Hybrid Indexing for Versioned Document Search with Cluster-based Retrieval

Xin Jin, Daniel Agun, Tao Yang, Qinghao Wu, Yifan Shen, Susen Zhao
Department of Computer Science
University of California at Santa Barbara, CA 93106, USA
{xin_jin, dagun, tyang, qinghao, yifanshen, susutou}@cs.ucsb.edu

ABSTRACT

The previous two-phase method for searching versioned documents seeks a cost tradeoff by using non-positional information to rank document versions first. The second phase then re-ranks top document versions using positional information with fragment-based index compression. This paper proposes an alternative approach that uses cluster-based retrieval to quickly narrow the search scope guided by version representatives at Phase 1 and develops a hybrid index structure with adaptive runtime data traversal to speed up Phase 2 search. The hybrid scheme exploits the advantages of forward index and inverted index based on the term characteristics to minimize the time in extracting positional and other feature information during runtime search. This paper compares several indexing and data traversal options with different time and space tradeoffs and describes evaluation results to demonstrate their effectiveness. The experiment results show that the proposed scheme can be up-to about 4x as fast as the previous work on solid state drives while retaining good relevance.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

Keywords

Versioned data, positional inverted index, query processing, search in document archives

1. INTRODUCTION

Organizations and companies archive many versions of digital data such as web pages, internal emails, source code, test data, and multimedia documents. Such data is critical for internal investigation, regulatory compliance, and electronic discovery [15]. It is not uncommon for many businesses to retain archived collections for 10 to 15 years and in some industries the retention periods may be as long as 100 years or more [16]. A long period of data retention implies many versions may have to be maintained. The size growth of an archival dataset also becomes rapid with advancements in cloud computing, content authoring, media sharing, and low-cost computer devices. The previous work on versioned data has studied the compression algorithms to identify shared data fragments among different versions [43]. Even though the index space for a versioned data collection can be compressed dramatically through fragmentation and deduplication, it is still time consuming to search the full index structure because a search procedure still has to deal with a large number of documents with many versions. A two-phase approach [22, 33] has been proposed to find top results first using a non-positional index and then rerank the selected top results with a positional index. Still there is a large number of versions to go through in Phase 1 even without a positional index.

This paper is focused on processing conjunctive keyword queries on versioned datasets and our key idea is to extend the concept of cluster-based retrieval [1, 28, 30, 40] for representative-guided two-phase search and develop a per-cluster hybrid index to localize data access at Phase 2. Using representatives of document versions with full positional information reduces the number of top clusters needed to retain a good relevancy. The tradeoff is that Phase 2 requires memory caching of index or the use of solid state drives (SSD). To speedup Phase 2 search, we develop hybrid per-cluster indexing with adaptive traversal of forward and inverted structure. Our evaluation shows that the proposed scheme is up-to about 4x as fast as the two-phase approach [22] when the search index is available from memory or from an SSD.

The rest of this paper is organized as follows. Next, we describe the background information and related work. Section 3 discusses design considerations in adopting cluster-based retrieval for searching versioned documents with representative guidance. Section 4 discusses the hybrid indexing and search options for fast Phase 2 query processing. Section 5 presents our experimental results on three real datasets which shows the accuracy and efficiency of the proposed techniques. Finally, Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

Search index can contain positional and non-positional information. The non-positional information indicates if a text word appears in a document or not while the positional information further captures the location of text words appeared in a document. The use of positional information is critical when query relevancy is sensitive to the proximity of query words matched in a document, but it incurs significant search cost. Non-positional indexing for search-

CIKM'16 , October 24 - 28, 2016, Indianapolis, IN, USA

ISBN 978-1-4503-4073-1/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2983323.2983733

ing versioned data is studied in [3, 8, 9, 11, 23, 24, 25]. Numbers in the index are compressed further [2, 35, 41, 45]. One approach for the versioned compact index by Broder et al. [9] considers content sharing patterns among versioned documents with a tree structure and this work is further extended by Herscovici et al. [25] based on multiple sequence alignment. Another approach by Claude et al. is based on run-length, Lempel-Ziv or grammar-based compression with self-indexing [11] and the grammar-based compression such as re-pairing is also used for the document listing problem which searches substrings or phrases in versioned documents [12, 18, 20].

Positional versioned data indexing is studied in [43, 11, 22]. The work of Zhang and Suel [43] uses the content-dependent partitioning method such as Winnowing [34] and 2MIN [38] to divide a versioned document into fragments and then each unique fragment after duplicate detection is only indexed once. The above partitioning technique is related to landmark-based indexing proposed by Lim et al. [29] for efficient index update when document content is changed. We adopt the fragment-based redundant content compression [43, 22] because fragments explicitly capture positional information and also because the work by He et al. [22] shows that it has a higher compression ratio than that of [11]. Grammar-based compression techniques [11, 20, 18, 12] have been used for phrase queries and we are more interested in exploiting more general positional information. It is possible to adopt some of such techniques in version cluster index compression and this can be considered in the future work. While the tree-based compression or sequence alignment technique for content sharing [9, 25] does not address the positional information, some of their ideas are leveraged in our work for computing the posting intersection within each document version cluster.

We illustrate the fragment concept in more details as follows. Once a page is represented by a set of fragments instead of terms directly, the inverted index contains fragments in its compressed data layer. The posting for a term is a set of fragments instead of a page list and there is another data structure that maps a fragment to a set of page IDs that use this fragment. Noted that the above index structure can be augmented with term frequency information. Figure 1(a) is an example of the term-to-fragment index, following the compression scheme in [43, 22]. Each document version is divided into a set of fragments. The inverted index shows a list of fragment IDs that contain a term represented by a term ID. In this example, term "t1" is in fragments f1, f3, and others. Figure 1(b) shows an example of the fragment-to-version reuse table which is the list of page IDs that use this fragment. In this example, fragment f1 is used by document versions v1, v7, and so on.

The above compression for versioned data does make inverted indices more complex and its data layout increases the cost of data traversal during document matching. In [43], the search algorithm first identifies a set of fragments that contain a query term and then uses the fragment-page reuse table to locate versions containing each term and the offset of each occurrence within the versions. The intersection for processing multiple conjunctive query words is arranged accordingly and those version pages which contain all the query terms are scored and ranked. The advantage of using the compact page-fragment-term structure is a substantially smaller index compared to the standard

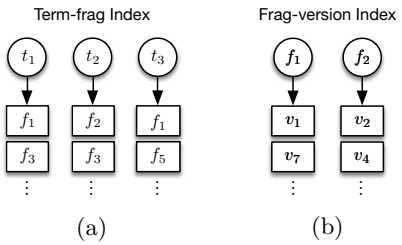

Figure 1: (a) Term-fragment inverted index. (b) Fragment-version reuse table that lists versions sharing each fragment.

document-term inverted index. On the other hand, the intersection of multiple postings takes more resources by going through the term-fragment-page mapping. Searching with this fragment-driven approach is still expensive.

The work by He and Suel [22] extends the fragment-guided two-phrase search based on [33, 43]. Phase 1 uses non-position index while Phase 2 contains detailed position information. Runtime query processing searches non-positional inverted index first to retrieve the top K results and then re-rank these top K results by accessing the full positional index of these versions. The above two-phase search is motivated by the earlier research on cluster-based retrieval [1, 40, 30] for non-versioned documents. Using a non-positional index at Phase 1 is also motivated by relevancy studies [5, 6] and the argument is that having a sufficiently large number of top K at Phase 1 without positional information can still deliver a good relevancy. Still there is a large number of versions to go through even without positional index.

Document ranking using cluster-based retrieval and language models is studied by Liu and Croft [30], and cluster ranking is addressed by Kurland and Krikon [28]. Index optimization for cluster-based retrieval for traditional disk storage is studied in Altingovde et al. [1]. We revisit the cluster-based retrieval techniques as the storage seek overhead of random access has been reduced significantly with today's SSDs. The recent work for searching non-versioned data in Bai et al. [7] considers the use of flash-memory drives to store the per-document forward index. We exploit adaptive use of hybrid cluster-specific index structure.

3. CLUSTER-BASED RETRIEVAL WITH REPRESENTATIVES

3.1 Design Considerations

Our objective is to develop a faster search scheme with much faster query processing time, especially when there is a large number of versions. Our design considerations are discussed as follows. 1) Since versions of documents have highly repetitive content, most likely there exists a version or a composed version that could capture the majority of text features for many versions of each document. Our work is motivated by the cluster-based retrieval [40, 30, 28, 1] which was proposed to rank non-versioned data by exploiting document similarity in clustered results. For versioned datasets, exploiting document similarity can be more important to reduce search time because of high similarity among versions of documents. Thus we adopt the concept of the cluster-based retrieval and consider versions of a document as a group. We compose a representative which captures positional and non-positional information for each version

cluster to facilitate cluster ranking. A phased search can start with a set of representatives instead of the entire document collection. Since the number of representatives is modest after removing the versioning effect, it is not necessary to avoid positional information in Phase 1 index and save index space cost.

2) There is a storage access cost to retrieve cluster-specific information for each selected cluster at Phase 2. For non-versioned data, the work in [1] studies a per-term cluster posting so that the number of disk seeks is controlled as the number of query words. We can extend this work to build per-term index structure for versioned documents, but the repetition of cluster information from one term to another becomes extremely large. Consider the SSD storage with low seek time (e.g. 0.1 ms) is getting popular, we can afford to access a modest number of clusters dynamically.

3) Fragment-based index compresses versioned data significantly while runtime index traversal becomes slow in order to fetch positional information access during Phase 2. This gives opportunities for optimization and we will present a hybrid indexing solution that combines the strength of forward and inverted index.

The two-phase query processing workflow is depicted in Figure 2. The first phase is going through the representative index and retrieving the top k representatives. The second phase is searching on the related index clusters and aggregating the matched results from these clusters. Since each document has many versions, to ensure diversified results, we restrict the number of versions shown for each document in the final result list and a user can select a document and view more versions of the same document if interested.

For final result ranking, we follow the previous work in text ranking with proximity (e.g. [10, 27, 36, 37, 44]). Our study focuses on the construction of index that can provide basic ranking features which can be used to compose scores for these ranking schemes.

3.2 Cluster Representatives

To aid the top k ranking of clusters at Phase 1, we compose a representative for each cluster to provide essential ranking features. The representative for each version group is a superdocument that is artificially composed from a specific version of a document. Our objective here is to create a superdocument that includes terms that appear in all versions of a document while capturing the majority of distance relationship among these terms. The position information of a representative superdocument is built based on the longest version. More specifically, let s_i be the super document for document d_i with m versions. Denote these versions as $v_{i,1}$, $v_{i,2}$ ... $v_{i,m}$. Let $v_{i,l}$ be the longest version for d_i. Let $T(s_i)$ be the set of terms included in super document s_i. Let $Pos(t, s_i)$ be a position information set for term t in $T(s_i)$. The index for representatives with the longest version is defined as

$$t \in T(s_i) \text{ if } \exists j, 1 \leq j \leq m \text{ and } t \in T(v_{i,j}).$$

$$Pos(t, s_i) = \begin{cases} Pos(t, v_{i,l}) & \text{if } t \in T(v_{i,l}) \\ \infty & \text{if } t \in T(s_i) - T(v_{i,l}). \end{cases}$$

Namely for the extra terms in set "$T(s_i) - T(v_{i,l})$" added to the superdocument we will treat the appearance of these terms in an unknown position with the ∞ offset value. In the ranking process for document retrieval, a position with the ∞ value will not contribute to the position related weight in the final score aggregation. We can also consider other options to form a superdocument, for example, based on the latest version. Since a superdocument contains terms from all members of the group, there may be false positives introduced during Phase 1 keyword matching.

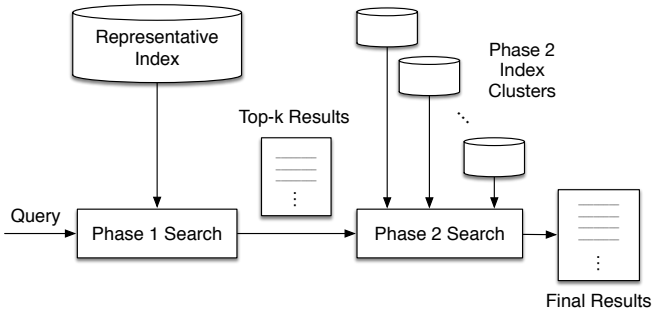

Figure 2: Representative-guided search workflow for query processing

During offline data processing, each document version is divided into fragments using a content partitioning algorithm called TTTD (Two Thresholds Two divisors) based on the work of [17, 31], which is a similar but faster method comparing the ones used in [43, 22]. Since content redundancy among representatives is modest or less significant, we can use the traditional index structure for representatives without using fragments for simplicity and efficiency. Note that the meta information for each version of a document also contains a timestamp. The time data will be useful for user queries containing temporal constraints. The offline index may also be partitioned based on a coarse-grained time interval to optimize the index search if that matches the users' query pattern. Studies on versioned data search with a time range are in [8, 21, 39]. This paper focuses on conjunctive query processing and the time range can be added as a filter in the search process.

4. HYBRID PER-CLUSTER INDEXING AND TRAVERSAL

Phase 2 query processing needs to identify document versions from selected clusters that really match required keywords, and compute a ranking score. It would be more expensive to compute scores for all versions and then filter out those that do not contain all conjunctive keywords. We discuss how to gather basic feature data from the index for intersection and scoring below. The basic feature data includes the positional information of each term in a version and frequency information can be derived from this process if it is not explicitly stored.

As we adopt the fragment-based compression in the per-cluster index with the positional information, one option is that an intersection algorithm uses the term-fragment list and looks up a fragment-version reuse table dynamically to determine if all conjunctive query words included in an version. From our experiments, we find the dynamic conversion is very time-consuming. It is much faster to construct the term-version posting first before performing the intersection operation and how to optimize the traversal of index data structure is the key to accomplish a low search time.

4.1 Indexing and Search Options for Phase 2

We discuss two indexing options first with different time and space tradeoffs in considering the traversal of cluster-specific inverted index or forward index and then design a hybrid option.

- Option A: Each posting list of a term in a version cluster index is composed of fragments and the positions of this term in each fragment. The runtime search process extracts this list from the term-fragment index and traverses a per-cluster fragment-version reuse table to reconstruct a term-version posting with positions for each term. This derived term-version posting includes the positional and frequency information and after that, a multi-keyword intersection is conducted using such postings.

- Option B: Compared to Option A, this option adds the extra storage space overhead to explicitly store the term-version posting of each term, and a version-fragment forward index, but it does not need to maintain the local fragment-version reuse table. As the version posting of each term without positional information is available in advance, the intersection of term postings can be conducted quickly first without a need to dynamically re-construct such postings. Once a set of matched versions is derived through the intersection, the query-time process derives positional information by finding all fragments included in these versions through the forward index, and then by using a binary search on a term-fragment posting to extract the positional information of each term at each document version.

  For example, given query "Marine Science", for each version cluster derived from Phase 1, we directly use the term-version postings of "Marine" and "Science" to conduct an intersection. Assume v_1 and v_2 contain both terms, then we use the version-fragment forward index to identify possible fragments and their positions in v_1 and v_2 that may contain each term. Finally a binary search using the term-fragment postings of "'Marine" and "Science" identifies the text positions in real fragments that truly contain these two words.

A detailed discussion of data structure choices of above two options is in Section 4.3 summarized in Table 1. There is a time and space tradeoff between Option A and Option B. Option B can be faster than Option A in many cases while it does use slightly more space. On the other hand, Option A can outperform Option B some time, for example, queries using rare terms. We model the time cost of Options A and B as follows.

$$TimeCost_A = k \cdot (\Gamma + \sum_{i=1}^{q} f_i \cdot \mu_i \cdot \rho_i \cdot \tau + \Pi), \text{ and}$$

$$TimeCost_B = k \cdot (\Gamma + \Pi + \sum_{i=1}^{q} m \cdot \gamma \cdot (\log(f_i) + \rho_i) \cdot \tau)$$

where k is the number of top clusters selected by Phase 1; q is the number of query terms; f_i is the number of fragments for term t_i's posting at a cluster; m is the number of matched document versions after query intersection; Γ is the average cost of loading a cluster index to memory from a fast storage; Π is the average cost of posting intersection for conjunctive keywords; μ_i is the average number of document versions using a fragment that contains term t_i; ρ_i is the average number of positions of term t_i in one fragment; γ is the average number of fragments included in one version. τ is the average memory access time.

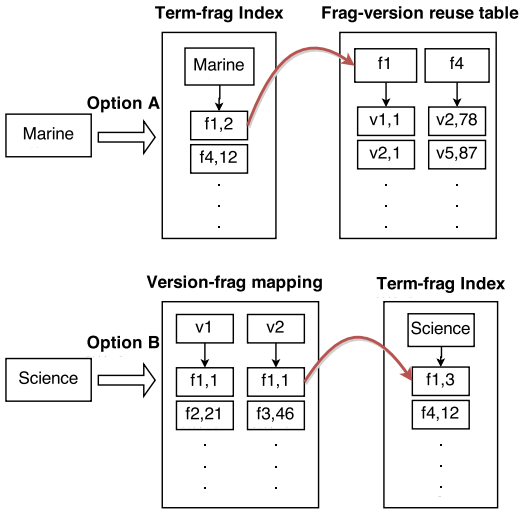

Figure 3: An example of data traversal in Option C that selects the Option A or Option B approach for each query word at each cluster.

The index data for each cluster is organized separately and the runtime system loads the data k times for the selected top k clusters from a fast solid state drive. For our tested applications, the data size for each cluster to be load is less than 5KB and the per-cluster index I/O time Γ is about 0.28 milliseconds or less using SSDs. The overhead is not overwhelming when fetching the index data for top few hundred clusters. Π represents the cost of intersection and we discuss this part in next subsection.

From the cost model of options A and B, one can infer that if a query term t_i is rare, and f_i is very small, this case favors the use of Option A. On the other hand, if t_i is a popular word, and f_i is very large, this case favors Option B. We are devising Option C that compares the relative cost ratio of Options A and B for extracting positional information and adaptively select one of them based on the characteristics of each term t_i searched in each cluster. The query-time work flow of Option C is summarized in Algorithm 1.

Figure 3 illustrates an example of Option C in processing the query "Marine Science". First, the intersection finds v1 and v2 contain "Marine" and "Science". For keyword "Marine", the threshold condition leads to the use of Option A traversal. From the fragment posting of "Marine", a list of fragments are found and they correspond to versions v1, v2, v5 and so on. Only v1 and v2 are acceptable from the intersection results, then the position at v1 is computed as 3 and the position at v2 is computed as 78+12 which is 90. For keyword "Science", a different path is taken: from version-fragment mapping, v1 contains a list of fragments, such as f1 and f2. By doing a binary search on the Term-frag list of "Science", we know in version v1 only f1 contains

```
1  Load the cluster index and accumulate statistics.
2  Perform the intersection of version postings.
3  for each query word t_i do
4  |  If f_i · μ_i · ρ_i ≤ m · γ · (log(f_i) + ρ_i),
5  |  Use the term-fragment posting of t_i to get
   |  fragments and term positions within fragments.
6  |  Convert into term positions within all versions
   |  by accessing the fragment reuse table.
7  |  Else
8  |  If needed, set L as a list of fragments for all
   |  matched versions.
9  |  Search t_i's fragment posting to find term
   |  positions for the fragments in L.
10 |  Derive term positions within matched versions.
11 end
```

Algorithm 1: Option C that extracts position and other meta information of the matched versions within each cluster.

"Science" and position 4 is derived at version v1. We do the same procedure with v2 and position 4 is derived at version v2.

It should be noted that Option C's decision path is cluster dependent. Namely this method selects Option A or B based on the difference of $f_i \cdot \mu_i \cdot \rho_i$ and $\cdot m \cdot \gamma(\log(f_i) + \rho_i)$ for each cluster. Thus for the same query word, one cluster can make a selection different from another cluster. The time cost of Option C can be modeled as

$$k \cdot (\Gamma + \Pi + \sum_{i=1}^{q} min(f_i \cdot \mu_i \cdot \rho_i, \quad m \cdot \gamma(\log(f_i) + \rho_i)) \cdot \tau)$$

which is usually smaller than that of Option A or Option B.

4.2 Term-version Posting Intersection

Term-version posting intersection without position information is used in Options B and C and we can leverage the ideas of the work in [9, 25]. When the number of versions is modest for each version cluster, the intersection to identify versions of a document containing all keywords can be extremely fast by representing the posting using a bit operation [13]. For a large number of versions, the intersection with bit operations is less effective. Since versions of a document tend to be similar, when a word appears in one version, it appears in another version with a good probability unless this word is added to a version and is deleted shortly after a few versions. With this data characteristic in mind, a long bit vector can be represented succinctly by a small set of intervals. Each interval represents consecutive version numbers that contain this keyword.

When each version posting is represented by a set of intervals, the intersection algorithm of posting intervals from multiple words can be conducted by modifying the integer set intersection algorithm by Culpepper and Moffat [13] with a change in Golomb search to accommodate the interval boundary comparison. For interval list intersection of two words, assume one word has n_1 non-overlapping intervals and anther word has n_2 non-overlapping intervals where $n_1 < n_2$. The maximum number of new intervals produced as the result of intersection is $n_1 + n_2 - 1$. The time cost of this intersection is $O(n_1 \log \frac{n_1+n_2-1}{n_1})$. For q word intersection, the corresponding bit-vectors have $n_1, \cdots n_q$ intervals with $n_1 \leq \cdots \leq n_q$. The total number of subintervals produced can be high, but less than H where $H = min(V, \sum_{i=1}^{q} n_i - 1)$ and V is the total number of document versions in a cluster. The total time cost is bounded by $\Pi = O((q-1)H \log \frac{H}{n_1})$. Because $n_1, n_2, \cdots, n_q$ tend to be small in a cluster index, the time cost Π is not significant. Note that the hashing-based intersection [14] does not work well as the intersection of two intervals requires an inequality comparison of interval boundary values. Skip-based synchronization points [13] may be used with extra space while adding them has not given a visible improvement for this problem in our experiments.

4.3 Index Storage Layout and Cost

We discuss the data layout for the cluster specific index structures used in the above three options, and assess the space cost difference in an approximated manner. The sequence of numbers used in each data structure is further compressed by storing their delta gaps and using one of number compression methods with fast query-time decompression such as var-Byte, Simple-9, PforDelta, and Opt-PFD [35, 45, 42, 41]. Through experimentations, we find that a combination of Simple-9 [2] and var-Byte gives a competitive compression performance in our context.

We estimate the storage need of each index option approximately before applying the aforementioned number sequence compression. This gives a rough space assessment assuming the number compression brings is at a similar level of space reduction proportionally. Since small integer values typically use less space after number compression, and to improve the accuracy of the space cost estimation, we associate the integer size as a coefficient. Document IDs and word IDs need a 4-byte integers in general, we assume that the version numbers of each document and position numbers need integers with 1 to 2 bytes. Also we perform the fragment ID localization under each cluster so that local fragment IDs may be ranged with 2 bytes or less.

We will use the following additional symbols in addition to ones used in time cost analysis. W: the number of distinct words at a cluster. R: the number of fragments at a cluster. V: the version number at a cluster. μ: the average number of versions using a fragment at cluster. ψ: the average number of fragments per term at a cluster. β: the average size of the posting bitvector of a term discussed below.

Term-to-version posting bitvectors: The extra term-to-version posting for Options B and C records version IDs that contain a term without positional information. When V is not too large, an internal bit vector representation is appropriate. When V becomes modestly large, we consider a hybrid compression as follows. Since versions of a document are similar, either many versions shared the same words or they have little in common for other words used in the cluster. The characteristic of a posting bit vector for our version dataset is that each vector either contains lots of 1's or lots of 0's. We can either use a few intervals to represent a bit vector or follow a hierarchical compression scheme from [19]. The root of a tree structure in [19] can use bit value 1 to represent a large consecutive number of 1's when 1 is dominating a bit vector. Otherwise the root uses 0 to represent a large consecutive number of 0's. The space for bit vectors of W terms is $\beta \cdot W$.

Term-to-fragment posting: This posting contains a fragment list and term positions at each fragment. Because

most of terms appear in a fragment once, we follow the idea from [43, 22] to store a sequence of fragment and position pairs. The sequence of numbers for each entry in this index is: (term ID, meta information, fragment ID, position, fragment ID, position, $\cdots$). The meta information represents the number of pairs and other control flags. The size of term-to-fragment index is $(5 + 4\psi) \cdot W$.

Local fragment-to-version reuse table: Following the same strategy for a term-to-fragment posting, we record the number of version and position pairs in the sequence: (fragment ID, meta information, version ID, position, version ID, position, ...). Total size is $(3 + 4\mu) \cdot R$.

Version-to-fragment mapping: Similarly, the number sequence of an entry in this mapping is: (version ID, meta information, fragment ID, position, fragment ID, position, $\cdots$). The total size of the forward index is $(3 + 4\gamma) \cdot V$.

Considering the total number of fragment occurrences in a local reuse table or in a forward mapping index across all the document versions remains a constant in a cluster, we can show that $\mu R = \gamma V$. Thus, the version-fragment mapping and fragment-version mapping have very similar size.

Following data structure choices used in each option, the total space cost of each option for each cluster before number compression is estimated as:

$$SpaceCost_A \approx ((5 + 4\psi)W + 3R + 4\gamma V),$$

$$SpaceCost_B \approx ((\beta + 5 + 4\psi)W + 3V + 4\gamma V),$$

$$SpaceCost_C \approx ((\beta + 5 + 4\psi)W + 3R + 3V + 8\gamma V).$$

There is some additional space need for meta information such as version document length, which is less significant. The difference ratio between Options A and B is approximately

$$\frac{\beta W + 3V - 3R}{(5 + 4\psi)W + 3V + 4\gamma V}.$$

The difference ratio between Options B and C is approximately

$$\frac{4\gamma V + 3R}{(\beta + 5 + 4\psi)W + 3R + 4V + 8\gamma V}.$$

Note that $R << \gamma V$ assuming high redundancy among versions of the same document. From the experiment data we have tested, $\psi \approx 2$ and γ is in between 13 and 20. When $V << W$, $\beta \approx 3$ and the difference between A and B is about $\frac{\beta}{5+4\psi}$ which is 23% and the difference between B and C is small. When $V >> W$, the difference between A and B is small and the difference between B and C is close to 50%.

Table 1 summaries the data structure choices in three options. Option C is faster than Option A or Option B while incurring a modest increase in storage cost.

Per-cluster data structure	Option A	Option B	Option C
Posting bitvectors	no	yes	yes
Term-to-frag. index	yes	yes	yes
Frag.-to-version reuse table	yes	no	yes
Version-to-frag. mapping	no	yes	yes

Table 1: Data structure choices for three options.

How much does storage space overhead increase by building separate index for each cluster? We compare space cost difference between Option C and a corresponding global index with no cluster separation. Since there are full or partial duplicates among documents, the separation of index by clusters has a space disadvantage that some of term IDs appear redundantly in the local term-to-fragment postings. Some of fragment IDs also appear redundantly in both term-to-fragment postings and the local reuse tables. On the other hand, the fragment IDs after localization and version IDs in a cluster index uses less number of bytes compared to a global index and we assume a 1:2 ratio for fragment IDs, version IDs but still 1:1 ratio for positions. The global index space is modeled

$$SpaceCost_{global} \approx \beta W n + 5M + 6\delta\psi W n + (6\delta R + 6V + 12\gamma V)n$$

where n is the number of documents; M is the number of distinct words globally; δ is the ratio of fragments which are not shared among documents. In our test datasets, $M \approx 0.05nW$ and δ is between 0.7 and 1. V, W, and R have the same meaning as before, except it is an average number per cluster. Thus the space difference ratio between global index and option C is

$$\frac{[5(1 - \frac{M}{nW}) + (4 - 6\delta)\psi] \cdot W + (3 - 6\delta)R - 3V - 4\gamma V}{(\beta + 5 + 4\psi)W + 3R + 3V + 8\gamma V}.$$

When $V >> W$, the difference ratio is about $\frac{-3-4\gamma}{3+8\gamma}$ which is close to -50%. Namely the global index is about 50% larger. When $V << W$, the difference is about $\frac{5+(4-6\delta)\psi}{\beta+5+4\psi}$ and thus the global index is up-to 28.8% smaller.

5. EVALUATIONS

5.1 Settings

Our evaluation objective is 1) to demonstrate the benefits of searching versioned data with cluster-based retrieval in terms of query search time difference, 2) to compare the index options in building the Phase 2 index in terms of time and space cost, 3) to assess the impact of relevance for using a core representative index with positional information. In this context, we study the impact of using different representatives. We have developed a prototype to implement the proposed approach with C++. Code is compiled with G++ using optimization flag -O1. Experiments are conducted on a Linux CentOS 6.6 server with 8 cores of 3.1GHz AMD Bulldozer FX8120, 16GB memory, Kingston HyperX 3K 240GB SSD and a 1TB Western Digital Caviar Black hard disk drive (HDD) with 7200 RPM. All experiments store the index data in the SSD except Table 3 which compares time performance when the search index is stored in HDD and in SSD.

Since there are no standard benchmarks for versioned search, we have crawled the following three versioned datasets for evaluation. 1) The first one is from Wikipedia (called Wiki), which consists of 4.1 million articles and on average each distinct document contains 13 versions. The versions are archived in a monthly crawled gap from April 2013 to April 2014. On average each document version has 1.9 KB and each fragment has 156 bytes. 2) The second dataset is a web dataset (called Web) of 5 million pages and there are 20 versions per document on average. The dataset was crawled in 2014 from two university domains. After removal of HTML tags, each document has 3.1KB and each fragment has 191 bytes on average. 3) The last dataset is from GitHub

(called GitHub) with 8.8 million versioned documents in total. Those are Linux code documents dated from April 2005 to April 2014 with 439 versions per document. Searchable text is extracted from the source code using function names and embedded comments, and each document corresponds to one source file. After pre-processing, each document has 3.35 KB and each fragment has 167 bytes on average.

Ranking function. Following the previous text ranking techniques (e.g. [10, 27, 37, 44]), the text feature in our implementation leveraging non-positional information is standard Okapi BM25, differentiated by where they appear in the fields of a document such as title and body. We also use a text proximity feature that leverages positional information: the length of a minimum text *span* [36] to cover query words at each field, scaled by the percentage of query words covered. The overall ranking score linearly combines weighted BM25 and proximity features.

We compare our Representative-guided Two-phase Search (RTP) with the following approaches: 1) One phase search (OP). The implementation is based on [43] using the positional index with fragments. This is essentially the same as Phase 1 of RTP except that OP searches the entire index. 2) Two phase search (TP). This is based on [22] and the Phase 1 implementation ranks top-K results with non-positional index. Phase 2 re-ranks the selected top K results using the positional index with fragments. For RTP, we select the number of representatives k ranked at Phase 1 between 10 to 100 to produce good final top 10 results as an answer. For TP, K is chosen to be $K = k * V$ so that there are enough good results selected from each cluster at Phase 1, which allows re-ranking at Phase 2 to produce relevant results competitive to RTP. The three datasets have different V values, affecting K values and performance difference of TP and RTP in addition to relevancy. We will also show the relevance results of RTP and TP with different k and K values.

For search time measurement, we report the average query processing time for each query set through 25 runs when excluding or including the index load overhead from a disk drive. We have generated 500 synthetic queries for each dataset with query length distribution following a query log study in [4] for the purpose of performance assessment.

For relevance, the standard relevance metric for document search is NDCG [26]. Since there are many versions for each document with similar content and some of which can be considered as near duplicates, showing too many results per document would affect result diversity [32]. Thus we restrict the displaying of top results and only show v versions per document. A user can request for accessing more versions from a selected document when needed. We expect v to be 1 or 2 at most in practice. In our evaluation, we use $v = 1$ and collect the NDCG value at top 10 positions. We call the modified NDCG score as vNDCG1@10.

5.2 A Comparison on Overall Search Time

For fair comparison, to prepare same number of candidate pages in Phase 2 for TP and RTP, we set $K = k \cdot V$. We use $K = 2000$ and $k = 100$ for the Web with $V = 20$, $K = 1000$ and $k = 77$ for Wiki with $V = 13$, and $K = 5000$ and $k = 12$ for GitHub with $V = 439$. Table 2 lists the time cost of RTP, OP and TP in milliseconds when the search index is kept in memory. OP that navigates all versions of documents is much more time consuming than TP and RTP. For example, it takes up-to 156x more time than RTP. In terms of TP vs. RTP, for GitHub with a large number of versions, RTP is about 4.12x as fast as TP. On the other hand, for the Web and Wiki datasets with a modest number of versions, RTP is about 3.99x as fast as TP on Web data and 3.83x on Wiki data.

Time (ms)		OP	TP	RTP
Web	Phase1	5899	34.56	16.26
	Phase2	0	115.5	21.36
	Total	5899	150.0	37.62
Wiki	Phase1	1047	5.378	3.868
	Phase2	0	119.9	28.81
	Total	1047	125.3	32.68
GitHub	Phase1	5846	46.50	2.394
	Phase2	0	539.0	139.7
	Total	5846	585.5	142.1

Table 2: Query processing time in milliseconds when the search index is preloaded to memory.

Time (ms)		OP	TP	RTP
Web	SSD	5901	153.4	67.28
	HDD	5950	252	938.6
Wiki	SSD	1049	128.7	55.92
	HDD	1098	227.3	738.2
GitHub	SSD	5848	588.9	147.4
	HDD	5897	687.5	303.6

Table 3: Query processing time in milliseconds including the index load cost from an SSD or HDD.

Table 2 also lists the cost distribution of Phase 1 and Phase 2 time in detail. RTP's Phase 1 is faster than that of TP because RTP's Phase 1 search scope focused on representatives is much smaller while Phase 1 of TP searches the index of all document versions even positional information is skipped. Phase 1 of RTP has more time advantage for the GitHub dataset in which there are more versions per document. Note that TP's Phase 1 time reported here is higher than "a few ms" reported in [22]. This is because K parameter selected for Phase 1 in our experiment is larger in order to improve relevance.

Comparing TP and RTP's Phase 2 time, RPT is 5.41x as fast as TP on Web dataset, 4.16x on Wiki dataset and 3.86x on GitHub dataset. While our optimization plays a significant role, one reason is that extracting the positional information in [22] uses a global term-fragment index. Following our analysis on Option B in Section 4.1, if we replace cost of searching local reuse table to a global reuse table, then the cost of position information extraction increases by a ratio of $\frac{\log F_i}{\log f_i}$ where F_i is the average posting length of global term-fragment index, which is much larger than f_i. This corroborates a benefit of cluster-based retrieval.

Table 3 shows the total processing time per query including the index loading overhead from an SSD or HDD. The gap between TP and RTP is narrowed because RTP has to access the index per version cluster separately and this results in more random disk block reads. Fortunately the fast SSD seek time for random I/O still allows RTP to outperform others. RTP is 2.28x as fast as TP on the Web dataset, 2.30x on Wiki and 3.99x on GitHub. RTP is 87.7x as fast as OP on the Web dataset, 18.8x on Wiki and 39.7x on GitHub. The SSD I/O cost is about 44%, 41%, or 4% of

the overall time, respectively for these datasets. Searching the GitHub dataset incurs the smallest I/O cost percentage because the number of versions per cluster is the highest and its k value is the smallest. The above I/O cost may be further reduced in the future with more code optimization. When the index resides in HDD, OP is still the slowest because its slow performance in Phase 1. RTP is 6.34x as fast as OP on the Web dataset, 1.49x on Wiki and 19.4x on GitHub. Comparing to TP, the time advantage of RTP diminishes in Phase 2 because of high random I/O cost. For GitHub with 439 versions per document, RTP is 2.26x as fast as TP. For Web wth 20 versions per page and Wiki with 13 versions per document, TP is 3.72x and 3.25x as fast as RTP. Thus we recommend RTP to be used when the search index can be stored in an SSD. It is suitable for HDDs only when handling a very large number of versions.

5.3 A Comparison of Phase 2 Indexing and Traversal Options

Time (ms)	Position extraction	Intersection	Scoring
Web	9.570	0.24	11.55
Wiki	15.87	0.25	12.69
GitHub	96.47	1.44	41.77

Table 4: Cost distribution of RTP at Phase 2

Table 4 shows the cost distribution of RTP in-memory search time at Phase 2. The feature extraction time is significant and this demonstrates the importance of reducing conversion time in Phase 2 computation.

Time (ms)	Web	Wiki	GitHub
Option A	35.74	20.81	128.5
Option B	9.778	23.57	124.6
Option C	9.57	15.87	96.47
Cluster-choice A	36.34%	91.67%	89.37%
Optimum	8.472	13.55	77.67

Table 5: In-memory search time with different options for Phase 2.

Table 5 lists the average query processing time of Phase 2 with various options. Row marked "Optimum" is computed by choosing the minimum value among Option A and B for each query in searching each cluster. The switch condition listed in Algorithm 1 may not find the best choice in all cases and thus this row represents the lower time bound an optimum algorithm may achieve. Row marked "Cluster-choice A" lists the percentage of the clusters that choose Option A as the traversal method decided by Option C. Option B can be 265% faster than Option A for Web, but can be 13% slower for Wiki. Option C adaptively predicts the winner between Option A and B, and is getting closer to what the optimum can accomplish. Compared to Option A and B, Option C is up-to 273% faster for Web , 48.5% for Wiki, and 33.2% for GitHub.

Why is Option A slower than Option B even 89.37% of clusters choose Option A traversal? The reason is that choosing Option A gives an improvement of 0.5523ms per cluster than choosing Option B in this case. In comparison, 10.63% clusters choose Option B, which gives an improvement of 5.484ms per cluster than choosing Option A. In general, we find that choosing Option A often delivers a relatively smaller time reduction while taking Option B often yields a bigger reduction. One reason is that Option B is often chosen when handling popular words, which tend to carry more shares of the overall search cost. For Wiki, choosing Option A reduces time by 0.0430ms per cluster while time reduction is 0.304ms per cluster when choosing Option B. For Web, time reduction 0.01191ms per cluster than choosing Option A and it is 0.8575ms per cluster when choosing Option B.

Index(MB)	Web	Wiki	GitHub
Option A	907	614	264
Option B	1225	905	452
Option C	1411	1008	654
Global	1237	1044	991

Table 6: Phase 2 index size of different options

Index(MB)	Web	Wiki	GitHub
Posting bitvectors	323	294	155
Term-fragment index	721	511	62
Fragment-version reuse table	186	103	202
Version-fragment mapping	181	100	235

Table 7: Phase 2 index size of different structures

Table 6 shows the compressed Phase 2 index storage size in megabytes under different options. Note that the uncompressed index size can be upto an order of magnitude larger than the numbers reported here. The row marked with "Global" combines all data structures without cluster separation as explained at the end of Section 4.3. The order of the space usage is Option C, Option B and Option A. Option C does use more space for accomplishing the fastest processing time. For Web and Wiki datasets in which V is modest, the space difference ratio between Option A and Option B is about 26% and 32%. This is close to our storage cost analysis in Section 4.3 which estimates an approximated difference bound as 23%. From GitHub's result in which the number of version is large, the space difference ratio between Option B and Option C is about 31%. That is also within the estimated upper bound 50%.

Option C has similar cost as the global index for Wiki. For GitHub, the global index takes about 50% more space than Option C, which is about the same as what the space anlysis has estimated when V is very large. For Web with a modest number of versions, the global index uses 12.3% less space than Option C, which does not exceed the estimated upper bound 28.8%.

Table 7 shows the size of different components in our index. For Web and Wiki's results with a modest number of versions, term-to-fragment index is the largest component. On the other hand, for GitHub dataset, term-to-fragment index becomes less significant. That is because the fragment-based compression [43] becomes more effective for a large number of versions.

5.4 A Comparison on Relevance Scores

For single-word queries, the relevance of RTP is about the same as that of TP and OP and we report the results of relevance evaluation on queries with two or more words. We

have randomly sampled 50 queries per dataset with a distribution following the log study in [4]. Here are some sample queries on the three datasets: 1) Web: dentist insurance, teaching assistant salary, student research grant application, international student center visa application; 2) Wiki: heart disease, England football team, second world war death, united nations security council members; 3) GitHub: Ethernet adapter, virtual memory access, virtual device data block, Linux kernel boot load device driver. There are four students involved in rating the final results of the different search methods with a score leveled from 0 to 3. Here relevance level 3 means the selected version is perfect for answering the query and level 0 means irrelevant. The names of search methods are not revealed to the evaluators as we union results from all algorithms together. Thus there is no bias in the rating process.

The default representative of RTP uses a superdocument that starts from the longest version (SLO) as the basis and then includes all words from versions. We have compared another way of selecting the representatives: the superdocument starts from the latest version(SLA).

vNDCG1@10		Web	Wiki	GitHub
OP		0.6478	0.7012	0.6889
K=200, 130, 4390 (k=10)	TP	0.4268	0.4833	0.5166
	RTP	0.6157	0.6460	0.6137
K=400, 260, 8780 (k=20)	TP	0.5856	0.6364	0.6253
	RTP	0.6557	0.6974	0.6664
K=1000, 650, 21950 (k=50)	TP	0.6460	0.6888	0.6780
	RTP	0.6560	0.6988	0.6782
K=2000, 1300, 43900 (k=100)	TP	0.6468	0.6912	0.6823
	RTP	0.6562	0.6988	0.6868

Table 8: Relevance scores of RTP compared with the other methods in terms of vNDCG@10 for different k values and K=k*V

Table 8 lists the vNDCG1@10 results of RTP with SLO method compared to OP and TP for the three datasets. The table lists the number of top documents (K) selected at Phase 1 for TP and the corresponding number of top clusters (k) selected at Phase 1 at RTP. For example, "K=200, 130, 4390 (k=10)" means that TP selects top 200, 130, and 4390 for three datasets Web, Wiki and GitHub respectively while RTP selects top 10 clusters which contain K pages in total. For a smaller K (and k), RTP is doing better than TP by taking the proximity into account earlier. Still both RTP and TP have an insufficient coverage of relevant results in Phase 1 and thus have a relatively lower score compared to OP. When increasing the number of top results in Phase 1, the relevancy gap between TP and RTP becomes smaller and also both are getting closer to OP.

Figure 4 depicts the vNDCG1@10 scores of different representative selection options when k is 20 and SLO is the best choice among these options. The relevance score of SLO is 7.5% higher than SLA for Web, 1.3% lower for Wiki, and 0.7% lower for GitHub. On average, SLO is better than SLA by 1.83%. We also assess the impact of adding all words from versions to the superdocuments. LA means using the latest version without adding words from other documents. LO means using the longest version without adding words from other documents. For all three datasets, the superdocument-based selection (SLA or SLO) is in general more effective than LA or LO. For the Wiki dataset, SLA is 4.39% better than LA and SLO is 0.87% better than LO. For the Web

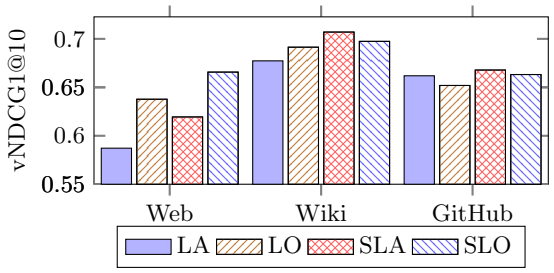

Figure 4: Impact of representative selection on vNDCG1@10 relevance scores

dataset, SLA is 5.48% better than LA and SLO is 4.39% better than LO. For GitHub data, SLA improves 0.89% over LA and SLO improves 1.73% over LO. The longest version is more effective in representing the positional information. Adding the extra terms in a super version provides 2.33% relevance improvement on average.

Ratio	LA	LO	SLA	SLO
Web	50.32%	95.19%	100.00%	100.00%
Wiki	94.26%	98.73%	100.00%	100.00%
GitHub	92.64%	96.60%	100.00%	100.00%

Table 9: Word coverage of the four representative selection methods

Table 9 shows the average distinct word coverage ratio (AWCR) for the four representative selection algorithms. Word coverage ratio of a document group is calculated by dividing the total number of distinct words in a document representative by that of all versions of this document. AWCR is the average word coverage ratio of all documents. Since SLA and SLO have a non-positional index which covers words from all versions of a document, the AWCR value of SLA and SLO is 100%. LO has a word coverage 89% higher than LA for the Web dataset, which indicates SLO has a significantly better positional information coverage than SLA. This explains why vNDCG score of LO is 8.6% better than LA and the score of SLO is 7.5% better than SLA in Figure 4. For other datasets, the AWCR value of LA is closer to LO in Table 9 and that explains the relevance score of SLA and SLO is close in Figure 4.

6. CONCLUDING REMARKS

The main contribution of this paper is a hybrid indexing method with adaptive runtime traversal in supporting fast two-phase versioned data search and an integration with cluster-based retrieval using guided representatives. Our evaluation with a prototype implementation using three datasets shows the following results.

1) RTP has a significant efficiency advantage on SSDs. It is suitable for HDDs only when there is a very large number of versions. RTP can be 3.83x to 4.12x as fast as the TP method if the search index is in memory. When the overhead of loading the index from an SSD is included, RTP can be 2.28x to 3.99x as fast as TP. Both approaches can be one or two orders of magnitude faster than a classical one-phase algorithm on SSDs while delivering competitive relevancy with a proper choice of top K or k value.

2) The hybrid index with adaptive traversal (Option C) can be up-to 273% faster than Options A and B in Phase

2 in-memory query processing. Option C design represents a time-space tradeoff as Option A can use up-to 59.6% less compressed space while Option B can use up-to 30.9% less. On average, the space cost of Option C is more or less comparable to that of a global index.

The proposed work is focused on conjunctive queries and one future study is to consider disjunctive queries. Another future study is to investigate the incremental index update with time-based partitioning. When a new version is added, fragments shared with other document versions need to be identified and an approximation under a certain time interval may be applied for cost reduction. The Phase 1 index may be changed following the traditional index update techniques if the cluster representative changes and the update for a cluster index is fairly local.

Acknowledgments. We thank the anonymous referees for their thorough comments. This work is supported in part by NSF IIS-1528041 and IIS-1118106. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

7. REFERENCES

[1] I. S. Altingovde, E. Demir, F. Can, and O. Ulusoy. Incremental cluster-based retrieval using compressed cluster-skipping inverted files. *ACM Trans. Inf. Syst.*, 26(3):15:1–15:36, 2008.

[2] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. of 15th Australasian Database Conference*, pages 61–67, 2004.

[3] P. G. Anick and R. A. Flynn. Versioning a full-text information retrieval system. In *SIGIR*, pages 98–111, 1992.

[4] A. Arampatzis and J. Kamps. A study of query length. In *Prc. of ACM SIGIR*, pages 811–812, 2008.

[5] D. Arroyuelo, S. González, M. Marin, M. Oyarzún, and T. Suel. To index or not to index: Time-space trade-offs in search engines with positional ranking functions. In *Proc. of 35th ACM SIGIR*, pages 255–264, 2012.

[6] N. Asadi and J. Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proc. of 36th ACM SIGIR*, pages 997–1000, 2013.

[7] J. Bai, J. Pedersen, and M. Yang. Web-scale semantic ranking. In *Proceedings of the 2014 SIRIP*, 2014.

[8] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. A time machine for text search. In *Proc. of 30th ACM SIGIR*, pages 519–526, 2007.

[9] A. Z. Broder, N. Eiron, M. Fontoura, M. Herscovici, R. Lempel, J. McPherson, R. Qi, and E. J. Shekita. Indexing shared content in information retrieval systems. In *EDBT*, volume 3896, pages 313–330, 2006.

[10] S. Büttcher, C. L. A. Clarke, and B. Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *Proc. of 29th ACM SIGIR*, pages 621–622, 2006.

[11] F. Claude, A. Fariña, M. A. Martinez-Prieto, and G. Navarro. Indexes for highly repetitive document collections. In *Proc. of 20th ACM CIKM*, pages 463–468, 2011.

[12] F. Claude and J. I. Munro. Document listing on versioned documents. *LNCS*, 8214:72–83, 2013.

[13] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29(1):1:1–1:25, Dec. 2010.

[14] B. Ding and A. C. König. Fast set intersection in memory. *Proc. of VLDB*, 4(4):255–266, Jan. 2011.

[15] L. DuBois and M. Amaldas. Building the Case for Moving Compliance, eDiscovery, and Archives to the Cloud., June 2011.

[16] EMC. Archive solutions for the enterprise with emc isilon scale-out nas. http://www.emc.com/collateral/white-papers/h11224-archive-solutions-enterprise-emc-isilon-wp.pdf, December, 2012.

[17] K. Eshghi and H. K. Tang. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Hewlett-Packard Labs. Technical Report, TR 2005-30, 2005.

[18] H. Ferrada and G. Navarro. A Lempel-Ziv compressed structure for document listing. *LNCS*, 8214:116–128, 2013.

[19] A. S. Fraenkel, S. T. Klein, Y. Choueka, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. In *Proc. of 9th ACM SIGIR*, pages 88–96. ACM, 1986.

[20] T. Gagie, K. Karhu, G. Navarro, S. J. Puglisi, and J. Sirén. Document listing on repetitive collections. *LNCS*, 7922:107–119, 2013.

[21] J. He and T. Suel. Faster temporal range queries over versioned text. In *Proc. of 34th ACM SIGIR*, pages 565–574, 2011.

[22] J. He and T. Suel. Optimizing positional index structures for versioned document collections. In *Proc. of ACM SIGIR*, pages 245–254, 2012.

[23] J. He, H. Yan, and T. Suel. Compact full-text indexing of versioned document collections. In *Proc. of 18th ACM CIKM*, pages 415–424, 2009.

[24] J. He, J. Zeng, and T. Suel. Improved index compression techniques for versioned document collections. In *Proc. of 19th ACM CIKM*, pages 1239–1248, 2010.

[25] M. Herscovici, R. Lempel, and S. Yogev. Efficient indexing of versioned document sequences. In *ECIR*, pages 76–87, 2007.

[26] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, Oct. 2002.

[27] K. S. Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: development and comparative experiments. *Inf. Process. Manage.*, 36(6):779–808, Nov. 2000.

[28] O. Kurland and E. Krikon. The opposite of smoothing: A language model approach to ranking query-specific document clusters. *J. Artif. Int. Res.*, 41(2):367–395, May 2011.

[29] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Agarwal. Dynamic maintenance of web indexes using landmarks. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 102–111. ACM, 2003.

[30] X. Liu and W. B. Croft. Cluster-based retrieval using language models. In *Proc. of 27th ACM SIGIR*, pages 186–193, 2004.

[31] T.-S. Moh and B. Chang. A running time improvement for the two thresholds two divisors algorithm. In *Proc of 48th ACM Ann. Southeast Regional Conf.*, pages 69:1–69:6, 2010.

[32] D. W. Oard, J. R. Baron, B. Hedin, D. D. Lewis, and S. Tomlinson. Evaluation of information retrieval for e-discovery. *Artif. Intell. Law*, 18(4):347–386, Dec. 2010.

[33] Y. Rasolofo and J. Savoy. Term proximity scoring for keyword-based retrieval systems. In *In Proc. of the 25th European Conf. on IR Research*, pages 207–218, 2003.

[34] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. of ACM SIGMOD*, pages 76–85, 2003.

[35] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of 25th ACM SIGIR*, pages 222–229, 2002.

[36] K. M. Svore, P. H. Kanani, and N. Khan. How good is a span of terms?: exploiting proximity to improve web retrieval. In *Proc. of 33rd ACM SIGIR*, pages 154–161, 2010.

[37] T. Tao and C. Zhai. An exploration of proximity measures in information retrieval. In *Proc. of 30th ACM SIGIR*, pages 295–302, 2007.

[38] D. Teodosiu, N. Bjorner, Y. Gurevich, M. Manasse, and J. Porkka. Optimizing file replication over limited bandwidth networks using remote differential compression. Microsoft Research TR-2006-157, 2006.

[39] L. H. U, N. Mamoulis, K. Berberich, and S. Bedathur. Durable top-k search in document archives. In *Proc. of ACM SIGMOD*, pages 555–566, 2010.

[40] E. M. Voorhees. The cluster hypothesis revisited. In *Proc. of 8th ACM SIGIR*, pages 188–196, 1985.

[41] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. of 18th Inter. Conf. on World Wide Web*, pages 401–410, 2009.

[42] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. WWW '08, pages 387–396.

[43] J. Zhang and T. Suel. Efficient search in large textual collections with redundancy. WWW '07, pages 411–420.

[44] J. Zhao and J. X. Huang. An enhanced context-sensitive proximity model for probabilistic information retrieval. SIGIR '14, pages 1131–1134. ACM, 2014.

[45] M. Zukowski, S. HÃl'man, N. Nes, P. A. Boncz, M. Zukowski, S. HÃl'man, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proc. of IEEE ICDE*, page 59, 2006.