Using Lucene to Index and Search the Digitized 1940 US Census

Liana Diesendruck
ldiesend@illinois.edu

Rob Kooper
kooper@illinois.edu

Luigi Marini
lmarini@illinois.edu

Kenton McHenry
mchenry@illinois.edu

National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign

ABSTRACT

An improved approach towards enabling search capabilities over large digitized document archives is described, in which Lucene indices were incorporated in a framework developed to provide automatic searchable access to the 1940 US Census, a collection composed of digitized handwritten forms. As an alternative to trying to recognize the handwritten text in the images, Word Spotting feature vectors are used to describe each cell's content. Instead of querying the system using regular ASCII text, any query is rendered as an image and a ranked list of matching results is presented to the user. Among other pre-processing steps required by the framework, an index must be compiled to provide fast access to the feature vectors. The advantages and drawbacks of using Lucene to index these vectors instead of other indexing methods are discussed in light of the challenges confronted when dealing with digitized document collections of considerable size.

Categories and Subject Descriptors

H.3 [**Information Systems**]: Information Storage and Retrieval; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation, Search process*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

General Terms

Experimentation, Performance, Design

Keywords

Content Based Retrieval, Searchable Access, Approximate Similarity Search, Lucene

XSEDE '13, July 22 - 25 2013, San Diego, CA, USA

1. INTRODUCTION

As the amount of scanned documents openly available to the public grows, providing searchable access to their contents becomes of vital importance. Access to the information contained in typed documents can be provided by extracting the text from the scanned images through OCR. Unfortunately, no similar, simple approach is available for handwritten content.

Word Spotting feature vectors were previously suggested as a good descriptor to the handwritten content of images [8, 9, 10]. In a previous work [4] we described a framework for using Word Spotting to provide automatic searchable access to the 1940 Census. The framework provides a means of generating Word Spotting feature vectors to describe the contents of hand-filled forms, indexing these vectors to enable later search, and leveraging the users' activity in the system as a passive crowdsourcing element to improve the search results.

To query the database of feature vectors, a similar vector must be computed for the query. Namely, the query must be provided in the form of an image that has its feature vector extracted and compared to the ones stored in the system. Since users mostly query the system by typing text, the input is rendered in 'Rage Italic', a font that resembles handwriting, generating the query image.

Of central importance when providing access to a collection through Word Spotting is the system index being used. The previously reported index was composed of a collection of binary trees where the nodes of the trees represented clusters of elements. While it achieved good results for the provided annotated testbed and real-time evaluations, the method used to build this index proved to be overly time consuming when dealing with large datasets [4]. We discuss here how to index the system with Apache Lucene, a Java library that enables textual search, adapting the approaches provided in [1] and [6].

2. INFORMATION EXTRACTION

In order to extract the Word Spotting feature vectors from the Census images, we must first segment the forms into cells, which are the smallest units of information available. We start by correcting any form rotation resulted from the scanning process. Next, a form template is matched to the lines detected in the form image and the image is segmented into cells based on the template's grid.

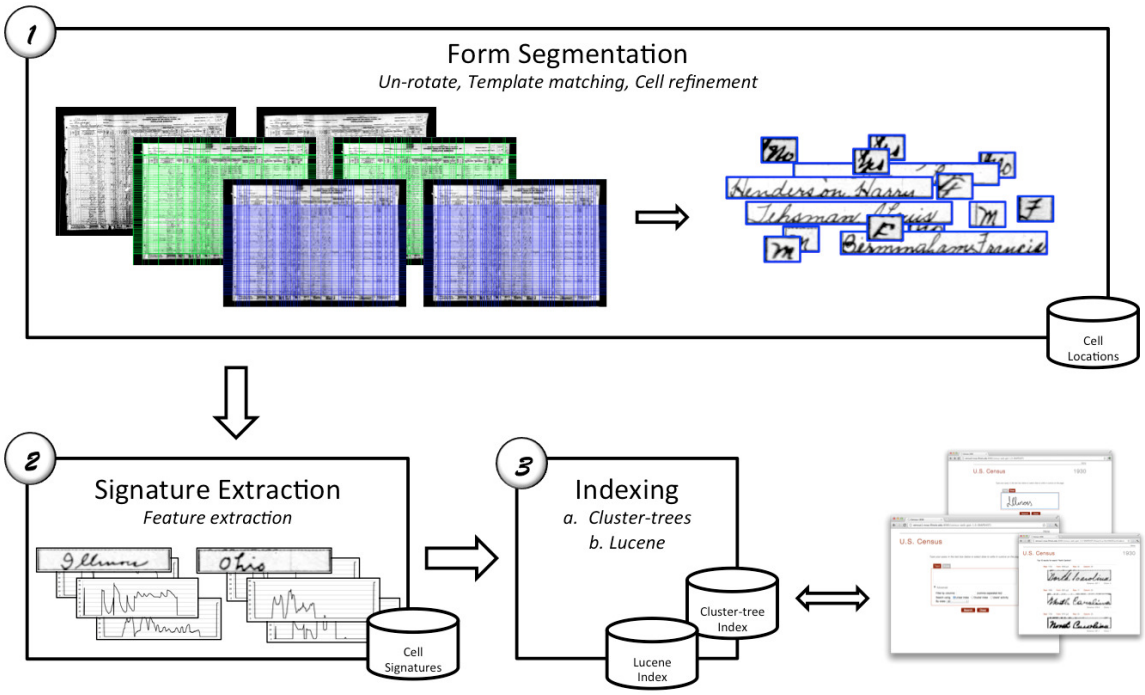

Figure 1: *The framework presented in this work. Step 1: the images are uploaded into the system, they are processed to rectify possible rotations, reduce smudges and bleeds, etc., and then are segmented into cells. The locations of the cells are kept in a database in order to be used in the next pre-processing step. Step 2: feature vectors are extracted from all cell images and stored in a database. These are the vectors that describe each cell image in the system. Finally, in step 3, the feature vectors are indexed in two possible ways: a. using cluster-trees, and b. using Lucene. The indices are used to speed up the query turnaround time. These indices can then be used to query the system.*

Noise reduction techniques are applied to the segmented cells in order to sharpen their contents. Finally, the Word Spotting feature vectors can be extracted from the cell images. Here, we used the same features as in [10], specifically the upper, lower, and total column ink profiles. A detailed description of the feature extraction process, including the segmentation of the forms, can be found in [4].

3. DATASET INDEXING

Once feature vectors were extracted from all cell images in the system, the best matches to any query could be computed by comparing its feature vector to all the vectors stored in the system. While this method would provide perfect results in the sense that no element would be skipped during the process and thus no good match could be possibly missed, it is not practical when dealing with a 7-billion-element dataset. Even if considering that each comparison takes a very conservative 1 millisecond, the time required to perform a single query would be 2.7 months in a single-core machine; even using a massive parallel machine to do these searches would still take more than a few seconds which is what a web user is used to. Therefore, it is necessary to use some kind of index to reach smaller sub-sets of the feature vectors during the query process. Two different indexing approaches are discussed in the following subsections. First, we give a brief overview of using cluster-trees to index the system as reported in [4]. Then, we show how we adapt and expand the approach in [6] to be used with our feature vectors.

3.1 Indexing with cluster-trees

Cluster-trees can be used to quickly query a collection of images represented by Word Spotting feature vectors [4]. A cluster-tree is a binary-tree where each node represents a cluster of vectors and contains its cluster's size and mean vector. A node's cluster is partitioned into two smaller ones represented by its right and left children. The tree's root represents a cluster of all elements in the tree and its leaves are singletons.

If a cluster-tree is used to index a dataset of feature vectors, the query procedure consists of descending the tree until a small enough cluster is reached. Then, all elements in the retrieved cluster are compared to the query feature vector and an ordered list of the best matches is produced. When descending the tree, the sub-tree chosen is the one with the average vector closest to the query's. This approach was shown to yield good results and fast query responses [4].

The cluster-trees used as indices here are built based on the dendrogram received by hierarchically clustering all the dataset's elements in a bottom-up manner. Specifically, in the case of the Census dataset, the complete-linkage [2] approach seemed to generate the cluster that best suited our needs. The complete-linkage hierarchical clustering of a dataset can be accomplished in different ways depending on the resources available [2, 3]. Nevertheless, the time complexity of even the fastest of the methods and the amount of memory it requires hamper the use of cluster-trees as indices of large datasets. This is the case even when dividing the data into subsets and having multiple cluster-trees com-

bined as an index (at the expense of a somehow increased query response time). A last point to consider is that since the cluster-trees are based on the clustering of the elements in the dataset, it is not trivial to add new elements to this kind of index without performing the clustering process once again.

3.2 Indexing with Lucene

Inverted indices are a popular way of indexing in systems that support textual search. In an inverted index, each index entry represents a significant term in the system and contains a list of documents where it can be found. Optionally, the index can be improved by also storing relevant term statistics and meta-data. Apache Lucene [5] is a Java library that enables textual search. It facilitates the process by building an inverted index for the documents in the system and managing memory and disk requirements during the querying and indexing processes. The process of using Lucene to perform textual search is divided in two main steps: indexing and searching. During the indexing, there is the need to define how the analyzed documents are going to be parsed, which information is going to be stored for later use, what kind of fields to use (e.g. numerical or textual fields), whether or not to store statistics with the terms or to keep the relative position of terms in a document, how many index segments to keep, etc. Later, during search time, the query is analyzed - i.e. it is parsed and the relevant terms are extracted - and the inverted index is searched based on a similarity function that can be defined by the user or pre-defined by the system.

Nevertheless, in the case of the 1940 Census, the system is not dealing with textual information, but with images. As previously described, in this case every cell image is represented by a feature vector in a 30-dimensional vector space. While standard for textual search, the classical inverted index approach has serious drawbacks when dealing with elements represented numerically as points in non-discrete space. The main problem is that most vectors are unique representations of an image and there is no repetition of elements that would take advantage of the inverted index structure. Even if we consider each dimension of the feature vector as a possible term to be indexed, the fact is that two vectors can contain the same value in many of their dimensions and still be much farther away in Euclidean distance than two vectors that have not even a single dimension matched.

Amato et al. [1] postulated that vectors that are close to each other in space "see" their nearby neighborhood in a similar way. Namely, in most cases, if vector $v1$ is very close to vectors $v2$ and $v3$, these two vectors are also close to each other. This concept was leveraged by Amato so to enable the indexing of feature vectors using inverted indices. Instead of describing each vector by its dimensions, each vector is described by its proximity to reference objects, i.e. vectors scattered in the vector space that are used as reference points. Each vector is represented by an ordered list of reference objects, from the closest one to the farthest away. The main idea is, when querying the system, to compute the query's reference object list and look for elements that have their reference list ordered similarly to the query's. In order to speed-up the search process, each element might be represented by a partial reference list containing only the closest n reference objects to the element. The search space can be further constrained by searching for the object on position j from one list only through positions $(j-k, j+k)$ of the second reference list. It is important to notice, however, that this approach performs an approximate search and may generate some incorrect results since the vectors are compared based on their reference list and not on direct Euclidean distance.

In order to adapt the approach in [1] so to be used with Lucene, Gennaro and colleagues [6] suggested to alter the reference listings so that every term on a reference list is repeated enough times to reflect its position. Namely, if a list contains 10 elements, the first element would be repeated 10 times, the second would appear 9 times, and so on. During the search process, an adapted reference list is also calculated for the query and Lucene produces an ordered list of the best matches found. It is important to notice that, by using this approach, we cannot reduce the search space by constraining the distance between positions as in [1].

We suggest instead to embed the position of a reference object on a listing as part of its descriptive term. For example, if reference object ro_i position on a reference listing is j, then it will be represented in the adapted reference list as ro_i_j. If m reference objects are used to index a dataset, there are potentially m^2 terms that can be used in the adapted reference lists. This results in faster query responses since (a) the elements are spread over more inverted index entries, and (b) it is possible to constrain the search space based on the positions of the elements in the query's reference list.

Choosing good reference objects is obviously of fundamental importance. In [1], it was estimated that a dataset containing n elements could be correctly indexed using $2 \cdot \sqrt{n}$ reference objects. The problems arise when the feature vectors are not normally distributed in the vector space. If the entire collection of feature vectors is known previously to the picking of the reference objects, the reference objects can be restricted to the vector sub-space that contains all the feature vectors. Although this constraint improves considerably the representativeness of the reference objects, it is not enough to guarantee good results. Since, by design, there are considerably fewer reference objects than feature vectors, if the reference objects were to be distributed in a regular grid spanning the vector sub-space, numerous feature vectors would likely be concentrated in certain "pockets" of the grid. If the concentration of feature vectors in these pockets were not very high, this could be an advantage. On the other hand, a high concentration of feature vectors in a single grid pocket would impair the effectiveness of the system. A simple and effective, but not deterministic, solution is to randomly choose the reference objects from the collection of feature vectors.

Once the dataset is indexed, it can be queried in multiple ways. One possible approach is to use Lucene's TermDocs, which retrieves all documents containing a certain term. If, for example, we are looking for reference object ro_i in position j, we will use TermDocs to find all documents containing the terms ro_i_y, where $y \in [j-k, j+k]$ and k is defined by the user. Then scores are assigned to the matches, such that the closer j and k are, the higher the score is. Using this strategy, the number of documents that are analyzed is reduced considerably, yielding faster turnarounds. A similar search procedure can leverage Lucene's capabilities even more by letting it score the matches automatically. This

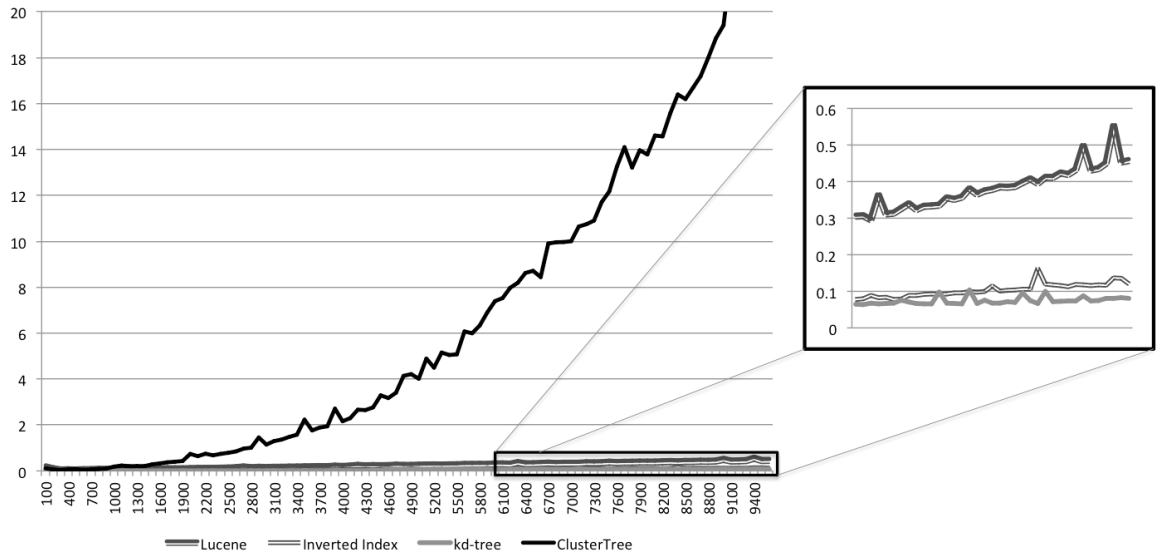

Figure 2: *Index building times (in seconds) as function of the number of elements indexed. A comparison between a Lucene index, a cluster-tree index, an inverted index, and a kd-tree index. The cluster-tree time complexity is clearly much slower than the other indexing methods. Although the Lucene indexing is slower than the kd-tree and the inverted index their time complexities are of the same order of magnitude*

has the potential to improve considerably query times since Lucene applies some search optimizations based on the scoring of matching documents. The idea is to let Lucene do the scoring based on a 'boosting factor' provided by us which reflects the distance between the ideal position and the actual position the reference object appears in the reference list of a document. The search is constrained in the sense that we still look only for terms ro_i_y, where $y \in [j-k, j+k]$, but each one of these will be part of a different TermQuery which will be appropriately boosted with a value proportional to $|j-k|$. A comparison of query response times and accuracy can be found in the experimental session.

4. EXPERIMENTAL RESULTS

We use an annotated dataset of 10,000 cell images to evaluate different indices' performances. The dataset is composed of cells in four different Census categories (marital status, profession, place of origin, and one of the several yes/no questions) associated with annotations that were manually generated. In addition to the cluster-tree and the Lucene indices, we also show comparative results for a kd-tree based index built from a slightly modified version of [7] and our own implementation of Amato's inverted index.

Throughout this section all mentions to Lucene refer to Lucene 3.6.1.

4.1 Index building

Four different indices were built using the annotated dataset - a cluster-tree index, a Lucene index, a kd-tree index and an inverted index - while indexing times were recorded. The results can be seen in Figure 2, which shows how the time required to build an index grows as a function of the number of elements being indexed. The Lucene index was built using 50 reference objects, while the reference list used to describe each feature vector consisted of the 20 reference objects closest to it.

As expected, the cluster-tree computation is considerably more demanding than the other approaches; a discussion of possible indexing approaches for cluster-trees can be found in [3]. The fastest index building time is the achieved by the kd-tree, although there seems to be no significant differences in time complexity for all methods other than the cluster-tree.

4.2 Index querying

The same indices of the previous subsection were used in order to analyze their querying speeds and accuracies. The Lucene index was queried twice, once using our own scoring system and once using Lucene's query boosting. When comparing the reference lists between the query and any element, we constrained the search to include only 5 positions, meaning that we compared any reference string in position i in the query's list only to the strings in positions $[i-5, i+5]$, as previously described. In addition, the comparison was restricted to the first 10 of the 20 reference objects in the query's reference list.

Here, the cluster-tree achieves the best results due to its binary-tree structure. As expected, the kd-tree is not as fast as the cluster-tree due to the high-dimensionality of the vectors. The leveraging of Lucene's term query boosting provides a significant speed-up in query time, making the boosted Lucene query the best performer other than the cluster-tree query.

An additional Lucene index holding all elements of the dataset was built with the representation used in [6]. Both Lucene indices were queried multiple times. Queries performed on the index implemented using our suggested vector representation, which enables constrained-position queries by embedding the position of the reference objects in their representative terms, performed on average 34% faster than the ones using the original representation and index. The speed-up is most likely due to the constraining of the search

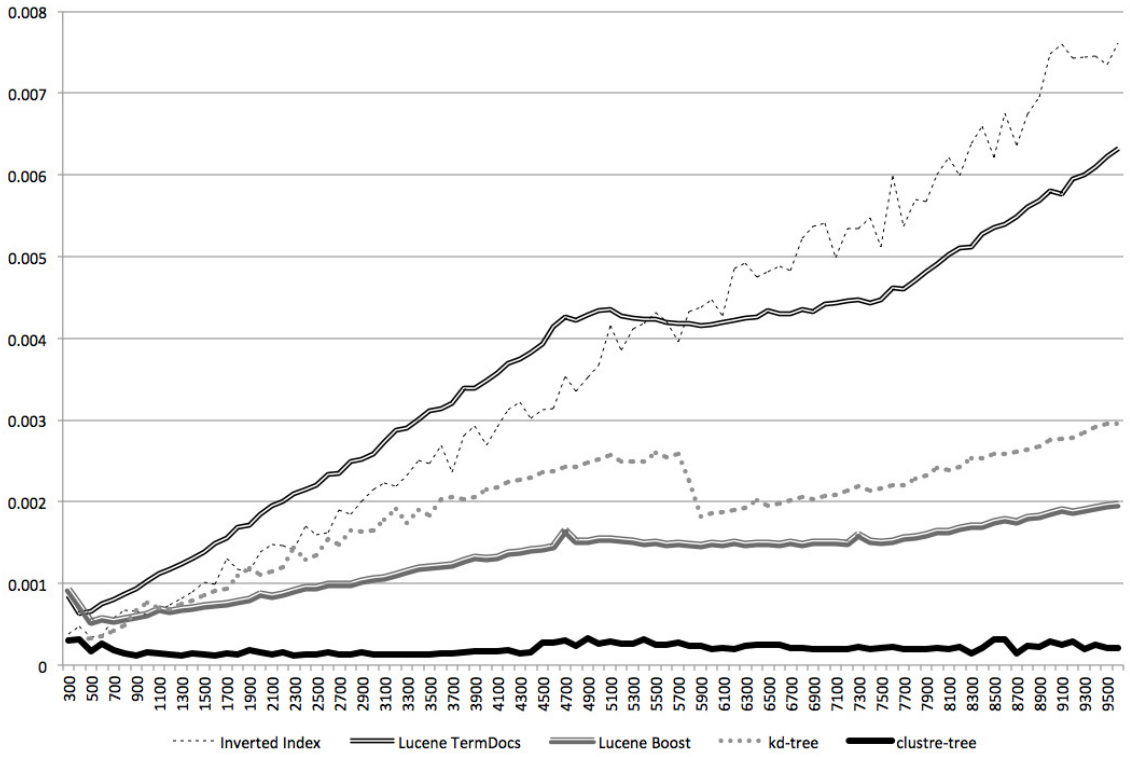

Figure 3: *Average query time (in seconds) as a function of number of elements indexed. The faster query response is achieved by the cluster-tree search followed by the Lucene boosted search.*

space enabled by this representation. Despite the reduction of the search space, it is interesting to notice that both approaches performed almost identically when considering the quality of the retrieved results.

4.3 Reference Objects Selection

Three different methods of reference objects election were used to select 50 reference objects to be used in different Lucene indices. The first method randomly chose vectors from the 30-dimensional vector sub-space that contains all the feature vectors in the system. The second approach randomly drafted 50 feature vectors and utilized them as the reference objects of the index in an attempt to have the distribution of reference objects somewhat similar to the distribution of the original feature vectors. The third approach uniformly distributed reference objects in the distance between the vectors composed by the lowest and the highest possible values in each dimension. This approach is clearly not sensible and a grid would be a better way of uniformly distributing the reference objects, unfortunately this grid is impossible to create when trying to generate only 50 reference objects in a 30-dimensional vector sub-space. Nevertheless, we use this approach exactly to illustrate the importance of choosing good reference objects.

We run each of the methods 20 times using the resulting reference objects to create Lucene indices. Then the indices were queried using each feature vector in the system as queries. A correct match to a query was one with the same transcription as the query's. Once a feature vector was being used as the query, it was not considered as a valid response to it. Although the accuracy pattern is similar for all querying methods, the results presented here were generated by using Lucene's boosting to score the matches to the queries. The resulting average accuracies for each method can be seen in Figure 4. Not surprisingly, the third approach yielded extremely poor results. The best results were obtained when using a subset of the system's feature vectors as reference objects. Although the advantage of this method seems to be very small in comparison to randomly choosing the reference objects, t-tests show this advantage to be statistically significant ($p < 0.0005$ for all 1 to 10 top matches comparisons).

4.4 HPC Benchmarking

In order to evaluate the amount of time needed to index the entire Census dataset, and estimate what would be the Lucene query response time in the real system, we indexed the entire dataset for the state of North Carolina. While our previous work used NCSA's Ember at the University of Illinois (since then Ember was retired) and the Steele cluster at Purdue University, here the indexing was executed on Pittsburgh Supercomputing Center's Blacklight. Blacklight is a shared-memory system with 256 blades, which contain two eight-core processors each. Thus, on Blacklight each core has 8 GB of memory available, in contrast to the 5 GB available per core on Ember and the 2-4 GB on Steele. The increased memory available on Blacklight not only reduced the required amount of writing to disk during the Lucene index building, but also enabled us to use RAMDisk to alleviate the I/O overhead during the process caused by multiple

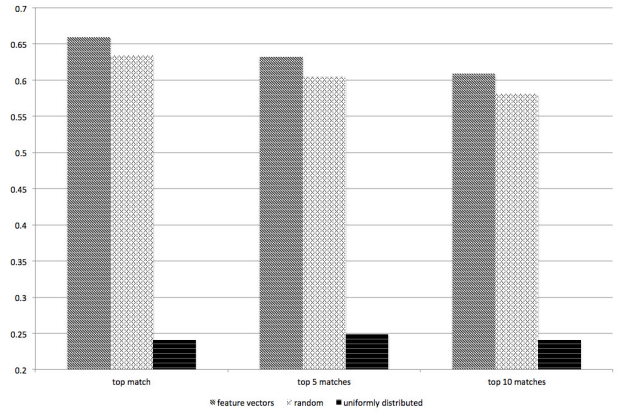

Figure 4: *Average query accuracy vs. reference object choosing method. The label 'feature vectors' designates the method of using the system's feature vectors as reference objects; average query accuracies are* 66.0% ± 0.4%, 63.2%±0.4% *and* 61.0%±0.5%, *for 1, 5 and 10 top matches, respectively. The label 'random' describes the selection of random reference objects in the vector sub-space encompassing all the system's feature vectors; average query accuracies are* 63.5% ± 0.2%, 60.5% ± 0.1% *and* 58.2% ± 0.1%, *for 1, 5 and 10 top matches, respectively. Finally, the label 'uniformly distributed' represents the reference objects distributed between the vectors composed by the lowest and the highest possible values in each dimension (average query accuracies are* 24.1%, 25.0% *and* 24.1%, *for 1, 5 and 10 top matches, respectively)*

instances of Lucene writing to and reading from many files while adding entries and updating their indices.

For North Carolina there is a total of 124 reels, each reel holding on average 1,000 forms. Each reel was processed by a single process using 4 GB for computation and 4 GB for RAMDisk. The processes divided the reels into 44 indices (one per form column). On average, each of these indices contained 32,628 feature vectors, which were fetched from the MongoDB database. The average building time of the Lucene indices was 71.78 seconds per index. When the indices were ready, an additional 0.17 seconds were needed on average to copy each index from scratch RAMDisk to scratch space. After processing all the reels we had 5,456 separate indices.

All small indices pertaining to the same column were later merged together resulting in a total of 44 final indices. The merging process was performed in our local computers in a total of 14.7 seconds per index. Later, the resulting merged indices were optimized into one single Lucene segment in order to speed-up the query procedures. The optimization of the new merged indices into one single segment took on average 47.8 seconds per index.

All of the time averages previously reported were acquired by building indices with our own reference list representation, which embed the position of the object in its descriptive string. Initially, indices combining all three different reference list representations were built for each column of the Census forms. These indices were queried in order to stipulate which representation yielded the best query response times. Querying the merged indices took, on average, 0.85 seconds when using the reference list representation and Lucene boosted query method suggested here and 1.56 seconds when using the method suggested by [6]. Using the indices that were merged and optimized, the average query response time dropped to 0.43 and 0.93 seconds, respectively. The average query response time improved even more for the final indices containing only the chosen representation dropping bellow 0.1 seconds.

Since each reel column is indexed separately, we can directly estimate that, running on Blacklight with a similar configuration of 124 cores using 8 GB of memory each, it would take 1.38 days to index all 44 columns of the complete 4,646 Census reels. Additional 0.31 days would be required to merge all 204,424 (4,646 reels x 44 columns) indices in a single index for the entire Census. The optimization of this index could be completed in 1.1 days. To estimate the time required for merging and optimizing all the indices, we built, merged and optimized variable amounts of indices. The plots reporting the merging and optimizing times as functions of the number of reels can be seen in Figure 5.

5. CONCLUSIONS

We discussed how a Lucene index can be incorporated into our framework for providing searchable access to the handwritten information contained in the 1940 Census. We showed that the average query matching scores for the Lucene indices can be of the same order of the previously reported cluster-tree scores. Namely, we do not foresee any loss of query accuracy if the indexing method is chosen to be based on Lucene instead of the cluster-trees. In addition, it is possible that the results for the Lucene indices could be further improved by selecting better reference objects. Being able to estimate the distribution of the feature vectors and thus finding the sub-spaces with high concentrations of elements would enable us to select more relevant reference objects. Nevertheless, it is challenging to develop such distribution estimation without considerably increasing the time complexity of the indexing process.

Although the faster real-time query responses of the cluster-trees seem impossible to duplicate using the Lucene indices, the results of the North Carolina benchmarking show that the query response times using Lucene are good enough to produce a responsive, interactive system. In addition, Lucene indices have the advantage of being incremental, meaning that new elements can always be added to it in sharp contrast to the cluster-trees. Finally, it is clear that Lucene indices also present a considerably shorter building time in comparison to the cluster-trees. Our estimate of less than 3 days for building the complete index for the entire 1940 US Census dataset is a huge improvement from the estimated 48 days required for completing the cluster-tree indexing on Blacklight running on a similar configuration. Overall, using Lucene to index a dataset of feature vectors seems to be a good solution, not only when no or limited HPC resources are available, but also when the dataset is of a considerable size.

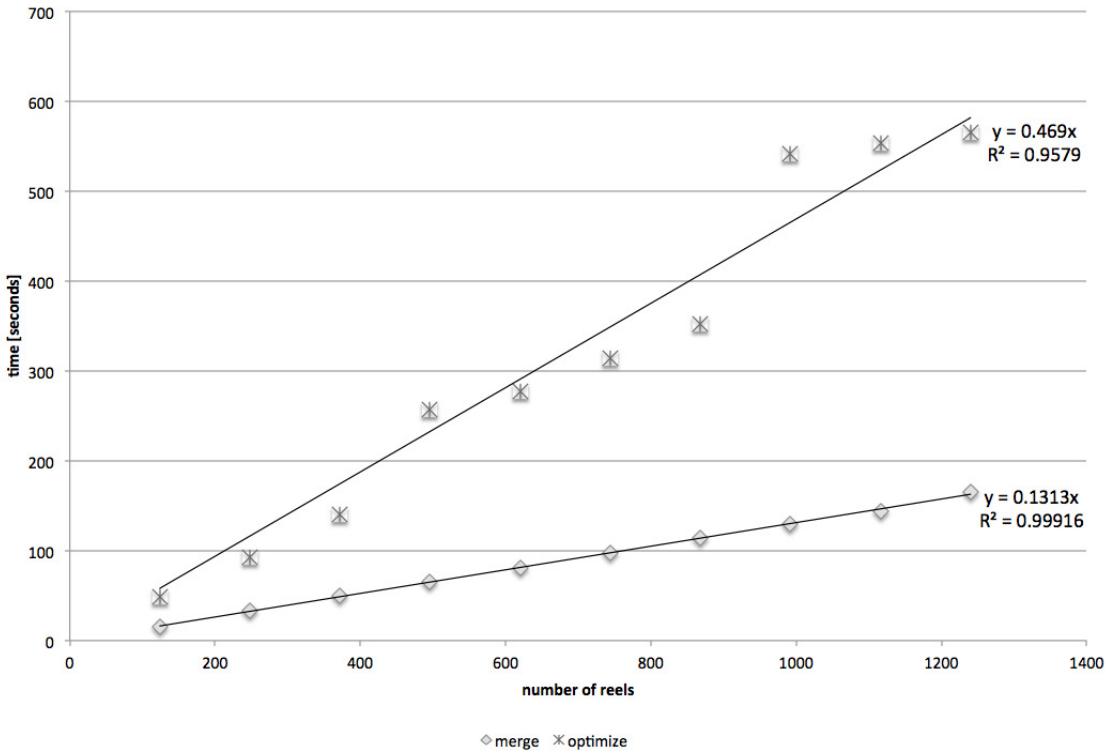

Figure 5: *Average merging and optimizing times (in seconds) as a function of the number of reels indexed. A linear fit is presented for both series.*

6. ACKNOWLEDGEMENTS

This research has been funded through the National Science Foundation Cooperative Agreement NSF OCI 05-25308 and Cooperative Support Agreement NSF OCI 05-04064 by the National Archives and Records Administration (NARA). This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

7. REFERENCES

[1] G. Amato and P. Savino. Approximate similarity search in metric spaces using inverted files. *International Conference on Scalable Information Systems*, 2008.

[2] W. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1984.

[3] L. Diesendruck, L. Marini, R. Kooper, M. Kejriwal, and K. McHenry. Digitization and search: A non-traditional use of hpc. *IEEE Conference on e-Science*, 2012.

[4] L. Diesendruck, L. Marini, R. Kooper, M. Kejriwal, and K. McHenry. A framework to access handwritten information within large digitized paper collections. *IEEE Conference on e-Science*, 2012.

[5] A. Foundation. Lucene apache. *lucene.apache.org*, 2011.

[6] C. Gennaro, G. Amato, P. Bolettieri, and P. Savino. An approach to content-based image retrieval based on the lucene search engine library. *Proceedings of the 14th European conference on Research and advanced technology for digital libraries*, 2010.

[7] S. D. Levy. Kd-tree implementation in java. *http://home.wlu.edu/ levys/software/kd*.

[8] R. Manmatha, C. Han, and E. Riseman. Word spotting: A new approach to indexing handwriting. *IEEE Conference on Computer Vision and Pattern Recognition*, 1996.

[9] T. Rath and R. Manmatha. Features for word spotting in historical manuscripts. *International Conference on Document Analysis and Recognition*, 2003.

[10] T. Rath and R. Manmatha. A search engine for historical manuscript images. *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2004.