Tradeoff Options for Bipartite Graph Partitioning

Joel Mackenzie – Matthias Petri – Alistair Moffat

Abstract—Web connectivity graphs and similar linked data such as inverted indexes are important components of the information access systems provided by social media and web search services. The Bipartite Graph Partitioning mechanism of Dhulipala et al. [KDD 2016] relabels the vertices of large sparse graphs, seeking to enhance compressibility and thus reduce the storage space occupied by these costly structures. Here we develop a range of algorithmic and heuristic refinements to Bipartite Graph Partitioning (BP) that lead to faster computation of space-reducing vertex orderings whilst continuing to apply the same broad algorithmic paradigm. Using a range of web graph and information retrieval system index data as test cases, we demonstrate an implementation that executes up to approximately four times faster than the baseline implementation we commenced with, while holding compressibility approximately constant. We have also improved the asymptotic execution time of BP by replacing a sorting step by a customized median-finding step.

Index Terms—Web graph, inverted index, graph clustering, document reordering, document clustering

1 Introduction

The enormous scale of commercial activities in web-related areas such as social media and internet search means that even slender savings in computational costs – expressed as processing effort and/or storage space – will generate substantial monetary and environmental payoffs. Minimizing the resources required to carry out any given information processing task is a challenge that has occupied researchers and practitioners alike for the last several decades.

In this work we revisit the cost associated with storage of *large sparse graph* structures, with an emphasis on two important application areas: *web graph* storage, where the focus is on the *adjacency lists* that record the connections between web pages within crawled web data, as expressed via hyperlinks from one page to other pages; and *inverted index* storage, where the focus is on the *postings lists* that record the occurrences of terms within crawled web pages (and documents from non-web sources), and allow documents matching a set of query terms to be efficiently found. These two storage challenges can both be thought of as large sparse graphs, albeit with different overall properties; and can be handled by similar approaches.

The atomic unit of a web graph G over N webpages (documents) numbered 0 to $N-1$ is the *edge* or *link*, a record $\langle d_s, d_j\rangle$ that indicates that document d_s (a "source") contains a hyperlink to document d_j, with $d_s, d_j \in [0 \ldots N-1]$. The set of links is usually stored as a collection of adjacency lists, with A_s the set of documents reachable from d_s, that is, $A_s = \{d_j \mid \langle d_s, d_j\rangle \in G\}$. We can also regard the set A_s as being ordered, so that, taking $f_s = |A_s|$, it is also possible to write $A_s = \{d_{s,i} \mid 0 \le i < f_s\}$, where $d_{s,i}$ is the ordinal index of the ith document to which d_s is linked. In a web graph, the number of adjacency lists will be equal to the number of documents; and if the total number of edges across all adjacency lists is denoted M, we have $M = \sum_{s=0}^{N-1} f_s$.

The atomic unit of an inverted index for a document collection D containing N documents numbered 0 to $N-1$ 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. The *postings list* for term t, denoted P_t, is then the sequence $P_t = \{\langle d_{t,i}, f_{t,i}\rangle \mid 0 \le i < f_t\}$, where f_t is the number of distinct documents in which t occurs. If the terms t are numbered from zero, $0 \le t < V$, where V is the size of the *vocabulary* of the collection, then the total number of postings is given by $M = \sum_{t=0}^{V-1} f_t$. Note that in an inverted index V might be smaller than, equal to, or greater than N, whereas in a web graph it is normal to have $V \approx N$.

With both A_s and P_t able to be stored in sorted form without any loss of generality, it is usual to store the document numbers as relative *document gaps*, or d-gaps in short. For example, a postings list P_t can be stored as $P_t = \{\langle d_{t,i} - d_{t,i-1}, f_{t,i}\rangle \mid 0 \le i < f_t\}$, with $d_{t,-1} \equiv -1$ for all terms t. Zobel and Moffat [1] and Pibiri and Venturini [2] survey these various concepts and discuss inverted index representations. Similarly, an adjacency list A_s can be stored as gaps $d_{s,i} - d_{s,i-1}$, again with $d_{s,-1} \equiv -1$ for all documents s.

To reduce the cost of storing postings or adjacency lists, integer compression techniques are applied to the d-gaps [1, 2]. As a combinatorial lower bound, if the f_t appearances of term t in documents in D are a random subset of the N documents that make up the collection (similarly, if the f_s links emanating from document s are to randomly-selected other documents amongst the N pages in web graph G),

- *J. Mackenzie was with the School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia at the time this work was undertaken. He is now with the School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, Australia. E-mail: joel.mackenzie@uq.edu.au*
- *A. Moffat is with the School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia. E-mail: ammoffat@unimelb.edu.au*
- *M. Petri is with Amazon Alexa AI, Manhattan Beach, CA, USA. E-mail: mkp@amazon.com*

This paper is an extended version of work that was presented in preliminary form as a Short Paper at the 2021 ACM SIGIR International Conference on Research and Development in Information Retrieval, see https://doi.org/10.1145/3404835.3462991.

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 two years (as has the term "unprecedented"), and will be tightly clustered in date-ordered collections. Similar effects hold in adjacency lists: documents in one web site are likely to link to other documents within that same web site via navigational assists and commonality of topics, and to pages at other similar-topic websites because of thematic connections. The outgoing links from any given web page are far from random.

A range of integer codes have been devised that automatically exploit such non-uniformity in target appearance [1, 2]; as well as techniques for identifying decompositions of postings (and hence, adjacency) lists into parts that can be coded using localized parameters [3, 4].

To further minimize index space, researchers have also explored methods for *document reordering*, permuting the sequence of documents so as to actively facilitate non-uniformity of d-gaps when considered in aggregate, across all postings or adjacency lists. These techniques can be thought of as performing a document clustering process that collects together like documents – where "like" is defined with respect to term usage for inverted indexes, and with respect to out-link commonality for web graphs – and then linearizes the clusters into a single sequential renumbering.

Our work in this paper takes the *Bipartite Graph Partitioning* (BP) mechanism of Dhulipala et al. [5] as a starting point (described in detail in Section 2), and uses it as a foundation on which to present four 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;
- algorithmic changes to eliminate the sorting operations, and hence improve asymptotic efficiency;
- a level-synchronized iterative version that reduces inter-task contention and achieves better throughput rates as a result.

We also provide an analysis of the parallelism that is possible in a BP implementation.

As an example of the considerable gains that have been achieved, on the largest of the experimental document collections the running time for computing the BP reordering for an inverted index is reduced from 95 minutes to 26 minutes (3.7× faster), with no loss of compression effectiveness. Full results appear in Sections 4 and 5.

Compared to our previous conference presentation of this work [6], we add a detailed exploration of issues to do with parallelism, bottle-neck analysis, and other implementation options that affect practical use; we include additional experimentation to show the strong relationship between *loggap* and compressed index size; we give a more detailed treatment of both the background material and of the new mechanisms that we describe; and our experimentation has been expanded to include additional datasets drawn from the more general web graph domain.

2 Document Reordering

This section introduces the bipartite partitioning technique [5] for reducing storage costs associated with adjacency lists and inverted lists. For simplicity of exposition we refer in the main to postings lists, but as was noted in Section 1, there are strong parallels between postings lists and adjacency lists, and any commentary in regard to one can be assumed to also apply to the other.

2.1 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 based on the observation that most compression codecs are more effective over dense regions of small gaps [7, 4, 8] than they are when handling the same number of approximately equal-sized gaps. Reordering is applied during the offline indexing phase of a search system as one of the more costly indexing phases [9]. Furthermore, document reordering can improve query throughput [10, 11, 12, 13, 14, 15], with newer schemes jointly optimizing for both index space consumption and query throughput [16].

Reordering techniques seek to cluster like documents together in the identifier space, to create – in aggregate, across the whole index – dense regions of term occurrences in the postings lists, and hence small gaps. One simple technique is to order the documents lexicographically by their URLs [17], on the basis of documents from the same domains likely to be topically coherent. More advanced techniques estimate pairwise document-to-document similarities, and then apply heuristic cost-based traversals in order to achieve clustering [18, 19, 20, 11]. Web graphs can also be reordered by URL or other unique document descriptor, provided the selected identifier relates to the origins of each document.

Dhulipala et al. [5] and Mackenzie et al. [21] give detailed coverage of document reordering techniques in the context of web graphs and inverted indexing, respectively. A range of other investigations should also be noted [22, 23, 24, 25, 26, 27, 28].

2.2 Measurement

Experimentally, document reordering techniques can be compared directly, 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, the average cost of storing a single document gap describing a single edge in an adjacency list, or a single posting in a postings list, can be computed using a *loggap* estimation, the mean binary logarithm of all gaps across all of the documents in all of the lists [5, 21]:

$$\mathit{loggap}(D) = \frac{1}{M} \cdot \left(\sum_{0 \le t < V} \sum_{0 \le i < f_t} \log_2(d_{t,i} - d_{t,i-1}) \right)$$

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

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

where D is the collection of documents, V is the number of distinct terms, and where $loggap(D)$ provides an aspirational compression target in units of "ideal bits per gap". We make use of *loggap* as an estimator of cost for both adjacency and postings lists, after first demonstrating in Section 3 that *loggap* is indeed correlated with the effectiveness of various compression codecs (see also Dhulipala et al. [5]). Where appropriate we also report exact compressed sizes in Gibibytes.

2.3 Bipartite Partitioning

In 2016 Dhulipala et al. [5] introduced a new way of viewing the document reordering problem. Their *Recursive Bipartite Graph Partitioning* (BP) approach is introduced in Figure 1, which presents the overall recursive structure. As can be seen, the algorithmic pattern employed is a hybrid between Quicksort and Mergesort, in that the local work in each recursive call is performed prior to the two recursive calls, as is the case with Quicksort, but the recursive calls operate on strict halves, as is the case with Mergesort.

Figure 2 then presents the details of the "local work" component, which partitions D into two halves, and then identifies document pairs that can be usefully swapped between them. There are three distinct phases shown in Figure 2. First, term statistics are collected for the two halves ("left", D_L; and "right", D_R) of the current document collection D, as if separate inverted indexes were being built, see steps 4–8 in Figure 2. Then two *bias* values are computed for each term (steps 9–10), estimating the change in index size that would accrue – measured in negative or positive *loggap* bits – if one posting for that term was to be moved from the left half of the collection to the right half (the *l2r_bias*), and symmetrically, if one posting was to be moved from D_R to D_L (the *r2l_bias*).

In our presentation those bias values have polarity indicating the "preference" exerted by each term. A negative $t.l2r\_bias$ indicates that instances of t in documents already in D_L prefer to stay where they are, if possible; whereas negative $t.r2l\_bias$ values indicate that instances of t in documents in D_R will try and "pull" their documents to the left. Similarly, positive bias values exert rightward pressure: a $t.l2r\_bias$ that is greater than zero indicates that left-side documents containing t will come under pressure to shift right into D_R; and $t.r2l\_bias > 0$ indicates that documents in D_R containing t will possess inertia, and be resistant to changing sides and switching to the left.

In the second phase the term biases are accumulated on a per-document basis (steps 12–14) using *l2r_bias* for documents in the left collection and using *r2l_bias* in the right. Each document bias value is also signed, with polarity indicating where – in aggregate – the terms contained in that document would like to push or pull the document. Negative document biases indicate documents that should either stay in D_L or be moved left into D_L; positive document biases indicate documents that should either remain in D_R, or be moved right to D_R.

```
1:  function $partition\_collection(D, N)$:
2:  // first, partition $D[0 \ldots N-1]$ into left half $D_L$ and
3:  // right half $D_R$ and compute local term frequencies
4:  set $D_L \leftarrow D[0 \ldots N/2-1]$ and $D_R \leftarrow D[N/2 \ldots N-1]$
5:  set $iter \leftarrow 0$ and $swaps \leftarrow$ false
6:  for each term $t$ appearing in any document in $D$ do
7:      compute $t.lfreq$ and $t.rfreq$, the occurrence
8:          counts of $t$ in $D_R$ and $D_L$ respectively
9:      use $t.lfreq$ and $t.rfreq$ to compute $t.l2r\_bias$ and
10:         $t.r2l\_bias$, the "attraction" of $t$ to $D_L$ and $D_R$
11: // second, compute document biases from term biases
12: for $d \in D_L$ do
13:     set $D[d].bias \leftarrow \sum\{t.l2r\_bias \mid t \in D[d]\}$
14: repeat steps 12–13 for $d \in D_R$ and using $t.r2l\_bias$
15: // third, swap pairs of documents between $D_L$ and $D_R$
16: // when a net gain in bias can be achieved
17: for suitable candidate pairs $d_\ell \in D_L$ and $d_r \in D_R$ do
18:     if $D[d_\ell].bias > D[d_r].bias$ then
19:         exchange $D[d_\ell]$ and $D[d_r]$
20:         set $swaps \leftarrow$ true
21: if $swaps$ and iteration limit $iter < L$ then
22:     set $iter \leftarrow iter + 1$ and $swaps \leftarrow$ false
23:     repeat again from step 6
```

Fig. 2: Details of the BP process. Negative term and document biases represent attraction to the left-half collection D_L; positive term and document biases indicate affinity with the right-half collection D_R. If swaps can be identified that yield a net gain (steps 17–20), they are carried out. The way in which document pairs are selected at steps 17–18, and the role of the iteration limit L at step 21 are discussed in the text.

Finally, in the third phase, any document pairs that generate a net overall saving in *loggap* are exchanged between the two sides (steps 17–20), seeking to explicitly optimize the index storage cost. The signs on the document biases, and the test at step 18, ensure that all such document exchanges do indeed reduce the estimated summed cost of the indexes for D_L and D_R, and hence reduce the estimated cost for the combined index D that is generated when the D_L and D_R indexes are concatenated.

Exchanging documents alters the term frequencies and hence erodes the quality of the bias estimations. To update the biases and retain estimation precision, the whole process is permitted to iterate as many as L times (step 21) before recursing (see Figure 1) into the two halves. The role of L is discussed in more detail below.

2.4 Signed Bias Values

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 always indicate terms and documents that have greater affinity with the right half of the collection. Hence, the test $D[d_\ell].bias > D[d_r].bias$ at step 18 only allows 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 with a net bit saving given by $D[d_\ell].bias - D[d_r].bias$. Note that this relationship holds regardless of the actual signs of $D[d_\ell].bias$ and $D[d_r].bias$.

The reason for this change in representation, and the process for selecting document pairs to be swapped, are discussed in more detail below.

2.5 Complexity Analysis

For an index (or web graph) containing M postings (or edges) over N documents the running time is $\mathcal{O}(M \log N + N \log^2 N)$ [5, Theorem 2], provided that L is taken to be a constant. Alternatively, if L is also regarded as being a parameter of the algorithm, then the running time should be expressed as $\mathcal{O}(LM \log N + LN \log^2 N)$. For brevity, we continue here with the former representation, and for the most part omit L when presenting analyses.

The $\mathcal{O}(M \log N)$ first component covers the $\mathcal{O}(M)$ cost in Figure 2 of counting the left and right term frequencies by traversing a "forward" representation of the input (steps 6–8). The second pass through the forward index to compute the document biases (steps 12–14) is also $\mathcal{O}(M)$ at each recursive call. But in both cases M is the local value, specific to that particular call to function *partition_collection*(). When summed across all of the recursive calls that co-exist at each level of recursion the total cost is $\mathcal{O}(M)$ still, with M now the global value, a similar argument as applies in the analysis of Mergesort. Moreover, there are exactly $\log_2 N$ levels of recursion, also as is the case with Mergesort.

The $O(N \log^2 N)$ second term in the analysis of Dhulipala et al. is the cost of sorting the sets of computed document biases as a prelude to step 17. Assuming that the sets $D[d_\ell].bias$ and $D[d_\ell].bias$ must be fully sorted, $\mathcal{O}(N \log N)$ time is required in each recursive call, where N is the local value. Across each of the recursive levels, that cost sums to $\mathcal{O}(N \log N)$ time, where N is now the global value. Since there are $\log N$ recursive levels, the total sorting time is $\mathcal{O}(N \log^2 N)$.

The relationship between M and N plays a key role in this analysis. In an inverted index M/N is the average number of distinct indexed terms per document, with 100 at the lower end of the typical range, and 1000 being toward the upper end. On the other hand, in web graphs, M/N is the average number of hyperlinks per page, with $M/N = 10$ perhaps typical, and $M/N = 100$ relatively rare. The range of values expected is important, because the relativity between M/N and $\log N$ affects which of the two terms $\mathcal{O}(M \log N)$ and $\mathcal{O}(N \log^2 N)$ is asymptotically dominant. As was noted earlier, this analysis treats the iteration cap L (step 21 in Figure 2) as a constant. If instead L is viewed as a variable, the two components of the execution time are $\mathcal{O}(LM \log N)$ and $\mathcal{O}(LN \log^2 N)$.

3 Experimental Setup

This section describes the hardware and data resources employed in our experimentation. It also reports preliminary experimentation in which we confirm the relationship between the bit rates achieved by integer compression codecs and the approximations provided by the *loggap* estimator.

3.1 Hardware and Software

Our experimental software was implemented in Rust and compiled with rustc 1.49 using a high level of optimization settings. All experiments were conducted in-memory on a Linux machine with two 3.50 GHz Intel Xeon Gold 6144 CPUs and 512 GiB of RAM. Parallel processing was managed via the Rayon crate,[1] which uses a *work-stealing thread pool* to manage concurrency. A total of 32-threads were employed during the document bias computations, for sorting, and for recursive calls. Issues to do with parallelism are considered in detail in Section 5.

3.2 Datasets

The experiments make use of four publicly available graphs, all of which are converted to unweighted bipartite graphs. We then employ these unweighted bipartite graphs as the basis for all experimentation and measurement:

- Enron: An email communication network from the now defunct Enron company.[2]
- Facebook: A social network of user-to-user links from the New Orleans Facebook network circa 2009 [29].
- Google: A hyperlink network from Google. The data was released in 2002.[3]
- LiveJournal: A social network of user-to-user links from the LiveJournal site in 2006 [30].[4]

Note that our normalization process yields different statistics to those reported by Dhulipala et al. [5, Table 1], and means that numeric values should not be compared across the two papers. In terms of index data, three publicly available text collections were processed to extract inverted indexes using Anserini [31]:

- Wikipedia: A snapshot of English Wikipedia articles from mid 2018.
- Gov2: A crawl of `.gov` domains from 2004.
- CC-News-En: A snapshot of English news documents from the Common Crawl between 2016 and 2018 [32].

As was also the case with other work on reordering of inverted indexes, short postings lists (fewer than 4096 elements) and long postings lists (more than $0.1N$ elements) were not taken into consideration when processing the four text collections [5, 21]. However all postings lists were included in the subsequent *loggap* and index size measurements. After that pre-processing, both graphs and text collections were represented using the common index file format [33]. Table 1 lists various parameters for the seven experimental datasets, including the derived value M/N, which for the web graphs is (with one exception) $\leq \log_2 N$, and for the indexes (even after suppression of very rare and very common terms) is $> \log_2 N$.

[1] https://github.com/rayon-rs
[2] https://snap.stanford.edu/data/email-Enron.html
[3] https://snap.stanford.edu/data/web-Google.html
[4] https://snap.stanford.edu/data/soc-LiveJournal1.html

TABLE 1: Datasets and their parameters as used in the BP computation, that is, after normalization. For web graphs, "nodes" are webpages, the "vocab" is the set of hyperlink targets, and "edges" are hyperlinks. For inverted indexes, "nodes" are documents, the "vocab" is the set of terms, and "edges" are postings. Density is computed as M/N, the mean number of out-edges per page for web graphs, and the mean number of postings per document for indexes.

Collection	Nodes, N	Vocab, V	Edges, M	Density, M/N
Enron	3.67×10^4	3.67×10^4	3.68×10^5	10.0
Facebook	6.37×10^4	6.37×10^4	1.63×10^6	25.6
Google	7.39×10^5	7.15×10^5	5.11×10^6	6.9
LiveJournal	4.31×10^6	4.49×10^6	6.90×10^7	16.0
Wikipedia	5.55×10^6	1.24×10^4	5.14×10^8	92.6
Gov2	2.50×10^7	4.53×10^4	3.28×10^9	130.9
CC-News-En	4.35×10^7	7.89×10^4	8.78×10^9	202.0

3.3 Calibrating the Measurements

Our first experiment documents the high correlation between *loggap* and true compression rates, to confirm the relationship also recorded by Dhulipala et al. [5]. The four panes in Figure 3 show the application of four different compression codecs to several different versions of each of the seven test files, which, as already noted, cover four web graphs and three inverted indexes. The four methods are the Binary Interpolative Encoding approach of Moffat and Stuiver [7]; the Partitioned Elias-Fano method of Ottaviano and Venturini [4]; the SIMD-BP128 code devised by Lemire and Boytsov [34]; and the StreamVByte approach of Lemire et al. [35]. Each makes different tradeoffs between compression effectiveness and operational throughput, and collectively they represent the current state of the art [2].

The pattern within each pane is clear: regardless of the dataset, and regardless of the ordering into which it is permuted, there is a strong correlation between the *loggap* measurement and the compression effectiveness achieved by that codec. Note also that the *loggap* measurement is always an under-estimate of the actual compressed size, and that the difference between trend effectiveness (the shaded confidence intervals) and the red lines indicated by the aspirational *loggap* measurements indicate how close that codec comes to being "impossibly perfect".

In a side experiment, we also computed the cost of storing the graphs with the Webgraph framework [22] (also known as the *BV* representation); and, as did Dhulipala et al. [5, Figure 8] before us, found it to be highly correlated with *loggap*. We thus report only *loggap* here, concluding that it can indeed be used as a codec-independent surrogate measurement for compressed size.

3.4 Baseline Systems

Our primary goal in this work is to focus on variants of the BP mechanism described by Dhulipala et al. [5], noting that Dhulipala et al. (in their Table 2) compared against a cross-section of other earlier graph reordering mechanisms, and found that BP performed consistently well. However, as simple reference points, we also include results for randomly ordered collections and URL-ordered collections, to highlight the additional gains that BP is capable of achieving.

4 Faster Estimator Heuristics

This section describes the estimators used for *t.l2r_bias* and *t.r2l_bias* that are assumed in Figure 2, which are computed from *t.lfreq* and *t.rfreq*, the frequencies of term t in $D.L$ and $D.R$ respectively. We start by describing the method proposed by Dhulipala et al. [5], and then introduce two approximations to those computations, along with our rationale for considering them.

4.1 Estimating Bias

Dhulipala et al. [5] observe that a uniformly-random postings list of f_t document gaps over an N-element universe contributes approximately

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

bits towards the numerator of the corresponding *loggap* computation. As a consequence of that definition, Dhulipala et al. go on to propose that the net bit gain $G_{l2r}(\cdot)$ associated with one left-collection posting for a term t that initially has $f_{t,\ell}$ occurrences among N_ℓ documents in the left collection and has $f_{t,r}$ occurrences among N_r documents in the right collection, be computed by taking the bit difference between the left and right "before" configurations, and the left and right "after" configurations:

$$\begin{aligned} G_{l2r}(f_{t,\ell}, N_\ell, f_{t,r}, N_r) = \\ B(f_{t,\ell}, N_l) - B(f_{t,\ell} - 1, N_l) \\ + B(f_{t,r}, N_r) - B(f_{t,r} + 1, N_r) . \end{aligned} \qquad (2)$$

The reader is reminded that in our formulation (see Section 2.4) 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. A symmetrical computation applies if a single posting originally in the right collection D_R were to be (hypothetically) swapped left into D_L. In combination, the two required definitions then emerge:

$$\begin{aligned} t.l2r\_bias &= G_{l2r}(f_{t,\ell}, N_\ell, f_{t,r}, N_r) \\ t.r2l\_bias &= -\,G_{l2r}(f_{t,r}, N_r, f_{t,\ell}, N_\ell) . \end{aligned} \qquad (3)$$

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 t with (say) $f_{t,\ell} = f_{t,r} = 1$. Then $G_{l2r}()$ yields a *t.l2r_bias* of $+1.17$ bits, correctly suggesting that total index size will be reduced if the left-collection posting for t can be swapped into the right collection, to join the posting already in D_R. But at the same time, *t.r2l_bias* is computed as being -1.17 bits, to similarly indicate that total index size will be reduced if the right-collection posting for t can be transferred into the left collection D_L. Hence, all other components being equal (or, more precisely, being within 1.17 bits of being equal), those two documents will be swapped, without the estimated 3.34 bits of *loggap* saving being realized. Worse, at the next iteration the same two documents will swap back to their original positions, and will continue to flip-flop until the iteration limit L is reached (step 21 in Figure 2).

Many other scenarios trigger similar repetitions, including complex cycles of rearrangement that return to a previous configuration after several intervening iterations, not

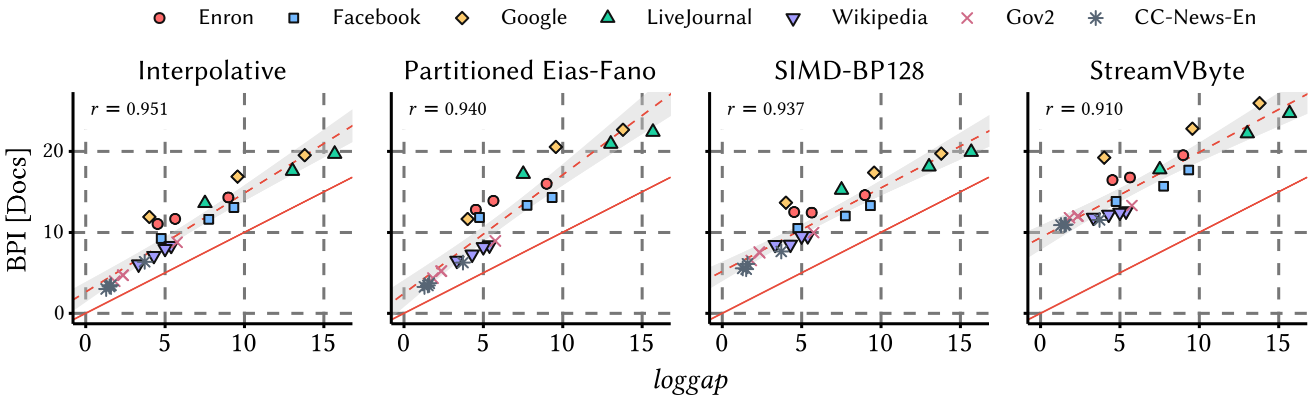


Fig. 3: Compression effectiveness (in bits per document identifier) plotted against *loggap* for four distinct codecs, seven collections, and a range of possible orderings (Random, URL (where applicable), Length, and BP). Each pane has a fitted linear model (shown as a dashed line with shaded 95% confidence intervals) as well as the $y = x$ line for reference. The value at the top-left of each pane reports the value of Pearson's r, shown as a dashed line

just two; nor is the problem restricted to the lower levels of the recursive hierarchy, it can and does happen at any level. Furthermore, tabulating all prior configurations and checking for re-occurrences would add significant computational overhead. Wang and Suel [16] have also noted the risk of endless swapping cycles. One of our goals in this investigation was to reduce the opportunity for such redundant computations to occur.

4.2 Avoiding Iterations: Cooling

As noted in Figure 2, Dhulipala et al. [5] propose that if any document swaps are performed, the document gains should be recomputed and checked for further swaps; with a hard limit of $L = 20$ iterations (step 21 of Figure 2). But if iterations in which little or no *loggap* reduction can be avoided, faster overall execution should be possible.

We thus suggest that a simulated annealing-type mechanism be employed. With variable *iter* recording the current iteration count, then rather than swap documents $D[d_\ell]$ and $D[d_r]$ whenever $D[d_\ell].bias > D[d_r].bias$ (step 18 in Figure 2), we propose swapping only if $D[d_\ell].bias > D[d_r].bias + C(iter)$ that is, if the projected advantage across these two documents is at least $C(iter)$ bits, where the cooling function $C(\cdot)$ satisfies $C(0) = 0$ and is monotonically increasing thereafter. In the first iteration $iter = 0$, and there is no change to the existing computation. But at each iteration thereafter *iter* becomes larger, making it harder for documents to swap. This *cooling* process can be expected to dampen the volatility of the gain scores, and thus stabilize the computation.

In our experiments we make use of the simplest possible cooling function, and take $C(iter) = iter$, making each iteration's swaps jump a hurdle that is one bit higher than was required in the previous iteration.

4.3 Avoiding Iterations: Alternative Estimators

As a second way of moderating the number of iterations carried out, we introduce two further estimators, seeking to downplay the anticipated benefit in the important $f_{t,\ell} \approx f_{t,r}$ case, and thus suppress some of the unnecessary swaps.

Equations 1 and 2 provide a general estimator that makes very few assumptions. But in the BP process described by Dhulipala et al. the binary recursive structure means that the left-half collection and the right-half collection will always differ in size by at most one, that is, $N_\ell \approx N_r$. The first of the two new estimators is derived from Equation 2 by assuming that $N_\ell = N_r$ (which means that many of the terms in the previous expression cancel, making it simpler to compute), and then applying the approximation $\log_2(1 + x) \approx (\log_2 e)x \approx 1.44x$ (which allows further algebraic simplification, and avoids calls to evaluate logarithms). Taken together these lead to the formulation:

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

The second new estimator then arises if it is further assumed that the posting that transfers from D_L to D_R is moved at its face-value cost prior to the move, that is, that the "before" values $f_{t,\ell}$ or $f_{t,r}$ remain the estimators that determine the bit-cost of the moved posting, even though they will in fact be altered by the move being carried out:

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

in which $\log_2 0$ is taken to be zero. A useful side benefit of these two approximations is a reduced number of floating-point $\log_2(\cdot)$ evaluations: computing the two values described by Equation 3 when $G_{l2r}(\cdot)$ is instantiated via Equation 2 involves six logarithms per term t; instantiation via Equation 4 requires four; and instantiation via Equation 5 needs only two $\log_2(\cdot)$ evaluations.[5]

Equation 5 is the only estimator of the three that is symmetric, with $t.l2r\_bias = -t.r2l\_bias$. Table 2 provides example values of $t.l2r\_bias$ and $t.r2l\_bias$ for the original and the two new estimators, fixing $N_\ell = N_r = 20$, and considering nine different combinations of $f_{t,\ell}$ and $f_{t,r}$.

[5] Or a corresponding number of lookups if the logarithms are memoized into a pre-computed table. In all of our BP implementations values of $\log_2 x$ for $x < 4096$ are pre-computed, saving approximately 20% of the running time for the baseline BP version, and lesser fractions for the other two estimators.

TABLE 2: Values of $G_{l2r}()$ and hence *t.l2r_bias* as calculated via Equation 2 (from Dhulipala et al. [5]), Equation 4, and Equation 5, with $N_\ell = N_r = 20$ and for a range of term frequency pairs $f_{t,\ell}$ and $f_{t,r}$. Note that Equation 5 gives matching negated values when the $f_{t,\ell}$ and $f_{t,r}$ inputs are swapped, whereas Equations 2 and 4 are not symmetric. When x is zero we take $\log_2 x$ to be zero.

$f_{t,\ell}$	$f_{t,r}$	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	3	1.12	0.96	0.58
2	5	1.75	1.57	1.32
5	2	−0.81	−0.80	−1.32
3	10	2.01	1.87	1.74
10	3	−1.41	−1.36	−1.74

4.4 Experiment: Estimators and Cooling

To assess the effectiveness and efficiency of the three estimators, and of the proposed cooling process, we compared them experimentally using the seven test collections described in Section 3.2. Table 3 lists the results, with effectiveness measured using *loggap* index cost (bits per posting), and efficiency as the time (elapsed seconds in the test environment, see Section 3.1) required to effect the reordering.

Three baseline document orderings were used as reference points, and are shown in the first three rows of the table: a randomly generated ordering (denoted Random); a URL-sorted ordering that is only applicable to the three inverted indexes; and a Length-based ordering, in which the documents are sorted by decreasing out-link count (web graphs) or by decreasing postings count (indexes). Note how both the URL and Length reorderings already give markedly superior compression effectiveness compared to Random.

The fourth row of Table 3 shows compression effectiveness and computation time for our reference implementation of the BP mechanism, following the description of Dhulipala et al. (Section 4.1). Input order does not have any great influence on the experimental outcomes, but for definiteness, the starting point for the BP measurements was always Length-ordered input for the four web graphs, and always URL-ordered input for the three inverted indexes. As a further external reference point, our measured throughput for the BP baseline in the fourth row of Table 3 is comparable to that of the optimized codebase described by Mackenzie et al. [21].

The remaining five rows in Table 3 then show all combinations of the cooling process (Section 4.2) and the two alternative estimators (Section 4.3). Cooling results in only slight loss of compression effectiveness on the web graphs, and almost no loss on the inverted indexes. Use of the alternative estimators brings more substantial compression degradation for the web graphs, but again has little or no effect on compression effectiveness on the inverted index data. On the other hand, both enhancements decrease execution times, and represent useful new options in the Pareto tradeoff space between effectiveness and efficiency.

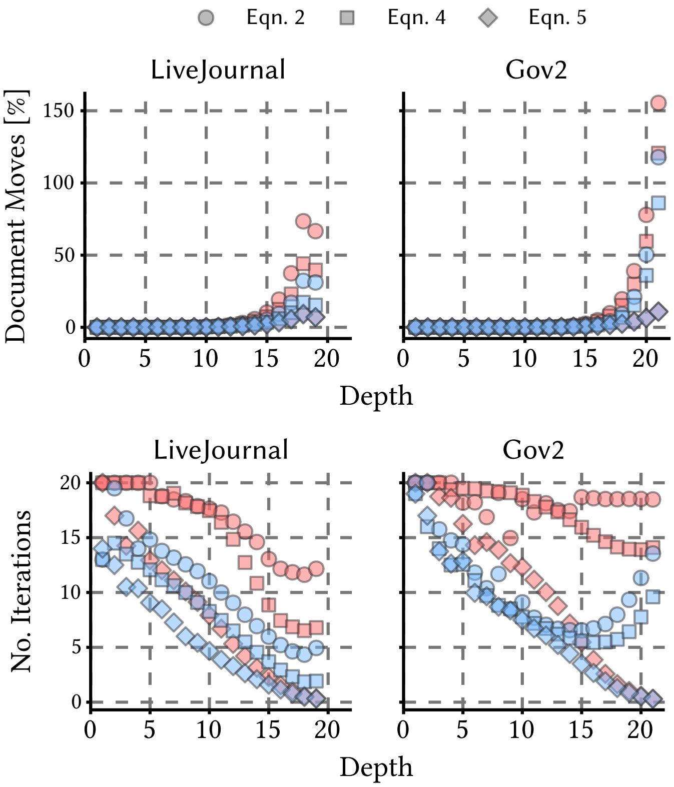


Fig. 4: Total number of documents moved at each recursive level, summed across all partitionings and iterations at that level, and then expressed as a percentage with respect to the total number of documents in the collection (top); and the average number of iterations in the set of recursions that take place at each level (bottom). Red-shaded points correspond to non-cooled methods; blue-shaded points show the addition of cooling.

Overall, the best tradeoff combination for web graphs appears to be estimation via Equation 2 together with cooling; and estimation via Equation 5 with cooling is the best combination for inverted indexes. The speed gain in the latter case is substantial – in combination the two factors reduce execution time to around 20–25% of the baseline BP implementation, with less than 1% of compression degradation introduced.

Table 4 gives total iteration counts summed across all of the recursive calls to function *partition_collection*() (Figure 2), discounted for each recursive depth d by a factor of 2^{d-1} to account for the fact that the iterations become cheaper as d increases, and then expressed as percentages of the same workload counts arising from the reference BP implementation (the fourth row of Table 3). The goal here is to assess workload by counting the "equivalent full-level iterations" required for each of the methods. As can be seen, both of the new estimators cut down the number of depth-adjusted iterations, and then the effect of the cooling process is additive. In the final column, between 50% and 80% of the original reordering effort has been bypassed, broadly matching the time savings shown in Table 3.

Figure 4 provides a further breakdown of some of this information, plotted as a function of increasing recursion

TABLE 3: Effectiveness (*loggap* bits per doc-gap) for three different initial document orderings and the baseline BP implementation, plus five enhanced BP versions, with time in elapsed seconds. Note that URL-based ordering cannot be applied to the four graphs. The best values in each of the columns are highlighted in blue.

	Enron		Facebook		Google		LiveJournal		Wikipedia		Gov2		CC-News-En	
	loggap	time	*loggap*	time	*loggap*	time	*loggap*	time	*loggap*	time	*loggap*	time	*loggap*	time
Random	8.98	–	9.33	–	13.79	–	15.68	–	5.38	–	5.75	–	3.71	–
URL	–	–	–	–	–	–	–	–	5.00	–	2.33	–	1.51	–
Length	5.63	–	7.75	–	9.56	–	13.02	–	4.29	–	2.36	–	1.59	–
BP, Eqn. 2	4.53	1.0	4.76	1.3	4.01	24	7.51	565	3.33	360	1.83	1967	1.29	5674
+Cooling	4.56	0.4	4.78	0.7	4.00	14	7.65	510	3.31	223	1.82	1475	1.28	4889
BP, Eqn. 4	4.61	0.7	4.85	0.7	4.02	21	7.82	491	3.30	150	1.82	1186	1.27	2954
+Cooling	4.70	0.2	4.93	0.4	4.08	13	8.08	441	3.32	77	1.82	794	1.27	2390
BP, Eqn. 5	4.82	0.2	5.01	0.4	4.16	14	8.23	446	3.37	92	1.84	557	1.29	1829
+Cooling	4.94	0.2	5.08	0.3	4.25	11	8.46	407	3.39	57	1.85	477	1.30	1535

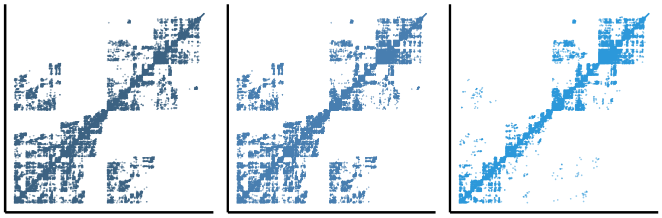
TABLE 4: Depth-adjusted iteration fractions (% of baseline BP) for different estimators, with and without cooling.

Collection	No Cooling		Cooling		
	Eqn. 4	Eqn. 5	Eqn. 2	Eqn. 4	Eqn. 5
Enron	75.1	37.4	31.5	24.3	20.2
Facebook	72.5	45.9	45.4	34.2	30.9
Google	91.1	48.2	55.3	43.1	28.4
LiveJournal	89.8	50.2	63.7	48.7	33.8
Wikipedia	76.3	58.3	48.9	33.0	29.1
Gov2	95.0	55.4	58.8	49.7	39.6
CC-News-En	84.7	66.0	76.7	64.5	52.8

level, and for one web graph and one of the text collections. The two top panes plot the number of documents moved, as a percentage of the number of documents in the collection, and show that at the lower levels of the recursion a non-trivial fraction of the collection gets moved, albeit relatively short distances, and more so for index data than graph data. The two lower panes show that cooling (the blue-shaded points) has a useful benefit at all levels in the recursive decomposition, reducing the number of iterations required before a fixed-point is reached.

4.5 Experiment: Comparing Estimator Permutations

To provide another perspective on the document orderings created by the three estimators (that is, Equations 2, 4, and 5), Figure 5 visualizes the relative document orderings generated for the Wikipedia inverted index. Each document is placed as a very small point in the two dimensional space corresponding to the two document orderings being compared; where sufficiently many documents are placed near each other, visible features emerge and can be discerned. As can be seen, 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.



Eqn. 2 vs. Eqn. 4 Eqn. 2 vs. Eqn. 5 Eqn. 4 vs. Eqn. 5

Fig. 5: Scatterplots of document orderings from three estimators on the Wikipedia collection. Visible features emerge only when a large number of documents occur in close proximity to each other.

4.6 Algorithmic Enhancement: Sort-Free Swapping

The focus of the previous few sections has been on reducing the number of iterations required at each recursive call, in order to reduce the overall execution time. There is also another way that the execution cost can be reduced, and that is via algorithmic improvement – it is possible to replace the $\mathcal{O}(N \log N)$ sorting steps (see Section 2.5) by an $O(N)$-expected time median finding step. When M/N is large, the substitution has no asymptotic effect on the overall expected time, but might nevertheless have a practical effect.

Consider Figure 2 again, and the emphasis we placed on the polarity of the signs associated with the biases. In our presentation, negative biases always indicate leftward attractions, and positive biases always indicate rightward attractions. That consistent relationship allows Figure 2 to be implemented more economically than previously.

In the description of Dhulipala et al., and in the previous experimental investigations of BP by Wang and Suel [16] and Mackenzie et al. [21], the "pair selection process" required at step 17 has been effected by first computing all the "*l2r*" document biases across D_L, then all of the "*r2l*" biases across D_R, with those calculations arranged so that positive values in both cases meant "this document would like to swap sides if it can". Each of those two sets of biases was then sorted into decreasing order, and a "zip" operation performed from largest to smallest, swapping in a pairwise manner by taking one document from each of the sorted lists until no net gain was possible. Those two sorting steps account for the $\mathcal{O}(N \log N)$ time factor that is required per iteration and per recursive level.

Figure 6 illustrates a new approach that makes use of our altered interpretation of the bias values. Now all document biases have a standardized polarity and a common scale,

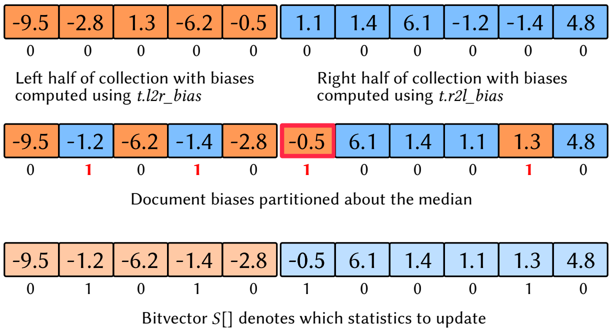

-9.5 -2.8 1.3 -6.2 -0.5 1.1 1.4 6.1 -1.2 -1.4 4.8
0 0 0 0 0 0 0 0 0 0 0
Left half of collection with biases computed using *t.l2r_bias*
Right half of collection with biases computed using *t.r2l_bias*
-9.5 -1.2 -6.2 -1.4 -2.8 -0.5 6.1 1.4 1.1 1.3 4.8
0 1 0 1 0 1 0 0 0 1 0
Document biases partitioned about the median
-9.5 -1.2 -6.2 -1.4 -2.8 -0.5 6.1 1.4 1.1 1.3 4.8
0 1 0 1 0 1 0 0 0 1 0
Bitvector *S*[] denotes which statistics to update

Fig. 6: Median-finding in the bias array: before the identification of the median (top); after the median of -0.5 has been placed in its correct location (middle); and before reconstructing the statistics in the left and right partitions (bottom). The role of bitvector $S[\,]$ is explained in the text.

and it is sufficient to identify the median bias m over the whole set D, which then automatically separates the documents in $D_L \cup D_R$ into $\leq\ m$ and $\geq\ m$ sets of the same size. Now all of the document swaps arise as a natural consequence of the median-finding process, ensuring that the left half of D contains exactly the required set of "more negative" document biases (in some order), and the right half similarly contains the corresponding set of "more positive" document biases. The critical factor that makes this an attractive change is that median finding requires only $\mathcal{O}(N)$ expected time [36], removing a factor of $\mathcal{O}(\log N)$ from that component of the execution time. Given that $M \geq N$ as a necessary condition on the inputs, the first term then unambiguously dominates, and the overall execution time (see Section 2.5) thus becomes $\mathcal{O}(M \log_2 N)$ on average; or becomes $\mathcal{O}(LM \log_2 N)$ on average with the factor L included, noting that L is common to both the previous and the new implementations.

The drawback of the median-based approach is the need for a bitvector $S[\,]$ of size N to be maintained, with $S[d]$ recording whether document d is currently considered to be a member of its original partition, or whether it has switched sides. Each particular document might switch sides zero, one, or more times while the median-finding is playing out. Once the median has been identified it is a simple matter to locate and exchange left-right pairs of documents d for which both $S[d]$ values have odd parity.

4.7 Experiment: Sorting or Selecting?

The baseline BP implementation includes two sorting calls per iteration, both implemented as parallel (that is, multi-threaded) functions using the flexibility afforded by the experimental environment (see Section 3.1). Sorting is a task that has a natural decomposition into subtasks, and hence allows useful speedups in elapsed running time when multiple processors are available. In contrast, median-finding requires less computational work than sorting, but is also less amenable to parallelism. It is thus an interesting question as to whether replacing the two sorting steps by a median-finding step will result in reductions in elapsed (that is, wall-clock) processing times.

TABLE 5: Elapsed computation time (seconds, Equation 5, with cooling).

	ParallelSort	SequentialSort	FloydRivest
Enron	0.162	0.157	0.169
Facebook	0.347	0.339	0.350
Google	8.68	9.42	9.06
Wikipedia	58.6	66.4	59.7
LiveJournal	438	480	475
Gov2	490	560	448
CC-News-En	1590	1940	1640

The first column in Table 5 shows the time required by that baseline configuration. The second and third columns then show what happens when that arrangement is altered: first, by using a sequential rather than parallel sort, to demonstrate the time saving attributable to the parallelism in the baseline sort; and second, when the sort is removed and replaced by the Floyd-Rivest median selection algorithm [36], which runs in $\mathcal{O}(N)$ expected time. Except for the largest collection, the modest differences between the first and second columns indicates that sorting is only a small fraction of the total resource cost required during reordering, confirming the discussion in Section 2.5 that suggested that the $\mathcal{O}(M \log N)$ term was likely to dominate the $\mathcal{O}(N \log^2 N)$ cost of sorting, because of the underlying relationship between M and N.

Comparing the second and third columns in Table 5 then shows that switching to the median-based approach yields CPU savings on all but the two smallest collections. The gain is smaller (or even non-existent) relative to the parallel sort, but it should be noted that we are reporting elapsed times, and embedded in the Floyd-Rivest approach are unused threads that correspond to reduced computational effort. Those CPU savings might be spent on other unrelated tasks if such tasks are available, but the lack of within-task parallelism means that elapsed time savings should not be expected. Comparing the first and the third confirms that position – even when looking only at five larger files for which CPU time savings can be achieved, there are just two for which the use of median-finding makes for smaller elapsed computation time. That is, the choice between a sorting-based implementation or a median-based implementation must be decided as a compromise between efficiency of resource usage, and minimizing elapsed computation time.

5 Engineering Considerations

The final sequence of experiments explores a number of engineering issues that arose as part of our investigation into BP computations. In this section we narrow the focus to one set of implementation options: the estimation of Equation 5, with cooling in operation, and employing the Floyd-Rivest median-based approach.

5.1 Excessive Parallelism

The experiments reported in Section 4 all employed the greatest possible level of innate parallelism – each recursive call was spawned as a new process, and within those recursive calls, the gain computations and sort calls (where

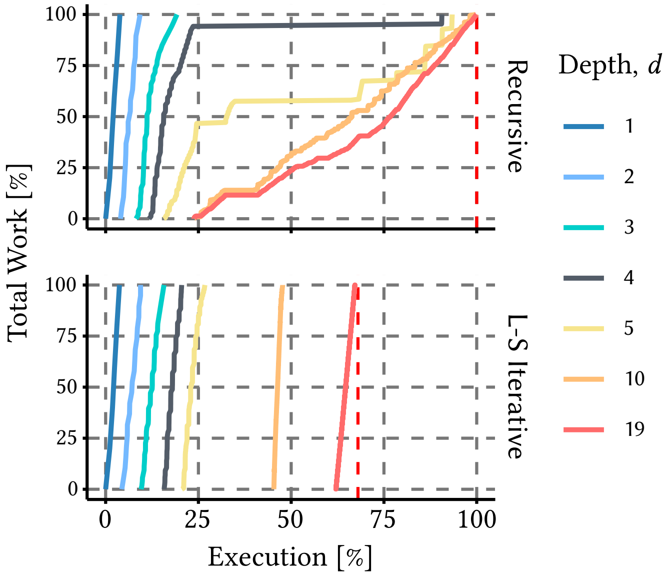


Fig. 7: Level-by-level performance tracking of the recursive "maximally parallel" recursive BP implementation (top pane, as described by Figure 1 and Figure 2, using Equation 5 estimation, cooling, and median-based swapping), and the level-synchronized iterative version (bottom pane, as described by Figure 8, with other options constant) when processing the Wikipedia dataset. Seven different partitioning levels were selected and tracked through the course of the two computations.

applicable, see Section 4.7) were also spawned as independent tasks, so that the Rayon work scheduler was free to pursue parallel execution if it had the capacity to do so. That means that there were potentially many thousands of tasks concurrently active and capable of being assigned to the available physical processing threads, with some of the tasks needing orders of magnitude more resource than others.

Profiling evaluations showed that such extreme levels of disparate-sized parallelism could result in significant contention bottlenecks. The top pane in Figure 7 illustrates the consequences of this, using the Wikipedia dataset as an example. Iteration commencement and completion timestamps were extracted for seven of the different partitioning depths, and the progress through each of those level's total work was tracked, where "work" was counted via completed iterations of the main loop in Figure 2, summed across all that depth's partitions. The recursive version is slowed by very high levels on inter-level contention, with the last of the level $d = 4$ partition delayed by an onrush of tasks representing partitions at level $d = 10$ and beyond. That imbalance can be tracked back to the early ending of one of the $d = 3$ partitions, which releases the first of the $d = 4$ partitions, and starts a rapid cascade of subsequent tasks.

We explored two possible remedies. The first is to cut down on the number of concurrent tasks spawned, shifting from a fully parallel to a *partially parallel* mode. Instead of allowing every set of term frequency and bias computations to be executed as parallel tasks, we implemented sequential counterparts of those operations as well. At the early levels of recursion, the parallel versions of those functions were called. Then, at some given depth d' in the recursive decomposition, the parallel calls were ended, and sequential functions employed, thereby "bulking up" the separated tasks into a smaller number of longer-duration activities. In this partially parallel version the recursive calls themselves (see Figure 1) continued to be issued in parallel at all recursive levels.

```
 1: function reorder_collection_itr($(D, N, d_0)$):
 2: // reorder $D[0 \ldots N-1]$ by iteratively processing it
 3: // as a hierarchical sequence of document sections,
 4: // starting at the $d_0$ th level of the hierarchy
 5: set $d\_max \leftarrow \lfloor \log_2(N/min\_size) \rfloor$
 6: set $Q_1 \leftarrow \langle D[0 \ldots N-1], N \rangle$
 7: for $d \leftarrow 1$ to $d\_max$ do
 8:     set $Q_{d+1} \leftarrow [\,]$
 9:     for every $\langle D', N' \rangle$ in $Q_d$ paralleldo
10:         if $d \geq d_0$ then
11:             partition_collection$(D', N')$
12:         set $Q_{d+1} \leftarrow Q_{d+1} \cup$
13:                 $\langle D'[0 \ldots N'/2-1], N'/2 \rangle \cup$
14:                 $\langle D'[N'/2 \ldots N'-1], N'-N'/2 \rangle$
15:     end parallelfor
16:     // wait here for all parallel tasks at level $d$ to finish
```

Fig. 8: The level-synchronized iterative implementation of the bipartite partitioning (BP) process. All subtasks at level d are permitted to execute in parallel; all subtasks at level d must complete before any subtasks at level $d+1$ are initiated.

The ideal is to have a task be available for any processing thread that becomes vacant, but not so many that it risks taking too long to select and assign a task compared with the cost of executing it. We hypothesized that a suitable transition point would be at $d' = \log_2 p$ where p is the total number of processors available. That is, we expected $d' \approx 5$ to be an appropriate choice for our hardware. In fact $d' \approx 10$ was a better choice for most of the experimental datasets; we comment further on this finding shortly.

5.2 Level-Synchronized Iterative Computation

The second possible remedy to limit the disparities in the sizes of competing tasks was to reimplement the BP logic as an *level-synchronized iterative* process, manipulating explicit queues of pending "sections" of the collection D, in the same way that non-recursive implementations of Mergesort and Quicksort can be implemented via nested loops and a queue. This approach is illustrated in Figure 8.

As each section of the collection is taken from the level-d queue Q_d (step 9 in Figure 8) it is split into two halves and then reordered through document swaps using the previous mechanism (step 11, which calls the function described in Figure 2); and then the two half-sized tasks that result are both appended to the Q_{d+1} queue, to be processed at some later time (steps 12–14). At each iteration of the global control loop (starting at step 7), all sections at the same depth d in the recursion are handled in parallel and are

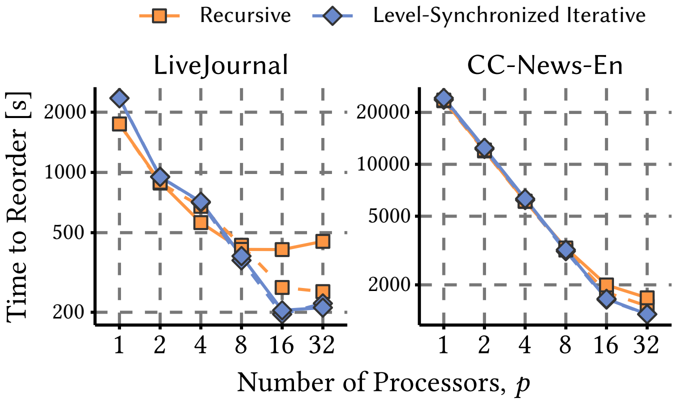


Fig. 9: Time to compute BP reorderings, plotted as a function of the number of permitted processing threads. The solid lines with squares represent the starting configuration: a recursive implementation based on Equation 5 estimation, with cooling, median-based reordering, and "maximal" parallelism. Dashed lines represent the addition of the partial parallelism technique; and the solid lines with diamonds indicate the use of the level-synchronized iterative implementation.

simultaneously active, with an explicit *wait* operation enforced at the end of that level-d batch (step 15) to ensure that all depth d sections are completed before any $d+1$ sections are allowed to become active. That is, a strict ordering is imposed, with sections at the same depth in the recursion tree permitted to execute in parallel with each other, but not ever allowed to compete with sections at levels $d-1$ or $d{+}1$. This final BP implementation has no recursive function calls, but if called with $d_0 = 1$ will compute exactly the same document reordering as the simpler implementation we started with, shown in Figure 1. The purpose of the additional parameter d_0 is discussed in Section 5.5, below.

The lower pane in Figure 7 shows the benefits achieved by this approach. There is now an orderly sequence of work performed, stepping through the various sub-tasks in a strictly level-order (breadth-first) manner. Moreover, more than 25% of the recursive version's elapsed execution time can be saved.

5.3 Experiment: Controlling Parallelism

We compared the recursive and level-synchronized iterative BP implementations, testing versions both with and without partial parallelism (four methods in total), measuring elapsed execution time as a function of the number of processing threads that were permitted. Figure 9 shows the results for one of the web graph inputs, and one of the inverted index inputs. While some of the effects are small (note that some of the lines in the graph are occluded), the recursive implementation clearly benefits from the imposition of partial parallelism. On the other hand, partial parallelism is only of limited benefit to the level-synchronized iterative implementation, because the iterative approach already explicitly ensures task size consistency via the *wait* constraint that occurs once per level d. As was anticipated by Figure 7, the level-synchronized iterative implementation is notably faster than the recursive version, and also slightly faster than the partial parallelism variant of the recursive implementation, possibly as a consequence of a reduced volume of book-keeping overheads.

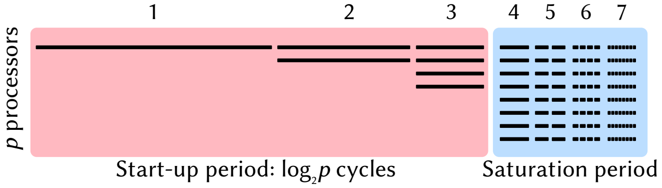


Fig. 10: Stylized depiction of task structure in a parallel BP implementation in which each problem of size M is processed on p (shown here as $p = 8$) processors via two parallel recursive calls of size $M/2$, with no other parallelism employed. This diagram assumes that all subtasks at each depth d complete before any at depth $d+1$ commence, and hence reflects the level-synchronized iterative implementation.

5.4 Parallel Processing Wastage

Another interesting aspect of the experiment in Figure 9 is the "imperfect" speedups attained when doubling the number of available processors, especially between $p = 16$ and $p = 32$. Why does doubling the available computing power only result in execution-time speedups of factors between 1.0 and 1.3?

To understand what is happening, consider Figure 10. It presents a highly stylized representation of how an input of size M would be processed recursively on a pool of p processors under the simplifying assumptions that the time taken at each recursive call is $\mathcal{O}(M)$ (that is, taking only the first term in the analysis, see Section 2.5); and that as each partition is processed there is no further parallelism possible. (As noted above, this is not the case, and in our implementation parts of Figure 2 are in fact parallelized, including term frequency counting and document bias calculations.) In the diagram it is also assumed that the two recursive calls are balanced in terms of M (whereas they are actually balanced in terms of N); that the *min_size* stopping point shown in Figure 1 is also applied to M (rather than to N); and that for now, *min_size* $= 1$. The effect of increasing *min_size* is discussed shortly.

Given these assumptions, the top-level task requires M steps of sequential computation, and then spawns two half-sized jobs that each require $M/2$ steps of computation. Then, when those two finish, four quarter-sized jobs are spawned, and so on – a standard recursive structure that leads to $\log_2 M$ recursive levels, with a total of M "units of work" across each recursive level. In this execution scenario there would thus be $M \log_2 M$ units of work required in total. If all p processors could be busy at all times, and perfect parallelism achieved subject to the assumptions listed, then $(M \log_2 M)/p$ units of elapsed time would be sufficient. But the start-up period shown in Figure 10 adds a non-trivial overhead, because during the start-up period there must be idle processors. In the assumed scenario it is only

TABLE 6: Using Equation 6 and the preceding analysis and assumptions to compute the elapsed duration counted in nominal "time units", together with the fraction of the total processor capacity through that period that is productively useful for the BP task, and the fraction that is wasted, for different combinations of p and *min_size*, and with $M = 10^8$. The "useful" percentages are a lower bound on what occurs in our BP implementations, and the "wasted" percentages are an upper bound.

M	p	*min_size* $= 1$			*min_size* $= 16$		
		Dur.	%usef.	%wast.	Dur.	%usef.	%wast.
10^8	1	2.7×10^9	100.0	0.0	2.3×10^9	100.0	0.0
10^8	2	1.4×10^9	96.4	3.6	1.2×10^9	95.8	4.2
10^8	4	7.6×10^8	86.9	13.1	6.6×10^8	84.9	15.1
10^8	8	4.7×10^8	70.7	29.3	4.2×10^8	67.2	32.8
10^8	16	3.3×10^8	50.5	49.5	3.0×10^8	46.5	53.5
10^8	32	2.6×10^8	31.8	68.2	2.5×10^8	28.4	71.6

when p parallel recursive calls have been spawned that all processors can be fully engaged, with that situation then continuing until all of the required work has been completed and the computation completes.

If there are p processors available, then under the simplifying assumptions the start-up period must span the first $\log_2 p$ recursive levels, as shown in the figure. Moreover, the start-up period is $2M(1 - 1/p)$ units long. Over the p processors the total available processing capacity during the start-up period is thus $2M(p-1)$ work units. But the simplifying assumptions mean that only $M \log_2 p$ units of useful work are performed during those first $\log_2 p$ levels, and so the start-up period must contain $w = 2M(p-1) - M \log_2 p$ units of unused processing capacity – or, more precisely, processing capacity that cannot be usefully employed on the BP computation, but can be deployed to other unrelated computations if any are available.

With $M \log_2 M$ units of useful work to be accomplished, the total work capacity required during the computation must be $M \log_2 M + w$, that is $M(\log_2(M/p) + 2p - 2)$. The fraction of that time spent on useful computation is thus

$$\frac{\log_2 M}{\log_2(M/p) + 2p - 2} \,. \qquad (6)$$

Given the restrictions imposed by the simplifying examples, this computation should be regarded as providing a *lower* bound. Nevertheless, it is informative. Table 6 shows the effect of Equation 6 when it is applied to values typical of the test sets described in Section 3.2. When $p = 1$ there is no wastage, and when $p = 2$ there is at most only a modest amount of wastage. But once $p \geq 4$ the wastage rate potentially exceeds 10%, and for $p \geq 8$, the wastage might be in excess of 30%.

The situation is further exacerbated if *min_size* is increased. Doing so reduces the total amount of useful work from $M \log_2 M$ to $M \log_2(M/\mathit{min\_size})$, without affecting the wastage that occurs during the start-up phase. That is, the fraction of wastage must increase as *min_size* increases. The three columns at the right of Table 6 show the impact of this relationship: processing duration decreases, but a greater fraction of the total available computational power is unused. All of the experiments shown in the earlier parts of this paper used *min_size* $= 16$.

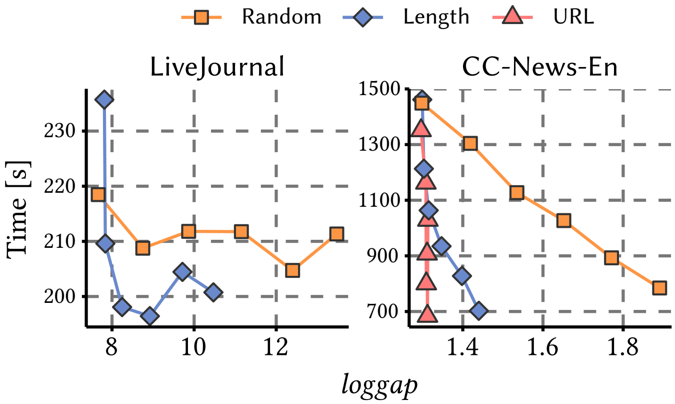


Fig. 11: Trade-off curves between execution time and *loggap* effectiveness when skipping the first $d_0 - 1$ recursive levels, for $d_0 \in \{1, 2, 3, 4, 5, 6\}$, with $d_0 = 1$ at the top-left of each curve and $d_0 = 6$ at the bottom-right. The method throughout is defined by Equation 5, with cooling enabled, and the level-synchronized iterative implementation. This heuristic is most effective when the initial configuration is already ordered in some way.

5.5 Experiment: Reducing the Start-Up Period

The potential for non-trivial wastage, documented in Table 6, also represents an opportunity: if the allocated resources can be used more productively, then elapsed computation times can be reduced. The obvious place to try for savings is at the start of the computation, when the greatest fraction of the processor pool is potentially unable to be used, see Figure 10. Figure 11 shows one straightforward way in which that can be done.

To create the two graphs in Figure 11, the level-synchronized BP implementation (using Equation 5, cooling, and median-based swapping) was executed in a $d_0 = 1$ configuration, then a $d_0 = 2$ configuration, and so on, where d_0 indicates which level of the hierarchical partitioning process the BP mechanism is commenced at (see Figure 8). For example, a $d_0 = 3$ computation commences processing with four quarter-collections, rather than one full collection; with no document swapping ever considered across those three section boundaries, and with hierarchical reordering commencing in parallel within those four sections. Documents initially far apart that might otherwise be placed near each other can thus get "marooned", meaning that compression rates might be affected. But bypassing the work associated with the first d_0 levels of partitioning, and more importantly, the potential resource wastage associated with them, might yield a desirable tradeoff.

The two panes in Figure 11 show, respectively, a web graph and an inverted index. Each curve is for a different initial ordering of the collection, as discussed in connection with Table 3. In the right-hand pane the inverted index exhibits the behavior we seek: if all partitioning levels are processed (the top-left ends of the three curves), the BP computation yields excellent compression regardless of

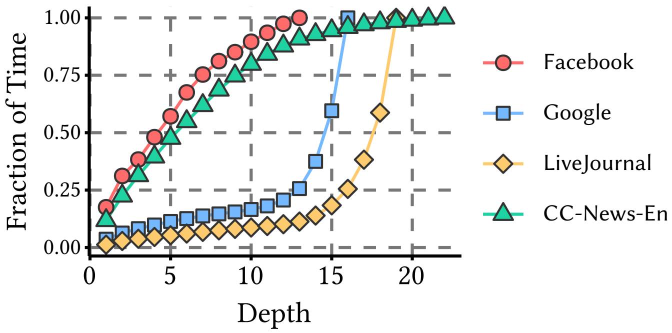


Fig. 12: Cumulative elapsed (wall-clock) time, as a function of partition depth, and expressed as a fraction of final processing time, with experimental settings equivalent to the $d_0 = 1$ curves in Figure 11. When applied to the LiveJournal and Google datasets it is clear that relatively little of the overall computation occurs in the first few levels, and that the deepest levels are where the majority of the effort is being expended.

the initial ordering, but is relatively slow. Moreover, if the collection is in a random order, all partitioning levels need to be processed to get maximum compression. But if the input collection already has some appropriate ordering – URL being ideal – then multiple levels of partitioning can be safely bypassed, resulting in substantial further time savings on top of those already documented in previous experiments, with almost no loss of compression effectiveness. Initial ordering by Length is also an effective strategy.

The situation in the left pane of Figure 11 is more complex. Starting with an ordered collection is still beneficial, but the time savings are relatively small, do not continue to accrue as d_0 is increased, and the compression loss as d_0 grows is non-trivial. Figure 12 shows why that different behavior arises. It plots cumulative wall-clock times as the level-synchronized BP computation proceeds through the levels, presented as fractions of total time. Two of the four collections shown have a distinctly different execution profile, and spend more time in deep computations than they do at the early levels. For these two datasets the assumptions made in order to undertake the analysis of Section 5.4, and that are visualized in the neat regularity of Figure 10, simply do not apply. On the other hand, CC-News-En and Facebook do somewhat match the anticipated computational structure, with the first few partition levels the slowest ones – but still not to the extent suggested by the simplified situation depicted in Figure 10.

5.6 Overall Gains

Table 7 summarizes the developments that have been presented in Sections 4 and 5. It compares the *loggap* compressibility and processing time of the initial BP reference implementation as documented in the fourth row of Table 3 with the final version we have arrived at in this section, with values less than 100% representing "better", and values greater than 100% representing "worse". This final version of BP uses the level-synchronized processing strategy, uses Equation 5 for estimation, uses the cooling strategy, and uses median-based swapping. As can be seen, we have made great strides in terms of reducing BP execution time, and have achieved those gains at a very modest cost in terms of loss of *loggap* compression effectiveness.

TABLE 7: Final compression effectiveness and elapsed computation times, expressed as percentages (and multiplicative speed equivalents) relative to our initial BP reference implementation shown in the fourth row of Table 3, and measured for the level-synchronized iterative implementation using Equation 5 estimation, cooling, and median-based swapping.

Collection	Relative effectiveness	Relative elapsed time	
Enron	96.1%	6.1%	(16.5×)
Facebook	102.2%	11.3%	(8.9×)
Google	100.9%	16.9%	(5.9×)
LiveJournal	104.0%	37.7%	(2.7×)
Wikipedia	101.5%	11.6%	(8.6×)
Gov2	101.8%	18.1%	(5.5×)
CC-News-En	100.6%	24.0%	(4.2×)

5.7 The Opposite Tradeoff

The majority of our work in this paper has focused on reducing execution times while – as far as possible – holding compression effectiveness unchanged. But the tradeoff curve between effectiveness and efficiency can also be extended in the other direction.

As a final experiment, we revisit the values of *min_size* (the minimum size of each BP partition, which has the effect of controlling the depth of the BP computation), and the lower term frequency cutoff (the minimum term frequency in the three inverted indexes, below which terms were not considered during the reordering process). All of the experiments reported above make use of $min\_size = 16$ and a minimum frequency of 4,096 postings. We are interested in the extent to which useful compression gains can be achieved if those limits are decreased, and also in the interplay between those two variables. Intuitively, when the index partitions are very small, most of the term statistics are likely to be similar, and hence will have relatively little impact on the reordering process. However, if at the same time we allow shorter lists to be considered, then BP may be able to "squeeze" more compression out of the small partitions, albeit by expending additional computation time.

To investigate these possibilities, we recomputed the three inverted index reorderings without any lower term frequency cutoff at all, and at the same time with *min_size* reduced from 16 to 8. Using the bottom row of Table 3 as a reference point, compression could be improved by up to 5%, but with pronounced increases in elapsed time: 12× longer, 10× longer, and 8× longer for Wikipedia, Gov2, and CC-News-En respectively. So, while slight improvements in compression effectiveness are still possible, they are achieved only with a concomitant – and rather dramatic – increase in computational resources required.

6 Conclusion

We have carried out a detailed exploration of Dhulipala et al.'s bipartite graph partitioning algorithm, and described several enhancements that allow the execution time to be reduced with little or no reduction in effectiveness, where effectiveness is quantified in terms of the "*loggap*" compressibility of the reorderings generated. In particular, we have developed a new cooling mechanism that helps limit the number of partitioning iterations required at each recursive level; have developed two additional swapping cost estimators, designed to provide more stability in the estimations that are generated as the computation proceeds; have shown that the computation can be reoriented, so that a sorting step can be replaced by a more efficient median-finding process; and have developed a level-synchronized iterative version that avoids the high degrees of inter-level contention experienced by the fully recursive implementation, and that readily supports accelerated operations if initial partitioning levels are to be bypassed.

To confirm our improvements, we have carried out a detailed experimental investigation using four typical web graphs and three typical inverted file indexes. A wide range of results have been presented, including a demonstration that *loggap* can indeed be used as a codec-independent surrogate for measuring compression effectiveness. Our main sequence of results has demonstrated that in combination our techniques allow BP reordering times to be reduced by a factor of around four, and by even more in terms of workload (that is, when summed across execution threads), with little or no loss of compression effectiveness. In addition, we have explored the time-savings that are possible if an initial ordering of the input is possible through the use of URL-based sorting or Length-based sorting, and the partitioning process commenced on sub-collections rather than the full collection. We have also observed that improved compression effectiveness can be achieved if further execution time can be traded-off to obtain it.

Finally, we note that our code is made available publicly so that others are able to build on our findings and continue to investigate this important and interesting topic.

Software

The experimental software is available at https://github.com/jmmackenzie/enhanced-graph-bisection.

Funding

This work was supported by the Australian Research Council (project DP200103136).

Acknowledgment

We thank the referees for their helpful comments.

References

[1] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys*, vol. 38, no. 2, pp. 6:1–6:56, 2006.

[2] G. E. Pibiri and R. Venturini, "Techniques for inverted index compression," *ACM Computing Surveys*, vol. 53, no. 6, pp. 125.1–125.36, 2021.

[3] S. Ding and T. Suel, "Faster top-k document retrieval using block-max indexes," in *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, 2011, pp. 993–1002.

[4] G. Ottaviano and R. Venturini, "Partitioned Elias-Fano indexes," in *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, 2014, pp. 273–282.

[5] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita, "Compressing graphs and indexes with recursive graph bisection," in *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 1535–1544.

[6] J. Mackenzie, M. Petri, and A. Moffat, "Faster index reordering with bipartite graph partitioning," in *Proc. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, 2021, pp. 1910–1914.

[7] A. Moffat and L. Stuiver, "Binary interpolative coding for effective index compression," *Information Retrieval*, vol. 3, no. 1, pp. 25–47, 2000.

[8] G. E. Pibiri and R. Venturini, "Clustered Elias-Fano indexes," *ACM Trans. on Information Systems*, vol. 36, no. 1, pp. 2.1–2.33, 2017.

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

[10] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *Proc. Conf. on the World Wide Web (WWW)*, 2009, pp. 401–410.

[11] S. Ding, J. Attenberg, and T. Suel, "Scalable techniques for document identifier assignment in inverted indexes," in *Proc. Conf. on the World Wide Web (WWW)*, 2010, pp. 311–320.

[12] D. Hawking and T. Jones, "Reordering an index to speed query processing without loss of effectiveness," in *Proc. Australasian Document Computing Symp. (ADCS)*, 2012, pp. 17–24.

[13] A. Mallia, M. Siedlaczek, and T. Suel, "An experimental study of index compression and DAAT query processing methods," in *Proc. European Conf. on Information Retrieval (ECIR)*, 2019, pp. 353–368.

[14] J. Mackenzie and A. Moffat, "Examining the additivity of top-k query processing innovations," in *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, 2020, pp. 1085–1094.

[15] J. Mackenzie, M. Petri, and A. Moffat, "Anytime ranking on document-ordered indexes," *ACM Trans. on Information Systems*, vol. 40, no. 1, pp. 13:1–13:32, Jan. 2022.

[16] Q. Wang and T. Suel, "Document reordering for faster intersection," *PVLDB*, vol. 12, no. 5, pp. 475–487, 2019.

[17] F. Silvestri, "Sorting out the document identifier assignment problem," in *Proc. European Conf. on Information Retrieval (ECIR)*, 2007, pp. 101–112.

[18] D. Blandford and G. Blelloch, "Index compression through document reordering," in *Proc. IEEE Data Compression Conf. (DCC)*, 2002, pp. 342–352.

[19] W. Shieh, T. Chen, J. J. Shann, and C. Chung, "Inverted file compression through document identifier reassignment," *Information Processing & Management*, vol. 39, no. 1, pp. 117–131, 2003.

[20] R. Blanco and A. Barreiro, "Document identifier reassignment through dimensionality reduction," in *Proc. European Conf. on Information Retrieval (ECIR)*, 2005, pp. 375–387.

[21] J. Mackenzie, A. Mallia, M. Petri, J. S. Culpepper, and T. Suel, "Compressing inverted indexes with recursive graph bisection: A reproducibility study," in *Proc. European Conf. on Information Retrieval (ECIR)*, 2019, pp. 339–352.

[22] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. Conf. on the World Wide Web (WWW)*, 2004, pp. 595–602.

[23] C. Cheng, C. Chung, and J. J. Shann, "Fast query evaluation through document identifier assignment for inverted file-based information retrieval systems," *Information Processing & Management*, vol. 42, no. 3, pp. 729–750, 2006.

[24] R. Blanco and A. Barreiro, "TSP and cluster-based solutions to the reassignment of document identifiers," *Information Retrieval*, vol. 9, no. 4, pp. 499–517, 2006.

[25] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2009, pp. 219–228.

[26] P. Boldi, M. Santini, and S. Vigna, "Permuting web and social

graphs," *Internet Mathematics*, vol. 6, no. 3, pp. 257–283, 2009.

[27] V. N. Anh and A. Moffat, "Local modeling for WebGraph compression," in *Proc. IEEE Data Compression Conf. (DCC)*, 2010, p. 519.

[28] D. Arroyuelo, M. Oyarzún, S. González, and V. Sepulveda, "Hybrid compression of inverted lists for reordered document collections," *Information Processing & Management*, vol. 54, no. 6, pp. 1308–1324, 2018.

[29] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proc. WOSN*, 2009, pp. 37–42.

[30] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2006, pp. 44–54.

[31] P. Yang, H. Fang, and J. Lin, "Anserini: Reproducible ranking baselines using lucene," *ACM Journal of Data and Information Quality*, vol. 10, no. 4, pp. 1–20, 2018.

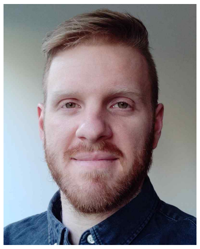
[32] J. Mackenzie, R. Benham, M. Petri, J. R. Trippas, J. S. Culpepper, and A. Moffat, "CC-News-En: A large English news corpus," in *Proc. ACM Int. Conf. on Information and Knowledge Management (CIKM)*, 2020, pp. 3077–3084.

[33] 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. ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR)*, 2020, pp. 2149–2152.

[34] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Software Practice & Experience*, vol. 41, no. 1, pp. 1–29, 2015.

[35] D. Lemire, N. Kurz, and C. Rupp, "Stream VByte: Faster byte-oriented integer compression," *Information Processing Letters*, vol. 130, pp. 1–6, 2018.

[36] R. W. Floyd and R. L. Rivest, "Expected time bounds for selection," *Communications of the ACM*, vol. 18, no. 3, pp. 165–172, 1975.



Joel Mackenzie completed a PhD in computer science at RMIT University in Australia in 2019. He has since worked as a postdoctoral research fellow at The University of Melbourne, and is now a faculty member at The University of Queensland, Australia. Dr Mackenzie is interested in developing efficient and effective representations for large-scale search systems, including indexing, compression, and retrieval.

Alistair Moffat completed a PhD in computer science at the University of Canterbury in New Zealand in 1986. Since then he has been a faculty member at The University of Melbourne in Australia, with interests in text and index compression, and algorithms for string search and information retrieval, including information retrieval evaluation. Professor Moffat was inducted as a member of the SIGIR Academy in 2021.

Matthias Petri completed a PhD in computer science at RMIT University in 2013. He then worked as a postdoctoral fellow at The University of Melbourne before joining Amazon Alexa in Los Angeles in 2019. Dr Petri has research interests in succinct data structures, techniques for efficient implementation of natural language processing systems, and in indexing, compression, and searching methodologies for information retrieval.