). The goal is to cluster (partition) S into p groups.

Note that this algorithm will be similar in kind to an algorithm that takes S and clusters it from scratch. It is not expected that the algorithm will give better results than such a direct clustering algorithm; however, we expect it will give comparable results at a great advantage in speed.

The technique employed is similar to that described above for Scatter/Gather. We find a small number, m, of nodes in the cluster hierarchy that are "good" in some way, and we cluster them using a linear-time clustering algorithm. As long as the number of nodes we select is bounded, the clustering takes constant time.

Because the number of nodes in the cluster hierarchy may be large ($O(\log N)$ is reasonable), we cannot afford to look at all the nodes to find "good" ones. Instead we fan out from the top of the cluster hierarchy. We start with the root node of H and immediately replace it with its children. We examine the k nodes in our current node set and pick the "worst" one. We remove that "worst" node and replace it with its k children. We repeat the process on the $2k-1$ nodes now under consideration. As we describe later, we actually do not always include all k children, but rather pick a subset of the children. When we have collected m nodes, we stop the algorithm. We cluster the m result nodes into p clusters, treating each node as a single entity.

At this point we use a function I_S that gives for each node n in the cluster hierarchy the intersection between S and the $D(n)$, the documents represented by n:

$$I_S(n) = S \cap D(n).$$

We will show how to construct I_S in $|S| \log N$ time. We use this table to replace each result node n by $I_S(n)$, ridding ourselves of documents represented in the hierarchy that were not in S. See Table 1 for a more formal description of the algorithm.

All that remains is to show how to calculate I_S; to define C, the function that picks children; and to define f, the function that determines "bad" nodes of H.

Algorithm: SELECT NODES
Input: S, a set of documents. H, a cluster hierarchy. m, the maximum number of nodes we collect. A child function C and a goodness function f.
Output: P, a set of clusters.

1. Construct I_S. ($I_S(n)$ computes the intersection of S and $D(n)$.)
2. $T \leftarrow \{r\}$, the root of H.
3. **repeat**
   (a) $w \leftarrow f(S, T)$. (f returns the "worst" element of T according to some metric.)
   (b) $T \leftarrow T - \{w\} + C(w)$. ($C(w)$ returns some of the children of w in H.)

   until $|T| > m$.
4. Cluster T using a linear time clustering algorithm. We obtain P, a partition of T into disjoint sets.
5. Replace each node n in P by $I_S(n)$.
6. **return** P.

Table 1: The clustering algorithm for an arbitrary data set.

2.1 Calculating the intersection

To calculate I_S, the intersection of S and $D(n)$ for an arbitrary node n, we construct a table indexed by nodes. Each entry of the table is originally empty. For each document d in S, we use a precomputed auxiliary function, I_H, to find which nodes contain the document. This function doesn't depend on S and can be computed off-line at the same time H is computed by accumulating a table entry for each $d \in C$. Since H has constant fan-out, it has depth $\log N$ and thus each document is in $\log N$ nodes.

We add d to the table entry for each node in $I_H(d)$. This update takes time $O(\log N)$ per document, or time $O(|S| \log N)$ to construct the full table. The resulting table implements I_S.

2.2 Picking children

When replacing a node with its children, we can clearly avoid "empty" children, that is, children that do not contain any documents in S. "Singleton" children, that contain only one document from S, can also be dealt with specially. There is no need to include the entire node when we only care about one document in it: we might as well just take out the document and treat it as its own node.[2] In

[2]This is equivalent to replacing the child node by an appropiate leaf descendent.

Algorithm: SELECT NODES WITH CUTOFF
Input: S, a set of documents. H, a cluster hierarcy. m, the maximum number of nodes we collect. A child function C and a goodness function f. c, a cutoff value.
Output: P, a set of clusters.

1. Construct I_S.
2. $T \leftarrow \{r\}$, the root of H.
3. $E \leftarrow \emptyset$.
4. **repeat**
   (a) $w \leftarrow f(S, T)$.
   (b) **If** $|I_S(w)| \leq c$, $E \leftarrow E \cup I_S(w)$, **else** $T \leftarrow T - \{w\} + C(w)$.

   until $|T| > m$.
5. $T \leftarrow T \cup E$.
6. Cluster T using a linear time clustering algorithm. We obtain P, a partition of T into disjoint sets.
7. Replace each node n in P by $I_S(n)$.
8. **return** P.

Table 2: The clustering algorithm for an arbitrary data set with cutoff values added. If a node includes few documents in S, we add those documents to a separate set E instead of spending time expanding the node.

general, we can replace nodes containing less than c documents by c single-document nodes. Since we only look at a constant number of nodes, the number of new nodes created this way is also a constant.

Since we don't want our choice of c to affect how many nodes we expand, we count the single-document nodes separately from normal nodes. Instead of keeping them in T, we move them into another set E. We still stop only when T reaches a given size. Since $|E|$ is bounded by a constant, this does not affect the analysis of the running time.

With this optimization, the algorithm is as shown in Table 2.

2.3 The RATIO test

All that remains is to define the goodness test used to select nodes for expansion. The most obvious test is precision: a node is good if most of the documents it contains are in S. This gives the following *ratio test* definition of goodness:

$$G_r(n) = |I_S(n)|/|D(n)|.$$

$f(S,T)$ returns that node in T with the lowest goodness. With the ratio test, it will favor nodes with small intersection, which probably have children with *no* intersection; hence this goodness test will result in extensive pruning, improving the results. On the other hand, large nodes with fairly good ratios will stay intact in T, even though they include many documents not in S.

2.4 The ROOT RATIO test

If one large node has a large intersection with S, the RATIO test will tend to preserve it. This can be a problem for clustering, since the clustering algorithm will treat all the documents in the node as a single entity, leading to potentially lopsided cluster sizes. We can encourage the expansion of such large nodes by re-expressing the RATIO goodness definition as follows:

$$G_{rr}(n) = \sqrt{|I_S(t)|}/|D(n)|.$$

In this case, having a lot of documents in S is not a guarantee of a good ratio; it is more advantageous to have a smaller value of $|D(n)|$. This helps ensure that the output nodes will all have approximately an equal number of documents from S.

2.5 The INFORMATION test

Another approach for testing goodness is to use an information theoretic measure. A node is a good candidate for replacement by its children if its children encode more information about S than the node itself. This implies that the matches in the parent are unevenly distributed among the children, allowing us to prune the bad children and keep the good ones.

Let n_i be the children of n. We define the information in a node to be

$$I(n) = -\frac{|I_S(n)|}{|D(n)|}\log_2\left(\frac{|I_S(n)|}{|D(n)|}\right).$$

Then the appropriate goodness measure is:

$$G_I(n) = I(n) - \sum_i \frac{|D(n_i|}{|D(n)|} I(n_i).$$

$f(S,T)$ will return that node in T with the highest information gain. This has the advantage of picking nodes that will benefit the most from being replaced by their children. On the down side, it will ignore large nodes with few matches if these matches are distributed evenly among the children.

3 Results

To evaluate the fast clustering methods with respect to the baseline performance of direct clustering, we used the methodology proposed in [6]. That is, we ran one Scatter/Gather step and measured the precision, at three cut-off values, of the best cluster (the one with the highest density of relevant documents). We used the TREC4 *ad hoc* evaluation corpus, consisting of 567,522 documents and 49 queries with relevance assessments [5]. The corpus was preprocessed by the Xerox PARC TextDatabase engine [4], including the construction of a cluster hierarchy with fan-out 5 and 201,772 internal nodes. We used a linear-time partitioning algorithm called *fractionation* as the baseline direct clustering method [3]. For the fast clustering methods we expanded until we had collected 200 nodes ($m = 200$), and then used fractionation to cluster back to five groups.

For each query we collected the 1000 TREC4 documents ranked most similar by the TextDatabase similarity search algorithm.[3] We then clustered these 1000 documents in four ways: with fractionation, fast clustering with the RATIO test (G_r), fast clustering with the ROOT RATIO test (G_{rr}), and fast clustering with the INFORMATION test (G_I). In each case, we clustered the documents into five clusters. In each cluster, we preserved the document order of the original ranking; hence documents within each cluster are ranked by similarity to the query.

Each TREC4 query is associated with a list of judged relevant documents. For each clustering scheme, we picked the cluster with the largest ratio of relevant documents to total documents. We then computed precision at three document cutoffs within this "best" cluster: at five, ten and twenty documents. This gave us three precision numbers for each technique. We used precision at document cutoffs rather than uninterpolated precision or average precision at 11 recall points since Scatter/Gather is intended as a tool for gathering positive exemplars and hence should be judged more on precision that recall.

Five queries were removed because the similarity search at cut-off 1000 found no relevant documents, reducing the number of queries considered from 49 to 44.

We judged one technique to be *better* than another for a particular query if at least two of the numbers were higher. Occassionally all three pairs of numbers were equal, or one pair would be equal while the other two pairs split. In that case we said the techniques were *equal* on that query.

An example run for a TREC4 query is shown in Figure 2.

We performed a statistical sign test on these results, comparing the number of *better* rankings to the expected number of *better* rankings assuming they were generated via a fair coin toss (i.e. from a binomial distribution with $p = .5$ and $n = |better| + |worse|$). This comparison can be expressed as a *p-value*, which is the likelihood of seeing an observation more extreme under the null hypothesis that $p = .5$.

[3] We use a simple cosine ranking scheme with tf.idf weights. In particular the similarity of document d to query q is computed as

$$S(d,q) = \frac{\sum d(w)q(w)}{\sqrt{\sum d(w)^2}}$$

where $d(w) = \sqrt{f_d(w)}$, $q(w) = \sqrt{f_q(w)}\ \log(N/n(w))$, $f_x(w)$ is the frequency of w in x and $n(w)$ is the number of documents in which w occurs.

```
Topic: 202

Text: Status of nuclear proliferation treaties --
      violations and monitoring.

Number relevant: 283

Cluster distribution:
 cluster  #-rel  size   density
 0:         141   700   0.20142858
 1:           0    56   0.0
 2:           0   104   0.0
 3:           0    69   0.0
 4:           0    71   0.0

Fast cluster distribution (ratio):
 cluster  #-rel  size   density
 0:         141   750   0.188
 1:           0    22   0.0
 2:           0   165   0.0
 3:           0    30   0.0
 4:           0    33   0.0

Fast cluster distribution (root-ratio):
 cluster  #-rel  size   density
 0:          97   514   0.18871595
 1:          25   110   0.22727273
 2:          19   144   0.13194445
 3:           0   170   0.0
 4:           0    62   0.0

Fast cluster distribution (info):
 cluster  #-rel  size   density
 0:         138   708   0.19491525
 1:           3    87   0.03448276
 2:           0    43   0.0
 3:           0   111   0.0
 4:           0    51   0.0

Precision at set cutoffs:
        cluster ratio w-ratio info
 5 docs:   0.8    0.8    0.6   0.8
10 docs:   0.7    0.7    0.5   0.7
20 docs:   0.5   0.55    0.5  0.55
```

Figure 2: Output from a comparison run for Topic 202 from the TREC4 collection. Each clustering technique groups 1000 documents into five clusters; the one with the largest density (not always the one with the most relevant documents) was examined. In this case INFORMATION and RATIO are equal and better than direct clustering, which is in turn better than ROOT RATIO.

	cluster — rat	cluster — w-rat	cluster — info	rat — w-rat	rat — info	w-rat — info
better	27	23	22	19	14	18
equal	6	4	10	9	13	7
worse	11	17	12	16	17	19
p-value	**.007**	.215	.061	.368	.360	.500

Table 3: Pairwise comparison of various clustering techniques. Straight clustering wins a majority of its head-to-head competitions, but the fast clustering techniques are all competitive. Using a significance cut-off of $p = .05$, we can say straight clustering is statistically superior to RATIO. We cannot distinguish, statistically, between any other pairs of tests.

We show the results of each pairwise comparison in Table 3.

Using a significance cut-off of $p = .05$, the bold entry in Table 3 is statistically significant. That is, we can say that RATIO is statistically worse than direct clustering but can make no other distinctions between the tests. Thus, for instance, we do not reject the hypothesis that ROOT RATIO performs as well as direct clustering. This indicates that, for at least this purpose (Scatter/Gather), the fast clustering methods perform as well as the slower direct method.

4 Time

We must justify our claim that the fast clustering techniques are actually faster than direct clustering of S. This is a particular concern when n is large and $|S|$ is small, since the $\log n$ factor in $|S| \log n$ may be comparable to the implied constant in the $O(|S|)$ fractionation clustering technique. In Figure 3 we demonstrate that, at least for $|S| = 1000$, the fast clustering techniques are indeed significantly faster. In addition, they show less variation between queries, indicating they are less susceptible than direct clustering to the vagaries of the data set contents. Testing shows that fast clustering is at least as fast as regular clustering, for most queries, even if $|S|$ is as small as 100. These tests used the ROOT RATIO goodness test for fast clustering; presumably RATIO would be slightly faster and INFORMATION slightly slower. We do not expect these differences to be significant, however.

Figures 5 and 4 show that the running time of these algorithms agrees very well with the predictions, and while the running time varies from query to query, general trends are clear. direct clustering takes time linear in $|S|$. Fast clustering, on the other hand, is insensitive to $|S|$ for $|S| \geq 500$. Though we only show $|S|$ up to 1000 on this graph, experiments with $|S|$ as large as 15000 confirm that the running time increases only very slowly with $|S|$ — there is usually less than one second difference in the running times for $|S| = 500$ and $|S| = 15000$. This is consistent with our claim of "almost-constant" running time: the time contribution of the $O(|S| \log n)$ step is dwarfed, in practice, by the contribution of the $O(1)$ clustering step.

For $|S| < 500$, the fast clustering algorithm reduces to the direct clustering algorithm. Recall that the fast clustering algorithm gains its speed advantage by clustering only m hierarchy nodes (for some constant m) instead of all $|S|$ documents. When $|S| < m$, however, the fast clustering algorithm behaves exactly like the direct clustering algorithm. Thus, we see the fast clustering algorithm running exactly as fast as the regular clustering algorithm for small S.

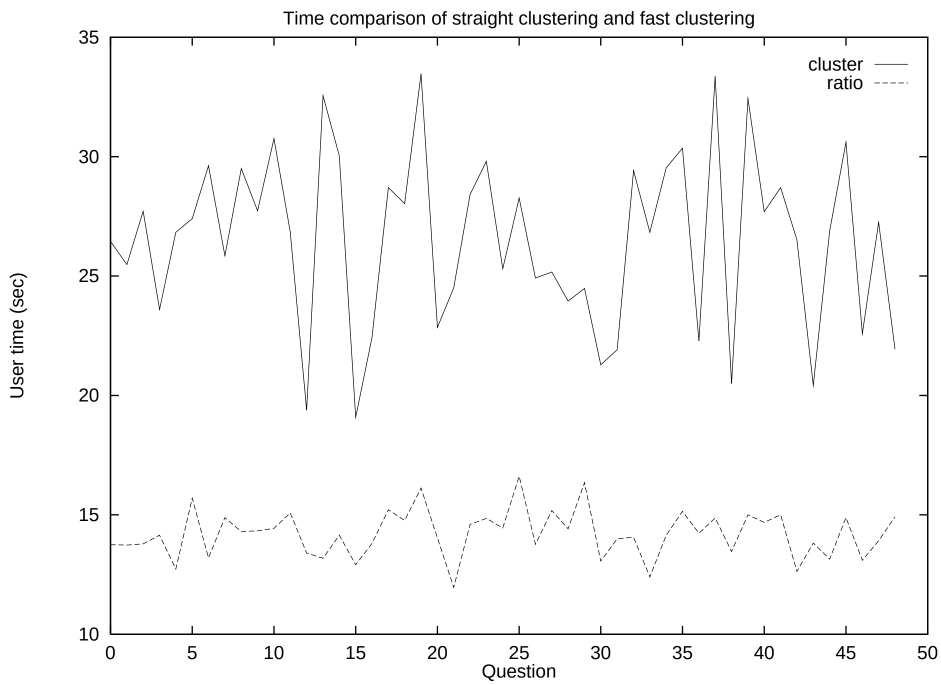

Figure 3: ROOT RATIO is taken to be representative of the fast clustering techniques, while "cluster" labels the time taken for direct clustering. ROOT RATIO is significantly faster.

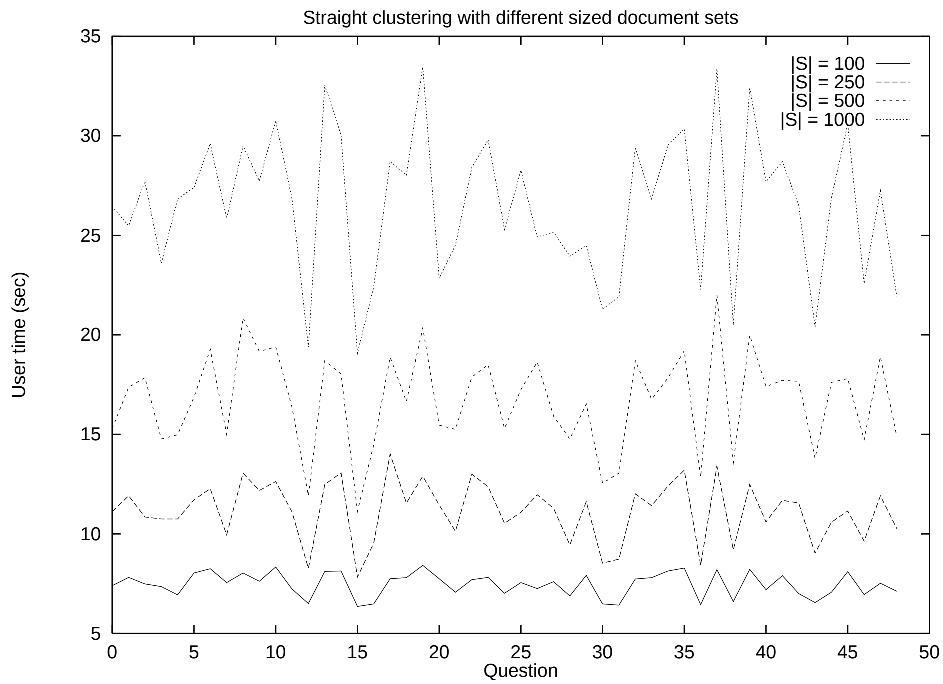

Figure 4: direct clustering on various-sized document sets. Though the time varies from query to query, it is clear that time spent increases approximately linearly with $|S|$.

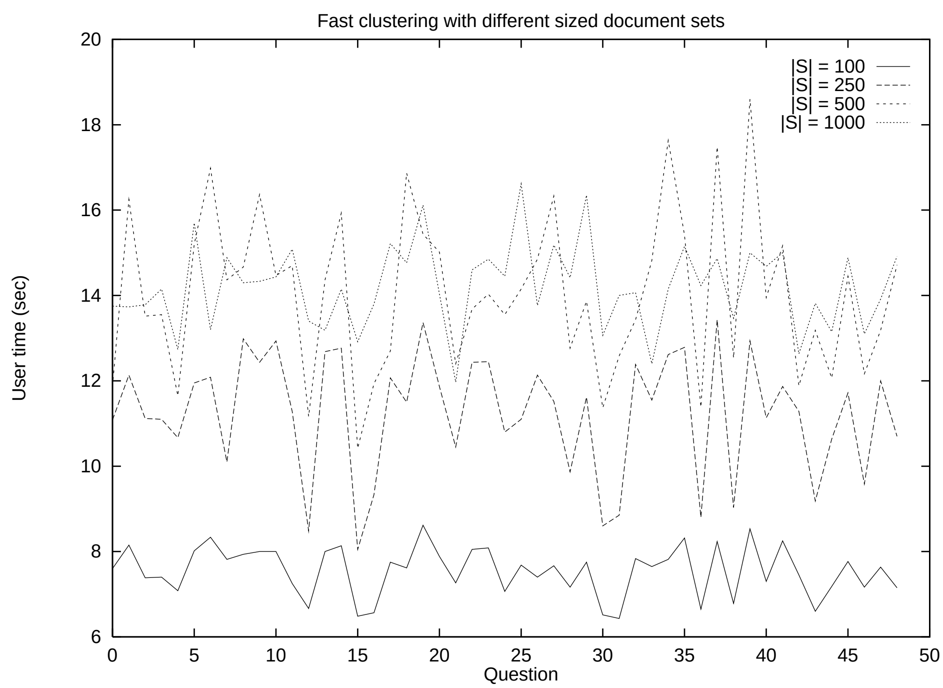

Figure 5: Fast clustering on various-sized document sets. Though the time varies from queries to query, it is almost constant for $|S| \geq 500$. Smaller S are faster because they are smaller than the constant ceiling imposed by the fast clustering algorithm. Therefore, they are exactly as fast as the direct clustering algorithm.

5 Conclusion

Almost-constant-time clustering is a fast and reasonable alternative to direct clustering of arbitrary document sets. While these "fast clustering" techniques did not always perform as well as direct clustering on the TREC4 queries, the performance loss was not statistically significant for the best fast clustering techniques, and the time savings were often quite large. As the size of the document set increases, fast clustering becomes increasingly appealing.

Of the various fast clustering techniques we examined — RATIO, ROOT RATIO, and INFORMATION — ROOT RATIO and INFORMATION performed the best in head-to-head tests, while statistical tests indicated RATIO is inferior. Since ROOT RATIO is simpler to calculate than INFORMATION, we recommend using it as an alternative to direct clustering when clustering arbitrary document sets.

References

[1] Manette B. Lazear Adrienne J. Kleiboemer and Jan O. Pedersen. Tailoring a retieval system for naive users. In *Fifth Annual Symposium on Document Analysis and Information Retrieval*, April 1996.

[2] 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 *Proc. 15th Annual Int'l ACM SIGIR Conference on R&D in IR*, June 1992. Also available as Xerox PARC technical report SSL-92-02.

[3] Douglass R. Cutting, David R. Karger, and Jan O. Pedersen. Constant interaction-time Scatter/Gather browsing of very large document collections. In *Proceedings of the 16th Annual International ACM/SIGIR Conference*, pages 126–135, Pittsburgh, PA, 1993.

[4] Douglass R. Cutting, Jan O. Pedersen, and Per-Kristian Halvorsen. An object-oriented architecture for text retrieval. In *Conference Proceedings of RIAO'91, Intelligent Text and Image Handling, Barcelona, Spain*, pages 285–298, April 1991. Also available as Xerox PARC technical report SSL-90-83.

[5] Donna Harman, editor. *The Fourth Text REtrieval Conference*. National Institute of Standards and Technology Special Publication 500-236, 1996.

[6] Marti A. Hearst and Jan O. Pedersen. Reexamining the cluster hypothesis: Scatter/gather on retrieval results. In *Proc. 19th Annual Int'l ACM SIGIR Conference on R&D in IR*, August 1996.

[7] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Pretice Hall, Engelwood Cliffs, N.J. 07632, 1988.