Distribution by Document Size

Andrew Kane
arkane@cs.uwaterloo.ca

Frank Wm. Tompa
fwtompa@cs.uwaterloo.ca

University of Waterloo
David R. Cheriton School of Computer Science
Waterloo, Ontario, Canada

ABSTRACT

Search engines split large datasets across multiple machines using document distribution. Documents are typically distributed randomly to produce good load balancing. We propose that documents be distributed by their size instead. This can make load balancing more difficult, but it produces immediate improvements in both index size and query throughput. To support our proposal, we show improvements to an in-memory conjunctive list intersection system running on the GOV2 dataset using simple16 compression combined with either skips or bitvectors. While our list intersection system does not implement ranking, we also expect significant performance improvements from using document size distribution in ranking based search systems.

In addition, implementations can be adapted to produce further performance improvements that exploit the document size distribution. We present some examples that apply to a full ranking based search engine and leave their verification for future work.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*

General Terms

Algorithms, Performance

Keywords

Information Retrieval, Distribution, Algorithms, Performance, Efficiency, Optimization, Compression, Intersection

1. INTRODUCTION

In order to quickly process a query in a large search engine, the processing must be done in parallel across many machines. In practice, this means splitting the dataset into partitions (or shards), which contain non-overlapping subsets of the documents. This process is referred to as document distribution, with the documents randomly placed into the partitions for good load balancing.

Each partition runs the query in parallel, after which the top-k results from each are merged to produce the final query result. The partitions do not need to communicate with each other to process the query, and the amount of communication needed to broadcast the query and merge the results is small, so the speedup from document distribution is nearly linear in the number of partitions. Other techniques, such as term distribution, cannot handle the scaling and latency requirements of most systems [5].

We propose that document distribution be done based on document size, rather than at random. This distribution causes skew in the placement of postings within the document identifier domain, which gives benefits that reduce the overall index size and improve the query throughput. While we expect that good load balancing can be attained by adjusting the range of document sizes assigned to each partition, we leave such exploration for future work.

We demonstrate space and throughput improvements using an in-memory list intersection system (i.e., queries are a conjunction of the postings lists) without ranking. In such a system, partitioning by document size yields a smaller index when using simple16 delta compression [25], as well as faster query results when skips are added to the lists or when large lists are stored as bitvectors [7].

Similar immediate improvements will be produced in ranking based systems, with additional improvements available with more tuning of the system. In addition, executing on a subset of the partitions to decide on subsequent execution can be effective, because of the differences among partitions.

The remainder of the paper describes related work in Section 2, experimental setup in Section 3, document size distribution in Section 4, resulting benefits in Section 5, and conclusions in Section 6.

2. RELATED WORK

In this section, we present related work on distributed search systems, methods of list intersection, and the effects of document reordering on performance.

2.1 Distributed Search Systems

Distribution of search engine query processing across many machines is typically done by spreading a dataset's documents across multiple partitions (or shards). Documents are typically distributed into partitions randomly to improve

LSDS-IR '14 New York City, USA

load balancing. Queries are broadcast to the partitions, each running in parallel on a separate machine; then top-k results from each partition are merged to produce the query results. Running the partitions in parallel produces low query latency, while the independence of partitions and ease of result merging allows for near linear scaling. As a result, document distribution is used in commercial search engines.

Some versions of document distribution cluster documents by topic [12] or top-k result counts [15] and place clusters in partitions. The queries then execute on a subset of the partitions. This process is called selective search (or collection selection), and it results in improved throughput and reduced resource use. Unfortunately, topic based partitioning has the potential to degrade ranking effectiveness, which limits its use. As it turns out, our approach effectively clusters by the number of times a document appears in the top-k results, but without using collection selection and without needing a large training phase.

In extremely large datasets, the data may be split into tiers [16] based on some global order such as PageRank or on past query results. Lower tiers are queried only if higher tiers do not return sufficiently good results. Each tier is typically independent and uses document distribution, so the use of tiers is orthogonal to the work in this paper.

2.2 Intersection Approaches

Many algorithms can intersect integer lists [2, 26]. We use set-vs-set conjunctive intersection of the term postings lists from smallest to largest list. Storing the lists as uncompressed integers (32 bits/postings) is too large, but the algorithms can exploit random access into the lists, which gives fast query runtimes. Compressing the lists using deltas and variable length compression uses much less memory, but prevents random access, resulting in slow query runtimes.

For illustrative purposes, we choose the simple16 (S16) [25] compression algorithm, which produces good compression and runtime performance comparable to other approaches. The S16 encoding is word (4-byte) aligned, using 4 bits to allocate the remaining 28 bits to fit a few deltas, meaning it uses a variable block size, but we combine these into larger fixed sized blocks [25] for faster decoding. We use the publicly available S16 implementation from the Polytechnic Institute of NYU. More recent advancements in intersection algorithms and implementations involve dynamically varying block sizes [21], decoding and delta restore using vectorization [13], and processing non-delta monotone sequences [23], but for the purposes of this paper, the S16 compression algorithm is sufficient to illustrate our results.

List indexes are usually included to skip over values and thus avoid decoding, or even accessing, portions of the lists. We use a simple list index algorithm that stores every X^{th} element in an array, where X is a constant [17]. Other approaches using segments [17] or variable length skips [6] are possible, but the differences are not important here. These list index algorithms can be used with compressed lists by storing the deltas of the jump points. For simplicity, we make the compressed block size equal to the skip size X [11].

When using a compact domain of integers, as we are, the lists can instead be stored as bitvectors, where the bit number is the integer value and the bit is set if the integer is in the list. To alleviate the space costs, lists with document frequency less than F can be stored using normal compression, resulting in a hybrid bitvector algorithm [7]. This hybrid algorithm is faster than non-bitvector algorithms, and some very large lists are more compact as bitvectors.

2.3 Reordering

Search engines intersect lists of integers that represent document identifiers assigned by the system to give a compact domain of values. This assignment of identifiers can be changed to produce space and/or runtime benefits, and we refer to this process as ordering the documents.

Search engines typically store their lists as compressed deltas. Reordering can reduce deltas and thus improve compression. In addition, an ordering can also be modified to get better compression in areas other than the document identifier storage, for example, reordering to improve the compression of term frequencies embedded in lists [24].

Reordering can improve runtime performance by producing data clustering within the lists as well as query result clustering within the document identifier domain. This results in larger gaps of integers that can be skipped during query processing. These runtime improvements are seen not just in list intersection performance, but with frequency, ranking and even dynamic pruning where knowledge of the ranking algorithm is used to avoid processing some parts of the lists [22]. Tuning the compression algorithms to an ordering can also give a better space-time tradeoff [24].

Below we present various document ordering techniques:

Random: If the documents are ordered randomly (rand), there are no trends for the encoding schemes or the intersection algorithms to exploit, so this is a base of comparison for the other orderings.

Rank: Reordering to approximate ranking allows the engine to terminate early when sufficiently good results are found. A global document order [14] can be used, such as PageRank or result occurrence count in a training set [10]. Individual postings lists could be ordered independently, as done in impact ordering [1], but this increases space usage and requires accumulators to process queries.

Content Similarity: Ordering by content similarity uses some similarity metric that is applied in various ways to produce an order. Ordering using normal content clustering techniques [4] or a travelling salesman problem (TSP) [18] formulation can produce space improvements. Unfortunately, even with various improvements [3, 20], these approaches are too slow to be used in practice.

Metadata: Ordering lexicographically by URL provides similar improvements in space usage as compared to ordering by content similarity [19], and it improves runtime substantially when using skips [24]. URL ordering is essentially using the human-based organization of website authors, which often groups the documents by topic, to produce content similarity in the ordering. The effectiveness can vary greatly based on the dataset and even the density distribution of the data within the chosen domain.

Document Size: The simple method of numbering documents in decreasing order of the number of unique terms in the document has been shown to produce index compression [5] and some runtime performance improvements [22]. Ordering by terms-in-document is approximately the same as ordering by the number of tokens in the document or by document size. The improvements obtained from terms-in-document ordering are not as large as occurs with other orderings, so it has been mostly ignored.

A full ranking based search engine is likely to order by global rank or impact order, so that the query can prune portions of the intersection, while URL order is the most common approach for list intersection systems. Some recent work has tried to combine these ideas [8].

Our experiments use random ordering so as not to prejudice the improvements for a particular document ordering, but we believe that distribution by document size will improve space and runtime for any ordering.

3. EXPERIMENTAL SETUP

We use the TREC GOV2 corpus indexed by Wumpus[1] without stemming and extracted to document postings. The corpus size is 426GB from 25.5 million documents giving 9 billion document postings and 49 million terms.

Our workload is a random sample of 5000 queries chosen by Barbay et al. [2] from the 100,000 corpus queries, which we have found to produce very stable results. These queries are tokenized by Wumpus giving an average of 4.1 terms per query. Detailed query information is presented in Table 1, including averages of the *smallest* list size, the sum of *all* list sizes, and the *result* list size over all queries with the indicated number of terms for the entire corpus.

terms	queries	%	smallest	all	result
1	92	1.8	131023	131023	131023
2	741	14.8	122036	1520110	39903
3	1270	25.4	194761	6203147	31730
4	1227	24.5	199732	13213388	17879
5	803	16.1	204093	20361435	13087
6	428	8.6	192445	29367581	15004
7	206	4.1	205029	36346235	8240
8	98	2.0	206277	46198187	5726
>=9	135	2.7	198117	63406170	3308
Total	5000	100.0	186070	14944683	24699

Table 1: Query information.

Our experiments simulate a full index; we load the postings lists for query batches, encode them, flush the CPU cache by scanning a large array, then execute the conjunctive intersection of terms to produce the results. Queries have copies of their encoded postings lists, so performance is independent of query order and shared terms. Intersection runtimes per step are recorded, and overall runtimes are the sum for all steps of all queries. Space and time values ignore the dictionary, positional information and ranking.

Our code was run on an AMD Phenom II X6 1090T 3.6Ghz Processor with 6GB of memory, 6mb L3, 512k L2 and 64k L1 caches running Ubuntu Linux 2.6.32-43-server with a single thread executing the queries. The gcc compiler was used with the -O3 optimizations to produce high performance code. The query results were visually verified to be plausible and automatically verified to be consistent for all algorithms.

We used the C++ language and classes for readability, but the core algorithms use only non-virtual inline functions, allowing a large range of compiler optimizations. We encode directly into a byte array for each list, and then include decode time in our runtimes to produce more realistic and repeatable measurements. The code was tuned to minimize memory access and cache line loads.

[1]http://www.wumpus-search.org/

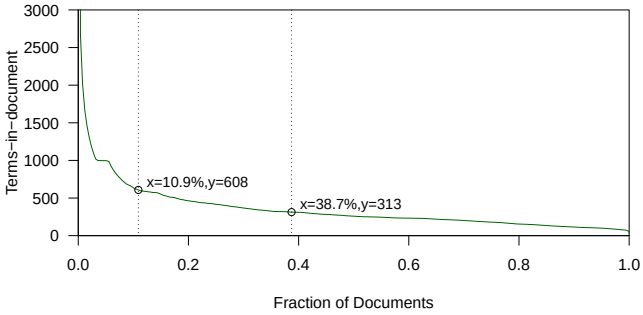

Figure 1: Terms-in-document distribution for the GOV2 dataset, cutoffs for three partitions are marked.

4. DOCUMENT SIZE DISTRIBUTION

4.1 Terms-in-Document Ordering

The size of documents can be measured in various ways, but for this paper we consider only the number of unique terms in the document. This measure is highly skewed in our GOV2 dataset, so that ordering by decreasing number of terms-in-document (td) results in more postings being packed into lower document identifiers. This skew in terms-in-document size is shown in Figure 1.

The skew in document size produces a skew in the density of postings in individual lists (i.e. front-packing of postings in lists), which produces smaller deltas and better compression. For conjunctive list intersection systems, this skew in postings makes processing in the dense front portion of the list more efficient through better locality of memory access and shared decoding work for multiple candidate documents. This locality of access will improve runtime performance for both skips and bitvectors. In addition, the less dense ends of the lists allow skips to work more effectively.

In the case of conjunctive list intersection systems, combining skewed postings lists gives *result lists* that are even more skewed. This is caused by the likelihood of a document occurring in the intersection of multiple lists increasing as the number of lists containing the document increases, which is exactly the number of terms in the document. This is similar to what would be expected from ordering by the document's usage rate in a set of training queries. For example, the document usage rate could be measured by the number of times a document occurs in the postings lists or result lists for the set of random training queries. Increased result list skew does not improve the compressibility of the data, but it does compound the benefits of locality of access and the effectiveness of skips.

In ranking systems, it is likely that more potential result documents are to be found in the partitions with large documents. This skew could be larger than the skew in the postings lists, but more investigation is need to verify this property for ranking systems.

4.2 Terms-in-Document Distribution

Ordering by terms-in-document is not the best ordering for list intersection or ranking based systems, so it is not used in practice. However, we can use the terms-in-document measure to distribute the documents into partitions, allowing for the documents to be ordered in a better way within the partitions, but still gaining some of the benefits found with terms-in-document ordering.

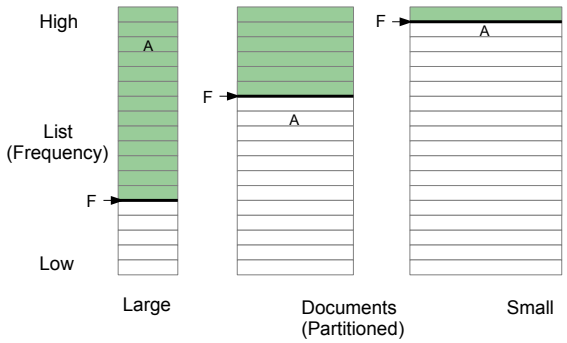

Figure 2: Schematic of bitvector distribution within index for terms-in-document partitioning (bitvectors are shaded).

An additional benefit occurs when document size distribution is used with bitvectors. This benefit is not available to terms-in-document ordering, so it was not discussed in previous work. Document distribution allows each partition to decide independently which lists to store as bitvectors, determined by the term frequency within the partition. With a random distribution, this decision is fairly consistent because the term frequencies are fairly consistent between partitions. In our document size distribution, the term frequencies are highly skewed among partitions, meaning that bitvectors can be used more efficiently.

Partitions with large documents have denser lists and use more bitvectors, while partitions with small documents have less dense lists and use fewer bitvectors. This is illustrated in Figure 2, where F is the cutoff point for using bitvectors and list 'A' demonstrates that the choice to use bitvectors is independent for each partition. As a result, more actual postings are stored as bitvectors, which improves query runtime since bitvectors are much faster than skips. In addition, having more bitvectors in partitions with large documents compounds the benefits from locality of access and increased density of results.

We implemented document size distribution to place an equal number of postings in each partition. While this does not produce good load balancing, it is sufficient to demonstrate the benefits from document size distribution. Such partitioning could be done using static ranges of document sizes, for example, splitting around 300 and 600 postings for GOV2, as shown in Figure 1.

In our experiments, we randomly order the documents within a partition to show that the improvements are not confounded by another ordering, and to convince the reader that benefits are possible for any ordering within partitions. In all cases, we found similar types of improvement when using URL ordering within partitions.

Our experiments split the GOV2 dataset into 3 partitions, even though more partitions might improve performance. We compare two approaches, the first splits GOV2 using random document distribution (rand-p-rand) and the second uses document size distribution as measured by the number of unique terms per document (td-p-rand). While the random partitions have similar document counts, the terms in document partitions have a large skew in document counts.

4.3 Validation of Skew

The skew in terms-in-document sizes for the GOV2 dataset has already been shown in Figure 1. This produces a skew in the individual postings lists used, as shown in Table 2 by the average density of the smallest lists in the workload of queries. The rand-p-rand partitions have very little skew as expected. Note: we order the values for the random partitions according to this measure and maintain this ordering in subsequent tables.

	rand-p-rand	td-p-rand
Partition 1	0.76	2.39
Partition 2	0.74	0.98
Partition 3	0.72	0.23

Table 2: Average portion of documents (%) occurring in the smallest list per query.

As expected, the conjunctive result lists for the td-p-rand partitions are even more highly skewed: the first partition contains 58.2% of the results, even though it contains a third of the postings and only 10.9% of the documents. The second partition contains 31.9% of the results, a third of the postings and 27.8% of the documents, while the third partition contains 10.0% of the results, a third of the postings and 61.3% of the documents. The rand-p-rand partitions contain no such skew, as shown in Table 3.

	rand-p-rand	td-p-rand
Partition 1	8405	13698
Partition 2	8240	8038
Partition 3	8054	2963

Table 3: Average number of results per query.

We have demonstrated that document size distribution produces skew in the postings lists of our GOV2 dataset, and additional skew in the conjunctive query result lists for our workload of queries. In the next section we demonstrate four significant benefits, namely: compressible data, locality of access, effective skips, and effective bitvectors.

5. BENEFITS OF DOCUMENT SIZE DISTRIBUTION

In this section, we compare the performance of random document distribution (rand-p-rand) and document size distribution (td-p-rand) for a conjunctive list intersection system in order to validate the benefits of document size distribution. The total resource usage limits the system throughput, and we can compute the resource usage by taking the sum of the single threaded execution times on the partitions. Load balancing also limits system throughput, but changing the partition cutoff points when using terms-in-document distribution can balance the load.

With these assumptions, our results are presented on space-time graphs, where space sums the per-partition index space and time sums the per-partition query runtimes for the configuration. We connect these space-time points to give curves over the parameters of the algorithm (X for the skips and F for bitvectors). We present the space axis as bits per posting and the time axis as ms per query.

B1: Compressible data: Document size distribution produces density skew between partitions, giving smaller and more uniform deltas and better compression. To demonstrate this, we calculate the entropy of the deltas for each partition and average them. Entropy is a measure of the

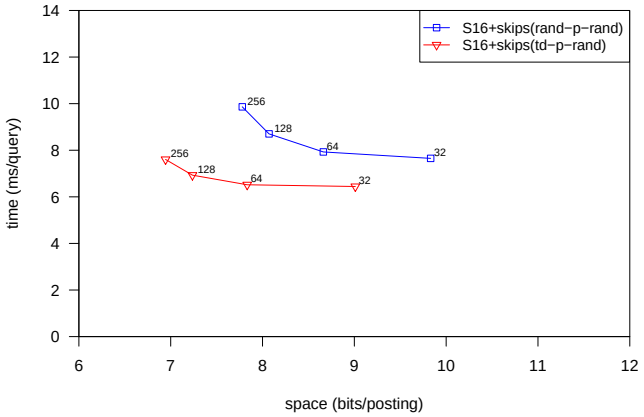

Figure 3: Space vs. time graph for the S16+skips algorithm.

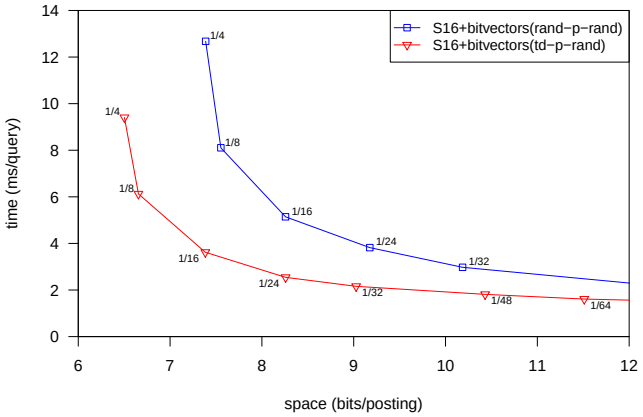

Figure 4: Space vs. time graph for the S16+bitvector algorithm.

amount of information in the data, and it gives an indication of how compressible the data might be. As shown in Table 4, td-p-rand partitioning has a lower entropy than rand-p-rand partitioning, meaning it is more compressible.

	rand-p-rand	td-p-rand
entropy	7.22	6.49

Table 4: Entropy of postings list deltas.

Actual storage using the S16 compression algorithm with block sizes of 256 is similar, as shown in Table 5.

	rand-p-rand	td-p-rand	improvement %
S16(256)	7.54	6.70	11.1

Table 5: Index space (bits/postings) using S16 compression.

These measurements are for storing the deltas, but we expect that term frequencies will also be more compressible, because the frequency values for a term occurring in a document should increase as the document gets larger, which correlates with our terms-in-document measure. This reduces the variance of frequency values within each partition, which makes them more compressible.

B2: Locality of Access: The partitions with large documents should get better locality of memory access and more blocks containing multiple values, thus sharing decoding costs. Clearly, the partition with large documents (partition 1 for td-p-rand) has a much higher density of results, since the number of results is high and the number of documents is low. This increased density of results gives more efficient processing of queries, since the average runtime per result returned is much lower than other partitions, as shown in Table 6. This efficiency suggests that the density of results has indeed given us better locality of access.

Ranking based systems will gain similar types of improvement from locality of access if they use our document size distribution technique.

B3: Effective skips: The partitions with small documents should get an increased benefit from skips. Indeed, the partition of small documents (partition 3 in td-p-rand) has the fewest results relative to the total list sizes for our query workload. As a result, skips are extremely useful in this partition. To directly demonstrate this, we calculate the amount of runtime per element in the non-smallest lists (i.e., the lists where skips can be used). As shown in Table 7, the partition containing only small documents uses the least runtime per skippable element and is thus more dependent on the speed of skipping.

	rand-p-rand	td-p-rand
Partition 1	320	205
Partition 2	319	298
Partition 3	324	446

Table 6: Runtime (ns) per result for S16+skips(64).

	rand-p-rand	td-p-rand
Partition 1	0.55	0.99
Partition 2	0.54	0.44
Partition 3	0.53	0.21

Table 7: Runtime (ns) per element in all but the smallest list of the query for S16+skips(64).

Many types of ranking based search systems use skips, so similar types of improvement are likely for such systems.

When the first three benefits are combined for the S16+skips algorithm, there is a significant improvement in both space and runtime, as shown in Figure 3.

B4: Effective bitvectors: The independence of the placement of bitvectors within each partition results in significant runtime improvements for td-p-rand partitioning, on top of the space improvements for the S16 compression algorithm, as shown in Figure 4 for the S16+bitvectors algorithm.

The use of bitvectors precludes optimizing for ranking in those postings lists that are stored as bitvectors, but they are dense and may not contribute much to improve ranking.

6. CONCLUSIONS

We have shown how skewed partitioning from document size distribution can be used to improve list intersection systems, allowing more compression and faster query throughput. Compression comes from lowering the size and variance of deltas in postings lists. Throughput improvements using skips are from better locality of access and sparse result regions. Throughput improvements using bitvectors are from better locality of access and more effective adaption of bitvector to varying list densities within partitions. We

demonstrated these benefits using an in-memory conjunctive list intersection system with random ordering, but similar types of improvement were seen with URL ordering (td-p-url) and are expected for ranking based systems.

Additional performance improvements are possible with ranking based systems. For example, executing on a subset of the partitions to decide on subsequent execution, either by distributing top-k ranking information to improve pruning/early termination, or by choosing query execution types such as AND or WAND (Weak AND).

Document size distribution is broadly applicable, being usable with any document ordering. Load balancing can be achieved by changing the partition cutoff points, or increasing replication of slow partitions. Scaling to a large number of nodes [9] may require a hierarchy of distribution approaches, such as distributing first by domain (e.g., .gov), and then by document size within each domain.

Overall, the benefits suggest that document size distribution warrants closer examination by the research community.

Acknowledgements. This research was supported by the University of Waterloo and by the Natural Sciences and Engineering Research Council of Canada. We thank the researchers at WestLab, Polytechnic Institute of NYU for providing their block based compression code [25].

7. REFERENCES

[1] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proc. of the 28th ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 226–233, 2005.

[2] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *Journal of Experimental Algorithmics (JEA)*, 14, 2009.

[3] R. Blanco and A. Barreiro. TSP and cluster-based solutions to the reassignment of document identifiers. *Information Retrieval*, 9(4):499–517, 2006.

[4] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. of the Data Compression Conference (DCC)*, pages 342–351, 2002.

[5] S. Büttcher, C. Clarke, and G. V. Cormack. *Information retrieval: Implementing and evaluating search engines*. The MIT Press, 2010.

[6] F. Chierichetti, S. Lattanzi, F. Mari, and A. Panconesi. On placing skips optimally in expectation. In *Proc. of the 1st Int. Conf. on Web Search and Data Mining (WSDM)*, pages 15–24, 2008.

[7] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)*, 29(1), 2010.

[8] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. of the 34th ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 993–1002, 2011.

[9] M. Feldman, R. Lempel, O. Somekh, and K. Vornovitsky. On the impact of random index-partitioning on index compression. *CoRR*, abs/1107.5661, 2011.

[10] S. Garcia, H. E. Williams, and A. Cannane. Access-ordered indexes. In *Proc. of the 27th Australasian Conference on Computer Science*, pages 7–14, 2004.

[11] S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *Advances in Information Retrieval*, pages 530–542. 2011.

[12] A. Kulkarni and J. Callan. Topic-based index partitions for efficient and effective selective search. In *the 8th Workshop on Large-Scale Distributed Information Retrieval (LSDS-IR)*, 2010.

[13] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 2013.

[14] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proc. of the 29th Int. Conf. on Very Large Data Bases (VLDB)*, pages 129–140, 2003.

[15] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In *Proc. of the 1st Int. Conf. on Scalable Information Systems*, page 34, 2006.

[16] K. M. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for Web search engines. In *LA-WEB*, volume 3, page 132, 2003.

[17] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. of the 9th Algorithm Engineering and Experiments (ALENEX)*, 2007.

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

[19] F. Silvestri. Sorting out the document identifier assignment problem. *Advances in Information Retrieval*, pages 101–112, 2007.

[20] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proc. of the 27th ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 305–312, 2004.

[21] F. Silvestri and R. Venturini. VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proc. of the 19th ACM Int. Conf. on Information and Knowledge Management (CIKM)*, pages 1219–1228, 2010.

[22] N. Tonellotto, C. Macdonald, and I. Ounis. Effect of different docid orderings on dynamic pruning retrieval strategies. In *Proc. of the 34th ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, pages 1179–1180, 2011.

[23] S. Vigna. Quasi-succinct indices. In *Proc. of the 6th Int. Conf. on Web Search and Data Mining (WSDM)*, pages 83–92, 2013.

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

[25] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. of the 17th Int. Conf. on World Wide Web (WWW)*, pages 387–396, 2008.

[26] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):6, 2006.