Faster Index Reordering with Bipartite Graph Partitioning

Joel Mackenzie
The University of Melbourne
Melbourne, Australia

Matthias Petri
Amazon Alexa
Manhattan Beach, CA, USA

Alistair Moffat
The University of Melbourne
Melbourne, Australia

ABSTRACT

We revisit the Bipartite Graph Partitioning approach to document reordering (Dhulipala et al., KDD 2016), and consider a range of algorithmic and heuristic refinements that lead to faster computation of index-minimizing document orderings. Our final implementation executes approximately four times faster than the reference implementation we commence with, and obtains the same, or slightly better, compression effectiveness on three large text collections.

CCS CONCEPTS

• **Information systems → Search engine architectures and scalability**; **Search index compression**.

KEYWORDS

Inverted index; document reordering; document clustering

ACM Reference Format:
Joel Mackenzie, Matthias Petri, and Alistair Moffat. 2021. Faster Index Reordering with Bipartite Graph Partitioning. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '21), July 11–15, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3404835.3462991

1 INTRODUCTION

Minimizing the resources required to carry out any given information retrieval (IR) task is a challenge that occupied researchers for the last several decades. The enormous scale of commercial web-related IR activities now means that even slender computational savings generate substantial monetary and environmental payoffs.

In this short paper we revisit the cost associated with storage of *inverted indexes*, one of the key data structures that underpin most IR systems. The atomic unit of an inverted index is the *posting*, a record $\langle d_{t,i}, f_{t,i}\rangle$ that term t appears in document $d_{t,i}$ a total of $f_{t,i}$ times, and that this is the ith document in the collection in which t appears, with the N documents in the collection numbered from zero through to $N-1$. A *postings list* is then the sequence $\{\langle d_{t,i}, f_{t,i}\rangle \mid 0 \le i < f_t\}$, where f_t is the number of distinct documents in which t occurs. When represented as *gaps*, the postings list is stored as $\{\langle d_{t,i} - d_{t,i-1}, f_{t,i}\rangle \mid 0 \le i < f_t\}$, with $d_{t,-1} \equiv -1$ for all terms t. Zobel and Moffat [28] survey these various concepts.

To reduce the cost of storing the index, integer compression techniques are applied to the gaps [22, 28]. As a combinatorial lower bound, if the f_t appearances of term t are a random subset of the N documents, then the best that can be done when storing the document gaps associated with t's posting list is approximately $f_t(\log_2(N/f_t) + 1.5)$ bits. While this limit typically represents a considerable saving compared to 32-bit integers, it can be further improved upon. In particular, non-random term usage patterns arise in many document collections because of the way the collections are constructed. For example, "covid" is one of many terms that has had unprecedented use over the last year, and will be tightly clustered in date-ordered collections. A range of integer codes have been devised that automatically exploit such non-uniformity in term usage [22, 28]; as well as techniques for identifying decompositions of postings lists into parts that can be coded using localized parameters [10, 21].

To further minimize index space, researchers have also explored methods for *document reordering*, permuting the sequence of documents so as to actively facilitate clustering. Section 2 considers the *Bipartite Graph Partitioning* (BP) mechanism of Dhulipala et al. [9] in detail, describing three improvements:

- a moderation mechanism that suppresses repetitive cycles and reduces the number of iterative passes needed;
- variant swapping-cost heuristics that result in more resilient estimations and fewer swapping operations being required; and
- algorithmic changes to eliminate the sorting operations, and hence improve asymptotic efficiency.

As an example of the considerable gains that have been achieved, on the largest of the experimental collections the running time for computing the BP reordering has been reduced from 95 minutes to 26 minutes (3.7× faster), with no loss of compression effectiveness.

2 DOCUMENT REORDERING

Motivation and Background. Document reordering re-assigns the underlying document identifiers so as to minimize the cost of storing the postings lists gaps, and is applied during the offline indexing phase of a search system as one of the more costly indexing phases [18]. Furthermore, document reordering can improve query throughput [11, 13, 15, 19, 26], with newer schemes jointly optimizing for both index space consumption and query throughput [25].

Most reordering techniques *cluster* similar documents together in the identifier space, resulting in dense regions of term occurrences in the postings lists, and hence small gaps. A simple, yet effective, technique is to order the documents lexicographically by their URLs [24], as documents from the same domains are likely to be topically coherent. More advanced techniques are based on the Traveling Salesman Problem or graph partitioning, and estimate document-to-document similarity in order to achieve clustering [2, 4, 11, 23], thereby *implicitly* optimizing the index storage cost.

Dhulipala et al. [9] and Mackenzie et al. [16] give detailed coverage of document reordering techniques. A range of other investigations have also been carried out in the related areas of network and web-graph compression [1, 3, 5–8].

SIGIR '21, July 11–15, 2021, Virtual Event, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8037-9/21/07…$15.00
https://doi.org/10.1145/3404835.3462991

```
function $reorder\_collection(D, N)$:
// reorder $D[0 \ldots N-1]$ by partitioning into halves, and then
// recursing on the halves
if $N \geq minimum\_size$ then
5:    $partition\_collection(D, N)$
      $reorder\_collection(D[0 \ldots N/2-1], N/2)$
      $reorder\_collection(D[N/2 \ldots N-1], N-N/2)$
```

Figure 1: Overview of the bipartite partitioning (BP) process.

Measurement. Document reordering techniques are compared according to the size of their compressed outputs. But different compression approaches have different strengths. To measure clustering effectiveness in a way that is independent of specific compression techniques, average posting cost is calculated using *loggap*, the mean binary logarithm of all deltas across all postings lists in the index [9, 16], providing an aspirational compression target in terms of "bits per gap". In this short paper we focus on *loggap* as the measurement of interest, noting that *loggap* is correlated with the effectiveness of various compression codecs [9].

Bipartite Partitioning. In 2016 Dhulipala et al. [9] introduced a new way of viewing the document reordering problem. Their *Recursive Bipartite Graph Partitioning* (BP) approach is given in overview in Figure 1, and in more detail in Figure 2.

Bipartite partitioning proceeds as follows. First, term statistics are collected for the two halves ("left" and "right") of the current document collection, as if separate inverted indexes were to be built. Then two *bias* estimates are made for each term, computing the change in index size that would accrue – measured in negative or positive *loggap* bits – if one document (and hence one posting) was to be moved from the left half of the collection to the right half (the *l2r_bias* in Figure 2), and likewise if one posting was to be moved from right to left (the *r2l_bias*). The term biases are next accumulated on a per-document basis using *l2r_bias* for documents in the left collection and using *r2l_bias* in the right; and any document pairs that would generate a net overall saving in *loggap* if swapped are exchanged between the two sides, *explicitly* optimizing the index storage cost. The process iterates as many as L times, before recursing into the two halves [9].

Our presentation here differs from that of Dhulipala et al. in one important way: negative bias values in Figure 2 always indicate terms (and hence) documents that are "attracted" to the left half; conversely, positive biases indicate terms and documents that have more affinity with the right half of the collection. Hence, the test $D[d_\ell].bias > D[d_r].bias$ at step 14 identifies document pairs where document d_r currently in the right half has a weaker affinity for the right than does d_ℓ, and can be displaced leftward by d_ℓ. The process for selecting document pairs to be swapped is discussed below.

Analysis. For an index containing M postings over N documents the running time is $O(M \log N + N \log^2 N)$ [9, Theorem 2]. The first component in that sum covers the counting of left and right term frequencies by traversing a forward index of total size M postings (steps 5–9); and then a second pass through the forward index using the term bias values to compute the document biases (steps 10–12), with the partition sizes summing to M postings and N documents at each level of recursion, through as many as $\log_2 N$ levels of recursion. The second term is the cost of sorting the set of computed document biases to facilitate identification of document swap-pairs at step 14, spending $O(N \log N)$ time in aggregate across the partitions in each recursive level, and with up to $\log N$ levels.

```
function $partition\_collection(D, N)$:
// partition $D[0 \ldots N-1]$ into left and right halves $D_L$ and $D_R$,
// placing each document $D[d]$ into the half that it best matches
set $D_L \leftarrow D[0 \ldots N/2-1]$ and $D_R \leftarrow D[N/2 \ldots N-1]$
5: for each term $t$ appearing in any document in $D$ do
       compute $t.lfreq$ and $t.rfreq$, the occurrence
           counts of $t$ in $D_L$ and $D_R$ respectively, then from them
       compute $t.l2r\_bias$ and $t.r2l\_bias$, the "attraction"
           of $t$ towards $D_L$ and $D_R$ respectively
10: for $d \in D_L$ do
       compute $D[d].bias$ as the sum of the $t.l2r\_bias$ values of
           the terms $t$ appearing in $D[d]$
repeat steps 10–12 for $d \in D_R$, now using the $r2l\_bias$ values
for $d_\ell \in D_L$ and $d_r \in D_R$ with $D[d_\ell].bias > D[d_r].bias$ do
15:    swap $D[d_\ell]$ and $D[d_r]$
if elements swapped and iteration limit $L$ is not reached then
       iterate again from step 4
```

Figure 2: Details of the BP process. Negative term and document biases represent attraction to the left-half collection D_L; positive biases affinity with the right-half collection D_R. If swaps can be identified that yield a net gain (steps 14–15), they are carried out.

However this analysis takes as constant the iteration cap L employed at step 16. If we include it as a variable, the execution time becomes $O(LM \log N + LN \log^2 N)$. With N and M regarded as given, this characterization suggests two main possibilities for speed improvements: reducing L; and/or finding ways of bypassing the sorting steps. We consider both of these options shortly.

Estimating Bias. Dhulipala et al. [9] observe that a uniformly-random postings list of f_t document gaps over an N-element universe gives rise to a *loggap* (in bits) of approximately

$$B(f_t, N) = f_t \left(\log_2 N - \log_2(f_t + 1)\right) . \tag{1}$$

They go on to propose that the *l2r_bias* associated with each left-collection posting for a term that initially has f_ℓ occurrences among N_ℓ documents in the left collection, and f_r occurrences among N_r documents in the right collection, be computed as

$$\begin{aligned} G_{l2r}(f_\ell, N_\ell, f_r, N_r) \quad = \quad & B(f_l, N_l) - B(f_l - 1, N_l) \\ & + B(f_r, N_r) - B(f_r + 1, N_r) , \end{aligned} \tag{2}$$

where (in our formulation) negative values indicate that postings for this term have a preference to remain in the left collection, and positive values indicate that switching the posting to the right collection will be beneficial. The right-to-left bias for the same configuration is given by:

$$G_{r2l}(f_\ell, N_\ell, f_r, N_r) = -G_{l2r}(f_r, N_r, f_\ell, N_\ell) \tag{3}$$

and reflects the favored alignment of a posting currently in the right collection, with negative values again indicating "prefers left", and positive values that (staying) right would be better.

While accurate to the cost model captured by Equation 1, Equations 2 and 3 have a drawback that becomes increasingly problematic as the partitions get smaller. Suppose that (say) $N_\ell = N_r = 20$, and consider a term with (say) $f_\ell = f_r = 1$. Then $G_{l2r}()$ yields 1.17 bits, correctly suggesting that total index size will be reduced if the left-collection posting for that term can be swapped into the right collection. But at the same time, $G_{r2l}() = -1.17$ to indicate that total index size will be reduced if the right-collection posting can be manipulated into the left collection. Hence, if those two postings are the only ones that affect the two documents that contain them, the documents will be swapped, without the estimated *loggap* saving being realized. Worse, at the next iteration they will swap back to their original positions. Many other not-as-trivial situations trigger similar effects, including complex cycles of rearrangement that return to a configuration after several intervening iterations, not just two. Moreover, the trivial "$f_\ell = f_r = 1$" situation is very common at the leaves of the recursion. Wang and Suel [25] have also noted the risk of endless swapping cycles.

Reducing L (Method 1). As already noted, Dhulipala et al. [9] propose that if any document swaps were performed, the document gains should be recomputed and checked for further swaps; with a hard limit of $L = 20$ iterations (step 16 of Figure 2).

We suggest that a simulated annealing-type mechanism be employed instead. If variable *iter* records the current iteration count, then rather than swapping documents $D[d_\ell]$ and $D[d_r]$ whenever $D[d_\ell].bias > D[d_r].bias$ (step 14 in Figure 2), we swap only if $D[d_\ell].bias > D[d_r].bias + iter$, that is, if the projected advantage across these two documents is at least *iter* bits. In the first iteration $iter = 0$, and there is no change. But at each iteration thereafter *iter* becomes one larger, making it another "one bit harder" for elements to swap. This cooling process dampens the volatility of the gain scores, and stabilizes the computation. If the instances in which swapping iterations are carried out without any *loggap* gain accruing can be avoided, we can expect to observe faster execution.

Reducing L (Method 2). Equations 1–3 are simply estimators. As a second way of moderating the number of iterations carried out, we introduce two new estimators, seeking to downplay the anticipated swapping benefit in the important $f_\ell = f_t$ case:

$$G_{l2r}(f_\ell, N_\ell, f_r, N_r) = \log_2(f_r + 2) - \log_2 f_\ell - 1.44/(f_r + 1) \tag{4}$$

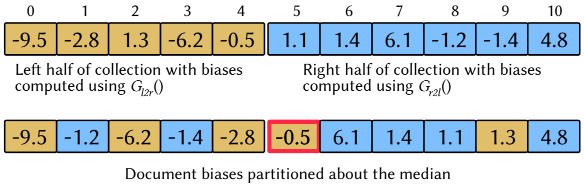
computed from Equation 1 by assuming that $N_\ell = N_r$, and using the approximation $\log_2(1 + x) \approx (\log_2 e)x \approx 1.44x$; and

$$G_{l2r}(f_\ell, N_\ell, f_r, N_r) = \log_2 f_r - \log_2 f_\ell\,, \tag{5}$$

which arises if it is assumed that the posting that transfers does not alter f_ℓ or f_r (taking $\log_2 0$ as zero). A side benefit of these two approximations is a reduced number of floating-point $\log_2()$ evaluations: Equation 2 requires (over two applications, the second embedded in Equation 3) six calls in total per term to compute the two bias values; Equation 4 requires four; and Equation 5 only two. Note also that Equation 5 is symmetric, with $G_{l2r}(f_\ell, N_\ell, f_r, N_r) = G_{r2l}(f_\ell, N_\ell, f_r, N_r)$. Table 1 provides example bias values for the original and new estimators; and Section 3 documents their effect in terms of execution time and *loggap* effectiveness.

Table 1: Values of $G_{l2r}()$ as calculated via Equation 1 (from Dhulipala et al. [9]), Equation 4, and Equation 5, with $N_\ell = N_r = 20$.

f_ℓ	f_r	Equation 2	Equation 4	Equation 5
1	0	0.00	−0.44	0.00
1	1	1.17	0.86	0.00
1	2	1.83	1.52	1.00
2	2	0.66	0.52	0.00
2	5	1.75	1.57	1.32
3	10	2.01	1.87	1.74
10	3	−1.41	−1.36	−1.74



0	1	2	3	4	5	6	7	8	9	10
-9.5	-2.8	1.3	-6.2	-0.5	1.1	1.4	6.1	-1.2	-1.4	4.8

Left half of collection with biases computed using $G_{l2r}()$
Right half of collection with biases computed using $G_{r2l}()$

-9.5	-1.2	-6.2	-1.4	-2.8	-0.5	6.1	1.4	1.1	1.3	4.8

Document biases partitioned about the median

Figure 3: Median-finding in the bias array: before the identification of the median (top); and then after the median of −0.5 has been placed in its correct location (bottom).

Sort-Free Swapping. The third change we suggest is the reason for our emphasis on "negative" biases indicating a left-leaning preference, and "positive" biases representing right-leaning tendencies. In previous implementations of BP the "pair selection process" required by step 14 in Figure 2 has followed the pseudo-code given by Dhulipala et al., which first computes all of the "*l2r*" document biases across D_L, and all of the "*r2l*" biases across D_R; with positive values in both cases meaning "this document would like to swap sides if it can". Those two sets of biases were then sorted into decreasing order, and a "zip" operation performed from largest to smallest, swapping in a pairwise manner until no net gain was possible. Those two sorting steps account for the $O(N \log N)$ time factor that is required per iteration and per recursive level.

Figure 3 illustrates a new approach that makes use of our altered interpretation of the bias values. Now all document biases have a common scale, and it is sufficient to identify the overall median, which automatically separates the documents in $D_L \cup D_R$ into $\leq$ and $\geq$ sets of the same size. Now all of the document "swaps" are a natural consequence of the median-finding process, ensuring that the left half of D contains exactly the required set of "most negative" document biases (in some order), and the right half similarly. The critical factor that makes this an attractive change is that median finding requires only $O(N)$ time, removing a factor of $O(\log N)$ from that component of the execution time.

3 EXPERIMENTS

Data and Experimental Structure. Implementations were in Rust and compiled with rustc 1.49 using the highest optimization settings. Experimentation was in-memory on a Linux machine with two 3.50 GHz Intel Xeon Gold 6144 CPUs and 512 GiB of RAM, including use of 32-thread parallelism during the document bias computations, for sorting, and for recursive calls. Parallelism was via the Rayon crate,[1] which uses a *work-stealing thread pool* to manage concurrency. Indexes were built using Anserini [27] and converted to the common index file format for our experiments [14].

Table 2 lists the three experimental text collections. The largest, CC-News-En [17], contains more than eight billion postings. Short

[1] https://github.com/rayon-rs

Table 2: Collections used and their parameters, N and M.

Collection	Documents, N	Terms	Postings, M
Wikipedia	5,550,447	12,375	514,216,782
Gov2	25,045,528	45,342	3,278,160,552
CC-News-En	43,463,480	78,851	8,777,847,467

Table 3: Effectiveness (*loggap* bits per doc-gap) for three document orderings and the baseline BP, with time in elapsed seconds.

	Wikipedia		Gov2		CC-News-En	
	loggap	Time	*loggap*	Time	*loggap*	Time
Random	5.38	–	5.75	–	3.71	–
URL	5.00	–	2.33	–	1.51	–
Length	4.29	–	2.36	–	1.59	–
BP, Eqn. 2	3.33	352	1.83	1967	1.29	5674

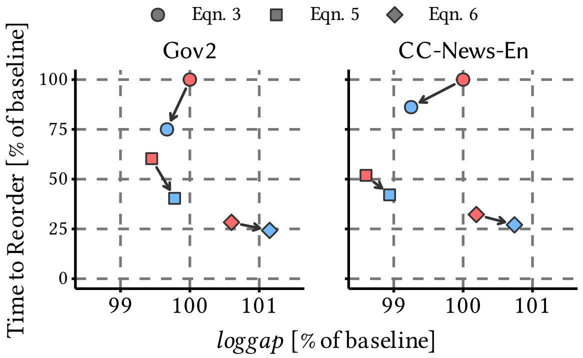


Figure 4: Efficiency vs. effectiveness without cooling (red points) and with cooling (blue), for two collections. Both axes are percentages relative to the baseline BP (last row of Table 3).

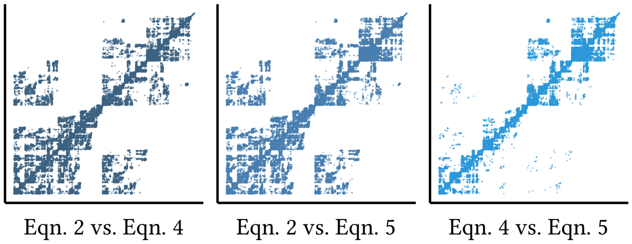
postings lists (< 4,096 elements) and long postings lists (> $0.1N$ elements) were not considered during the computation [9, 16], but this removed fewer than 1% of the postings, and all postings lists were included in the *loggap* measurements.

Efficiency vs. Effectiveness. Table 3 lists *loggap* index costs (bits per posting) for three baseline document orderings, where "Length" ordering is by decreasing document postings count. The last row of the table shows compression effectiveness and reordering computation time for the BP mechanism (Section 2), as a baseline for our experimentation, taking a URL-ordered input as the starting point in each case. Note that the input order does not alter the experimental outcomes. Our measured throughput is comparable to that of the optimized codebase of Mackenzie et al. [16].

Figure 4 uses those BP baseline measurements on two of the three collections as reference points (red circles), and shows the relative performance of the cooling technique, and of the two new bias estimators. Equation 4 (red squares) gives slightly better compression effectiveness than the Dhulipala et al. mechanism (Equation 2), and is around twice as fast. Equation 5 (red diamonds) allows even more efficient computation, but with a small amount of compression loss. Cooling (blue points) results in compression and speed gains for Equation 2; and further speed gains for Equation 4 and Equation 5, albeit with a very slight compression loss in those two cases. Similar relativities were observed for Wikipedia: for example, Equation 4 with cooling required 77 seconds (a 4.6× speedup), and resulted in no loss of effectiveness, with *loggap* = 3.32 bits per posting. Furthermore, applying binary interpolative coding [20] to the alternative indexes gave compressed sizes within 1% of the initial BP reordered index.

Table 4: Computation time (seconds, Equation 5, no cooling).

	Wikipedia	Gov2	CC-News-En
ParallelSort	92	557	1829
SequentialSort	100	637	1979
Floyd-Rivest	87	556	1780


Eqn. 2 vs. Eqn. 4

Eqn. 2 vs. Eqn. 5

Eqn. 4 vs. Eqn. 5

Figure 5: Scatterplots of document orderings from three estimators on the Wikipedia collection.

Sorting vs. Selecting. The baseline BP implementation includes two sorting calls per iteration, both implemented as parallel (that is, multi-threaded) calls. Table 4 shows the time required when that arrangement is altered: first, by using a sequential rather than parallel sort, to demonstrate the time saving attributable to the parallelism; and second, when the sort is removed completely and replaced by the Floyd-Rivest median selection algorithm [12], which runs in $O(N)$ expected time. Sorting is only a moderate fraction of the total reordering time, but even so, switching to the new median-based approach results in a 10% reduction in total computation, validating our asymptotic improvement over the baseline BP algorithm.

Overall Orderings. Figure 5 plots each document according to its computed position for each combination of estimators. The three estimators create quite different arrangements of the Wikipedia collection, with each visible square block corresponding to a recursive call that yielded a different decomposition. Equations 4 and 5 are more like each other than they are like Equation 2.

4 CONCLUSION

We have carried out a detailed exploration of Dhulipala et al.'s bipartite graph partitioning algorithm, and reduced the elapsed computation times by a factor of around four, and by even more in terms of workload (that is, summed across execution threads).

Software and Funding. The experimental software is available at https://github.com/mpetri/faster-graph-bisection. This work was supported by the Australian Research Council (project DP200103136).

REFERENCES

[1] D. Arroyuelo, M. Oyarzún, S. González, and V. Sepulveda. Hybrid compression of inverted lists for reordered document collections. *Inf. Proc. & Man.*, 54(6): 1308–1324, 2018.

[2] R. Blanco and A. Barreiro. Document identifier reassignment through dimensionality reduction. In *Proc. ECIR*, pages 375–387, 2005.

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

[4] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. DCC*, pages 342–352, 2002.

[5] P. Boldi and S. Vigna. The webgraph framework I: Compression techniques. In *Proc. WWW*, pages 595–602, 2004.

[6] P. Boldi, M. Santini, and S. Vigna. Permuting web and social graphs. *Internet Mathematics*, 6(3):257–283, 2009.

[7] C. Cheng, C. Chung, and J. J. Shann. Fast query evaluation through document identifier assignment for inverted file-based information retrieval systems. *Inf. Proc. & Man.*, 42(3):729–750, 2006.

[8] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proc. KDD*, pages 219–228, 2009.

[9] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proc. KDD*, pages 1535–1544, 2016.

[10] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. SIGIR*, pages 993–1002, 2011.

[11] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *Proc. WWW*, pages 311–320, 2010.

[12] R. W. Floyd and R. L. Rivest. Expected time bounds for selection. *Comm. ACM*, 18(3):165–172, 1975.

[13] D. Hawking and T. Jones. Reordering an index to speed query processing without loss of effectiveness. In *Proc. Aust. Doc. Comp. Symp.*, pages 17–24, 2012.

[14] J. Lin, J. Mackenzie, C. Kamphuis, C. Macdonald, A. Mallia, M. Siedlaczek, A. Trotman, and A. de Vries. Supporting interoperability between open-source search engines with the common index file format. In *Proc. SIGIR*, pages 2149–2152, 2020.

[15] J. Mackenzie and A. Moffat. Examining the additivity of top-k query processing innovations. In *Proc. CIKM*, pages 1085–1094, 2020.

[16] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel. Compressing inverted indexes with recursive graph bisection: A reproducibility study. In *Proc. ECIR*, pages 339–352, 2019.

[17] J. Mackenzie, R. Benham, M. Petri, J. R. Trippas, J. S. Culpepper, and A. Moffat. CC-News-En: A large English news corpus. In *Proc. CIKM*, pages 3077–3084, 2020.

[18] A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel. PISA: Performant indexes and search for academia. In *Proc. OSIRRC Wrkshp. at SIGIR 2019*, pages 50–56, 2019. URL http://ceur-ws.org/Vol-2409/docker08.pdf.

[19] A. Mallia, M. Siedlaczek, and T. Suel. An experimental study of index compression and DAAT query processing methods. In *Proc. ECIR*, pages 353–368, 2019.

[20] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[21] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. SIGIR*, pages 273–282, 2014.

[22] G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM Trans. Inf. Sys.*, 53(6):125.1–125.36, 2021.

[23] W. Shieh, T. Chen, J. J. Shann, and C. Chung. Inverted file compression through document identifier reassignment. *Inf. Proc. & Man.*, 39(1):117–131, 2003.

[24] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. ECIR*, pages 101–112, 2007.

[25] Q. Wang and T. Suel. Document reordering for faster intersection. *PVLDB*, 12(5): 475–487, 2019.

[26] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.

[27] P. Yang, H. Fang, and J. Lin. Anserini: Reproducible ranking baselines using lucene. *J. Data Inf. Qual.*, 10(4):1–20, 2018.

[28] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2):6:1–6:56, 2006.