Cost-Effective Updating of Distributed Reordered Indexes

Joel Mackenzie
The University of Melbourne
Melbourne, Australia

Alistair Moffat
The University of Melbourne
Melbourne, Australia

ABSTRACT

Index reordering techniques allow document collections to be renumbered, with the goal of developing a permutation of the initial document ordinal identifiers that places documents that are (somehow) like each other into positions near each other in the permuted ordering. The clustering that results allows inverted index size to be reduced, since each term's posting list is more likely to contain a non-uniform set of inter-document integer gaps. Reordering is normally performed once, at the time the index is created.

Here we consider the role of index reordering in collections that grow over time, noting that simply appending new documents to the collection may erode the effectiveness of an earlier reordering. In particular, we discuss methods for maintaining and reinstating reorderings as document collections grow, and measure the effectiveness of those techniques on a large corpus of English news articles. We also provide experimental results that illustrate the benefits of reordering in terms of query execution time.

KEYWORDS

Extensible collection; document reordering; document clustering; querying; dynamic pruning

ACM Reference Format:
Joel Mackenzie and Alistair Moffat. 2021. Cost-Effective Updating of Distributed Reordered Indexes. In *Australasian Document Computing Symposium (ADCS '21), December 9, 2021, Virtual Event, Australia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3503516.3503528

1 INTRODUCTION

We consider the competing tensions that arise when the goal is to minimize the size of the index associated with an information retrieval system, but, at the same time, accommodate an ongoing stream of documents that must be incorporated while the system is operational and serving answers to queries. In particular, document reordering techniques assume that the ordinal numbers attached to the stored items can be permuted, to create an arrangement that is more amenable to compression. Such techniques are readily applied when the collection is static and the ideal document sequence can be computed prior to the deployment of the retrieval service. But extensions to the collection might disrupt such arrangements, and render them unhelpful; nor is it likely to be possible for new documents to be inserted in amongst the current document ordering, because effective index compression relies on the document numbering being dense.

Our investigation in this paper considers methods for reconciling these two competing objectives – retaining the benefits of a reordered index (less storage for the index, and faster query processing) but allowing documents to be appended to the growing collection. We note that Wang and Suel [23] explicitly mention index maintenance as a possible confound for document reordering, and so our work here can be seen as an investigation that at least in part responds to their question.

Section 2 provides a general background to inverted indexing and an overview of document reordering techniques, with a focus on one particular recent method; and then describes the operational cycle that we assume for extensible indexing. Sections 3 and 4 then consider, respectively, the implications of document acquisition and reordering on index size, and the effect that various possible approaches have on query processing times.

2 BACKGROUND

Inverted Indexing and Compression. The inverted index is widely used as a structure for supporting ranked "bag of words" querying over collections of documents. Each term in the collection has a postings list associated with it, containing the ordinal numbers of the documents that contain one or more instances of that term. Queries are evaluated by merging and/or intersecting the postings lists of the terms that appear in the query. Zobel and Moffat [26] provide an overview of these processes.

The postings list I_t for some term t in a collection of N documents consists of a sequence of tuples $\langle d_{t,i}, f_{t,i}\rangle$, where $0 \le d_{t,i} < N$ is the ordinal number associated with the i th document in which t appears, and $f_{t,i}$ is t's occurrence frequency in document $d_{t,i}$. Each term has a collection frequency f_t associated with it; its posting list is thus $I_t = [\langle d_{t,i}, f_{t,i}\rangle \mid 0 \le i < f_t]$. Note that the $d_{t,i}$ values are unique within each postings list, and that the tuples are normally (in a document-sorted index) arranged in ascending $d_{t,i}$ order. It is then usual for "d-gaps" to be formed, and for a sequence of tuples $\langle d_{t,i} - d_{t,i-1}, f_{t,i}\rangle$ to be stored, with $d_{t,-1} \equiv -1$ for all terms t. Again, see Zobel and Moffat [26] for more information. Finally, those gaps are represented using a variable-length integer code that assigns shorter codewords to smaller integers [21, 26]. Terms that are frequent across the collection thus get shorter codewords on average than terms that are rare, and the inverted index is reduced in size compared to using the raw $d_{t,i}$ values, and compared to using a fixed-width binary code of $\lceil \log_2 N \rceil$ bits for the $d_{t,i}$ values.

Document Reordering. Blandford and Blelloch [1] noted that further space savings can be achieved via *index reordering* – permuting the ordinal arrangement of the documents in the collection so that the bit-cost of storing the gaps decreases. The space required for

ADCS '21, December 9, 2021, Virtual Event, Australia

ACM ISBN 978-1-4503-9599-1/21/12…$15.00
https://doi.org/10.1145/3503516.3503528

any given term is minimized if the $d_{t,i} - d_{t,i-1}$ gaps are as non-uniform as possible; for a single term, optimal rearrangement is trivially achieved by assigning the documents containing that term to the permuted range 0 to $f_t - 1$, and the remaining documents to the range f_t to $N - 1$. But over the whole collection, the situation is more complex, since a global document reassignment is required that is the same for all terms.

To tackle that obstacle, Dhulipala et al. [5] developed an approach based on bipartite graph partitioning. Their BP mechanism splits the document number range into two halves, each of size $N/2$, and then identifies pairs of documents, one in each half, that if swapped will result in a net predicted reduction in index size. The estimation process is based on the two documents' terms' frequencies within the two halves. Swapping a pair of documents between the halves alters those frequencies for all terms that occur in the two documents, meaning that re-estimation must then occur too. The estimation and swapping process iterates until either stability (a fixed point) is achieved, or some maximum number of iterations is reached. The process then recursively considers the first $N/2$ documents, and then, separately, the second $N/2$ documents, further permuting the ordinal assignment within those two subranges, but not allowing any more document transfers across the primary $N/2$ boundary. The recursion ends when small ranges spanning ten or twenty documents are reached.

Mackenzie et al. [12, 14] explore variants of the BP approach, achieving consistent efficiency and compression improvements across a variety of document collections. Similarly, Wang and Suel [23] proposed a variant of the BP technique which improves the efficiency of conjunctive query processing. Bipartite partitioning has also been shown to accelerate query processing for disjunctive dynamic pruning algorithms [11, 18].

Partitioned Collections. Large-scale search systems are usually supported by a cluster of processors, each responsible for its share of the overall collection. The partitioning of documents across machines is usually random, to achieve effective load-balancing and efficient computation; with these attributes usually measured by query latency (perhaps as a mean or median, or perhaps at some high percentile as part of a service-level agreement or SLA [15]). Each machine in the cluster must comply with the SLA if the cluster as a whole is to be able to. To achieve that compliance, some minimum number of processing nodes, denoted P, must be employed. Each query that arrives is processed in parallel on all P machines, with their answer sets joined and then (perhaps) perturbed by subsequent detailed ranking phases. If fewer than P processing nodes are available, either it will not be possible to store all of the documents in the collection; or not possible to process queries quickly enough; or both.

Now consider one of those P machines and its allocated data subset. At any given point in time that machine is operating at some fraction of its available capacity, both in terms of disk space for documents and index, and also in terms of processing power and its ability to handle queries within the SLA. Moreover, from a cost-effectiveness (that is, commercial competitiveness) point of view, the goal is to handle the required data, and the current query load, not only within the SLA, but using as few resources as possible.

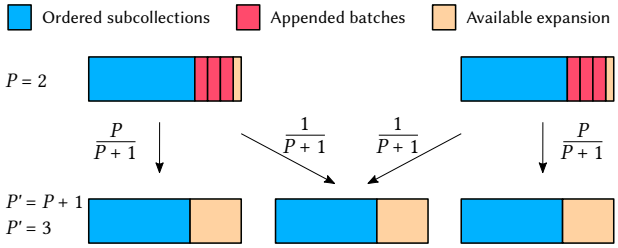

Figure 1: Restructuring after three batches of new documents have been appended, here with $P = 2$ and $P' = 3$, just before a thinning and reassignment step, and then just after. Both of the initial partitions are thinned by approximately 33% to allow creation of $P' = 3$ equal-sized new partitions, ready for further expansion. All three new partitions might then be fully reordered as part of the restructuring.

Extensible Collections. The discussion so far has assumed a static collection, and that an index is to be prepared for it as an off-line task. In an extensible collection documents arrive throughout the life of the index, and must be integrated into it as they arrive, so that they become findable shortly after their acquisition.

Taking all these constraints into account, we consider a strategy that involves repeated cycles of growth. In each cycle the starting configuration is an arrangement in which the collection is randomly partitioned across P machines. We then consider what must happen as the collection grows. In the short term, $1/P$ of any batch of new documents can be assigned to each processor. But eventually one or more additional machines needs to be assigned to the cluster, because – as noted above – the assumption is that each machine is near (but not beyond) its limit in terms of data storage ability (for example, a memory-resident index is limited by the amount of memory that is available), and near (but not yet breaching) its query throughput SLA expectation. For this purpose "near" is defined as "at or above z_0 percent of capacity", where z_0 is (perhaps, in an ideal situation) 75% or 80%. There will also be an upper bound $z_{\max}$ (perhaps 95% or 98%) which, when reached in terms of either storage or query load, triggers a restructure.

Figure 1 provides a schematic of this process. The main part (shown in blue) of both of the $P = 2$ initial partitions is a fully reordered collection. Batches of documents are then added (three batches are shown in red in the example), and both of the partitions expand, absorbing an equal random share of the new documents. Eventually the point is reached at which no more growth is possible; and additional resources must be added. At that point both of the existing partitions shed approximately 33% of their documents in order to establish a third partition. The new partition, or all three partitions, might be reordered. Querying then restarts, with all of the (now) P' partitions having expansion capacity available (the brown zones).

Restructuring. Hence, an extensible collection operates in cycles that start with a freshly balanced assignment of the collection across some number of machines P, with P chosen such that the index storage z_s at each machine satisfies $z_0 \le z_s \le z_{\max}$, and simultaneously the system load z_q satisfies $z_0 \le z_q \le z_{\max}$ at each

machine, with $z_0 \le z_{\max}(P-1)/P$ as a further constraint. Data growth is then handled by accumulating arriving documents into a batch, with a tailored "stop press" index used to resolve queries against the batch while it is building. Once either a batch of new documents of suitable size has been accumulated, or some pre-specified interval of time has passed, the batch is split randomly across the P machines, each of which must integrate its $1/P$ share of those documents.

This batch-at-a-time growth phase continues until the trigger point at which $z_s > z_{\max}$ or $z_q > z_{\max}$ is reached on one or more of the machines making up the cluster. Indeed, with random allocation it is likely that all of the machines in the cluster will approach these limits at roughly the same time. When that moment arrives, new hardware must be introduced, so that the collection can be redistributed across $P' > P$ machines.

In Figure 1, $P' = 3$ is shown relative to $P = 2$, but any value of $P' > P$ might be used. In particular, when P is large and still growing, too-frequent reorganizations will give rise to a high total workload. More useful (as is also done when deploying `realloc()` in memory for dynamic arrays) is for the new cluster size P' to be computed as $P' = \lceil G \cdot m \rceil$ for some constant $G \approx z_{\max}/z_0$; that is, as an integerized geometric sequence with a parameter chosen in terms of the overall system utilization targets.

Given that context, each machine in the cluster follows a cycle of activity that has five components.

Q[uerying]: It receives queries from the centralized control and resolves them against its current local document collection and index, returning top-k answer sets back to the control;

G[rowing]: From time to time it receives a batch of new documents, and must incorporate them into its local index;

M[onitoring]: It monitors its local storage utilization z_s and local query load z_q, and if either of them exceed $z_{\max}$, it reports that fact to the centralized control;

T[hinning]: It will then receive back an instruction to "thin by t%", a request that it select a random subset of t% of its current documents, remove them from the local index, and inform the centralized control of the culled set, so that they can be reassigned to one of the $P' - P$ new machines that will shortly be hosting the collection; and

R[eorganizing]: It then undertakes any necessary localized reorganization of its depleted local index, possibly including document reordering, prior to resuming query processing.

The first of these five components (**Q**) is already the subject of a great deal of experimentation; and the third step (**M**) is straightforward. It is the other three steps (**G**, **T**, and **R**) – and the interactions between them – that are the subject of this paper.

Related Work. We are not the first to consider implementation issues associated with extensible collections. For example, Busch et al. [3] discuss the Twitter "Earlybird" search engine where real-time ingestion and processing is very important. Mohammed et al. [20] explore a different aspect of growing collections, examining how the number of bits used when quantizing an index degrades quality over time for append-only updates. Other work has considered batched document updates [8], and a geometric scheme that cascades updates into larger and larger units in escalating transitions [9]. Büttcher and Clarke [4] also discuss sub-index merging. Moffat et al. [19] consider the compression of the documents themselves as a collection grows, exploring similar ideas of periodic complete reconstruction. None of those previous authors consider the interaction between document renumbering and collection extension.

Datasets and Methodology. Finally, to end this section, we describe the resources used through Sections 3 and 4, which consider how best to structure the index of an extensible collection; and then the consequences of those structural options in terms of querying throughput, including in cases where the query stream displays temporal drift.

The CC-News-En collection contains approximately 43 million English-language news documents taken from the Common Crawl, see Mackenzie et al. [13] for further details. The documents span the period August 2016 to March 2018, and each has its crawl date associated with it. We are thus able to simulate the behavior of a collection that grows by daily, weekly, or monthly batches of documents. In most of our experiments we start with documents acquired in the first half of the overall time period of the collection (approximately 40% of the documents) and then trace the acquisition of the other 60% of the collection, assuming them to arrive in date order in ten batches each corresponding to approximately one month of insertions, and each of a similar (but not equal) size. That is, our experiments cover one complete operational cycle as the collection more than doubles in size.

Index reordering is carried out with the open-source toolkit provided by Mackenzie et al. [14].[1]

We also focus on the operation of one processing node. The thinning operation can be simulated by removing a random t% of its documents; and insertions are enacted by appending a batch of documents. Measurement of a system as a whole, across multiple processing nodes, can then be inferred from the fact that the document assignment is random. Mackenzie et al. [15] describe experiments that support this claim.

3 INDEX THINNING AND EXPANSION

This section describes implementation options that might be employed in an extensible collection, and shows how they affect the compressibility and hence size of the resultant index. Section 4 then considers issues to do with query latency, and the effect that collection structure has on query throughput relative to a temporally-varying query stream.

Reorganization After Thinning. The first experiment explores the degree to which a newly-thinned collection remains compact. Recall that the thinning operation (**T**) takes an existing index, removes a random subset of t% of the documents, and then adjusts the index accordingly, so that future querying operations will not result in references to the removed documents. The thinned documents are transferred to one of the $P' - P$ newly added processing nodes, and responsibility for them no longer sits with this one.

There are three strategies that might be employed at each node, each of which might be attractive in a cost-effectiveness sense.

LG: In the "leave gaps" strategy the retained (that is, non-thinned) documents keep their ordinal numbers within this partition, and

[1] https://github.com/mpetri/faster-graph-bisection

the postings that referred to culled documents are removed from the index in a simple sequential pass. For each such removed posting the next subsequent posting's d-gap increases, to span past the removed document identifier, but the index should get smaller, although not by t%. Even when t is very small, the majority of the postings may need to be relocated, even if not recomputed and recoded; and when $t \approx 50\%$ a majority of postings will need to be recomputed and recoded prior to being rewritten. Note that this option leaves the partition with a sparse document domain. After the thinning operation it holds $N' < N$ documents, where $N' \approx (1-t) \cdot N$, but their labels still span close to the whole range from 0 to $N-1$, and so the average d-gap of postings will have increased.

PL: The second option is to "pack left" the document domain in this partition, so that the post-thinning ordinal document space spans 0 to $N'-1$. The benefit of doing this is that index compression effectiveness is more likely to be maintained. The retained documents are in the same order as in the **LG** approach, and in the same relative order as the initial configuration. The same number of postings must be relocated as in the **LG** approach, but they will also require recomputation and recoding as they are moved. A complete renumbering of the documents is also required, which may affect other data structures, for example, the mapping that converts a document number to the address from which that document can be retrieved, perhaps slightly adding to the cost.

RO: The third, and most expensive option, is to fully reorder the thinned collection using bipartite partitioning. The reduced document set is handed to the recursive partitioning process, and a completely fresh document ordering received back. A new index is then constructed based on that ordering.

To measure the implications of these three options, we took CC-News-En, and constructed a BP-ordered index for those documents, with the goal of exploring the extent to which compression might be eroded by the three approaches to reorganization once thinning had taken place. The results appear in Table 1. The three columns in each of the three groups correspond, respectively, to a surrogate for compressibility, defined for postings list I_t as $\sum_{0 \le i < f_t} \log_2(d_{t,i} - d_{t,i-1})$, then summed over all postings lists and divided by the total number of postings (loggap, with units of bits per posting d-gap); the measured average size of the compressed d-gaps using binary interpolative coding (bpp, compressed bits per posting d-gap); and the actual index size in GiB. The latter includes the $f_{t,i}$ values as well as the d-gaps, and also a number of other small overheads, such as alignment bytes.

As anticipated, the **LG** strategy is demonstrably worse for all non-trivial values of t. More interesting in the table is that the **PL** and **RO** approaches provide comparable effectiveness right across the range of thinning ratios t. Applying the full BP process to a 50% randomly thinned collection yields the same net outcome as does retaining the non-thinned documents in their initial ordering. Indeed, if anything, **PL** has a slight advantage.

Finally, note that in this experiment it was assumed that the collection that was thinned was a fully BP-ordered one. In the context sketched in Figure 1, this is not appropriate – in the schematic, the collections being thinned are composite, consisting of a main part that has been properly reordered, and then smaller batches that have been appended. That difference is examined next.

Index Expansion and Regrowth. The complement operation is that of growing the index – adding multiple batches of documents in such a way that the index is queryable after each such addition, and retains its structure and "efficiency" characteristics while those additions are taking place. For example, the initial configuration depicted in Figure 1 supposes that three batches of fresh documents have been added to the (blue) partitions that resulted from the previous thinning step.

Given that context, the methodology employed to compare different approaches is summarized as:

- the chronologically first 40% of CC-News-En is taken to be a starting point, and an ordered index built for it;
- the remaining documents are also ordered chronologically, and broken into ten batches each of approximately 6% of the CC-News-En collection, and with approximately one month of new documents in each of those ten batches;
- each batch is appended in some form to the current index, and statistics computed.

We then considered four possible approaches to incorporating each batch into the growing index.

SA: In the "simple append" approach the documents in the batch are retained in the order they were acquired, that is, chronological by date.

BR: In the "batch reordering" mechanism the documents in the batch are reordered based only on information provided in the batch, and then appended to the original index. Two variants of this approach were considered: one where the ordering applied to the batch was based on source page URL; and a second in which bipartite partitioning was applied to the batch.

GR: The third approach is "global reordering", in which the batch and the previous index are combined, and then completely reordered via an application of bipartite partitioning.

The **GR** approach provides the reference point for index space, since the final document ordering is independent of arrival order.

Note also that there was no particular operational reason for taking batches to be one month, and we could instead have used one week, one day, or one hour. However using one month or one week per batch yields measurements that are less likely to exhibit local volatility, making the overall trends easier to discern.

Figure 2 uses the methodology described above to compare the four strategies. The three graph panes show the effect of growth strategy on three different aspects of the per-posting cost of storing the index, as described in the figure caption. Horizontal lines indicate strict linear growth in size relative to the starting point, and increasing lines reflect super-linear index cost growth. Fully reorganizing the index (the **RO** approach) after each batch arrives yields the most stable compression performance, and as noted above, represents the ideal situation. But – as is demonstrated in the next set of results – it also incurs significant computational cost.

Total Cycle Cost. Table 2 summarizes the difference between the **BR** and **GR** approaches to reordering. In the table's first section, a full reordering of 40% of CC-News-En is assumed, immediately

Table 1: Reorganization options applied to CC-News-En. The first row provides a reference point, and corresponds to a BP-ordered index for CC-News-En. The remaining rows demonstrate the outcome on index size and compression when thinning by t%. The "loggaps" (sum of the binary logarithms of the d-gaps) and bits per posting values are computed on the document identifiers only; in the third column in each group, index size in GiB covers all index components, including within-document frequencies and a number of overhead components. The integer d-gap lists and $f_{t,i}$ frequency values are compressed using binary interpolative coding in all cases.

t	LG strategy			PL strategy			RO strategy		
	loggap	bpp	GiB	loggap	bpp	GiB	loggap	bpp	GiB
0	1.28	2.96	10.06	1.28	2.96	10.06	1.28	2.96	10.06
1	1.29	3.00	10.06	1.28	2.96	9.97	1.28	2.96	9.96
2	1.30	3.05	10.06	1.28	2.96	9.87	1.28	2.96	9.86
5	1.34	3.16	9.99	1.29	2.97	9.59	1.29	2.97	9.58
10	1.41	3.34	9.82	1.29	2.98	9.11	1.29	2.98	9.11
20	1.55	3.67	9.32	1.31	3.01	8.15	1.32	3.02	8.18
50	2.16	4.74	7.03	1.37	3.12	5.24	1.40	3.15	5.28

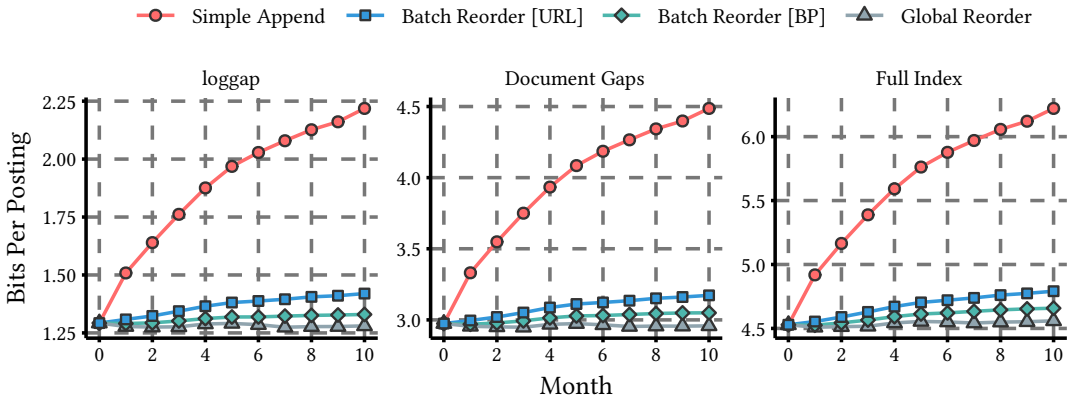

Figure 2: Compression effectiveness drift, measured as ten batches are added to an index for the first 40% of the documents in CC-News-En. As for Table 1, three measurements are provided: average loggap values in bits per d-gap across all terms (left); actual cost of storing the average d-gap using the binary interpolative code (center); and the complete cost of the index including the $f_{t,i}$ values, here reported as average bits per posting (right). All three graphs show the total index cost through to that point as ten batches of documents are appended and incorporated using different strategies. For example, the values at 5 on the horizontal axis correspond to approximately 70% of the CC-News-En collection. Note that the vertical scales in the three panes are truncated to fit the corresponding data ranges.

after a thinning operation. Ten batches of updates are then processed, each of roughly the same size, each which is reordered independently and then appended to the existing index. The arrival of any of those batches might trigger the next cycle of thinning, but in this experiment we assume that does not occur. (Or it might be the upcoming eleventh batch that triggers thinning.) On average, each batch reordering requires 120.5 elapsed seconds on our test hardware (see Section 4 for a detailed description); yielding a combined total index reordering time of 2222 seconds. If URL ordering is used in conjunction with **BR** the time is halved: sorting a list of URLs takes negligible time, and only the 1017-second immediate-post-thinning reordering is required.

In contrast, if the **GR** approach is used, eleven complete reorderings are required, spanning (approximately) 40%, 46%, 52%, and so on, through to 94% and then 100% of the collection. As with the **BR** method, it is assumed that none of the ten batches triggers the next thinning cycle. The eleven full reorderings range from 1107 seconds (for 40% of the eventual collection) through to 3186 seconds (for the full 100% collection), and sum to the total shown in Table 2. That is, **GR** incurs a significant overhead compared to **BR**, an overhead that would be even higher if the batches were weekly or daily.

Note that in all of these calculations we are counting the cost of computing the reordering, but not the cost of carrying it out. While the cost of mechanically renumbering the documents remains consistent between strategies, the resultant distributions may impact the efficiency of compressing the postings lists – better orderings are likely to result in faster index construction. We leave a more thorough investigation to future work.

Summary. Our conclusion from these experiments is that effective index compression can be readily maintained in the face of repeated cycles of thinning and then batched growth using relatively straightforward strategies, and need not require regular expensive "full

Table 2: Time spent computing reorderings, comparing **BR** and **GR** and measuring elapsed CPU time for a multi-threaded implementation of bipartite partitioning. The calculation covers one full thinning-expansion cycle: starting with a newly thinned index, a bipartite reordering is performed; and then ten one-month document batches are added, taking the system through to the moment at which the next thinning cycle would start.

Processing mode employed	Time (sec)
BR operation (using BP):	
– reordering following straight after thinning	1017
– average cost per batch reorder	121
total time over whole cycle	2222
GR operation:	
– total time over sequence of ten full reorderings	22,557

Table 3: Query processing cost in milliseconds per query, computing the top-ranked $k = 10$ answers for each query, using the CC-News-En collection with various orderings. The base collection used in the first row has 17,167,810 documents; after all ten document batches have been added to it that count increases to 43,495,426 documents, or 2.5× bigger (the other four rows).

Method	Ranked AND			MaxScore		
	Mean	Median	P_{99}	Mean	Median	P_{99}
Base index	5.8	2.2	57.4	12.0	7.9	60.9
SA	20.9	7.5	197.1	39.3	33.4	159.2
BR [URL]	15.0	5.3	151.5	31.1	20.7	149.6
BR [BP]	14.0	4.9	145.4	31.0	20.0	151.5
GR	13.4	4.8	139.2	28.0	17.9	142.0

reordering" steps. That is, the batches can be treated independently, with full reordering only necessary at the processing nodes that are added at each thinning/redistribution step.

Moreover, the two **BR** options – batch reordering by document URL, and batch reordering via the application of bipartite partitioning within that batch – are much less computationally demanding than the **GR** approach, and yet are still capable of maintaining very good compression effectiveness in the index.

4 QUERY PROCESSING

This section switches focus, and explores a second important aspect of overall cost – query processing time. To measure query processing efficiency, we employ the CC-News-En queries and consider two representative processing tasks:

- Ranked conjunctive processing, in which the set of documents that contain every query term is identified, and then ordered by decreasing score computed by the BM25 similarity computation across all terms, with the top $k = 10$ documents identified and reported (method "Ranked AND"); and
- Ranked disjunctive processing, in which the set of documents that contain any of the query terms is identified and then ordered by decreasing BM25 score computed across the terms common to query and document, using the MaxScore pruning approach [22], with $k = 10$ documents again returned as the final answer (method "MaxScore").

All experiments were performed entirely in-memory on a Linux machine with two 3.50 GHz Intel Xeon Gold 6144 CPUs and 512 GiB of RAM, with query timings measured as the mean elapsed latency over three independent runs. Document collections were indexed using the Anserini [25] system with Porter stemming and stopping enabled. Those indexes were then converted to the PISA [17] format using the common index file format [10]. Postings lists were compressed using SIMD-BP128 [7]. All query timings were measured in the context of the PISA search system. Note that bipartite partitioning allows a high degree of parallelism, and that the implementation employed to obtain the results in Table 2 makes use of up to 32 processing threads. That is why we reported elapsed computation times there, rather than CPU-only times. The temporally relevant CC-News-En query log was used in our experimentation, containing 10,437 user query variations corresponding to 173 unique topics. Each topic relates directly to a single *target document*, allowing the log to be split into temporal batches; each month has around 500 queries corresponding to an average of nine topics.

Query Processing Latency. Table 3 shows measured per-query execution costs, listing the average, the median, and the 99 th percentile time, for two query processing modalities, and for collections of two different sizes. The first row gives times for the first 40% of CC-News-En, and serves as a reference point. The four remaining rows then give querying times for the four different batch-appending strategies, in all four cases for the full CC-News-En collection when constructed by adding ten batches of documents to the initial 40% collection; that is, by, in effect, more than doubling the initial collection via ten monthly updates.

As can be seen from the table, query processing time more than doubles as the collection doubles, lending credence to the earlier statement that both storage space and query load need to be monitored at each processor. Of the four growth strategies, **SA** is again a poor choice (in part simply because its index is bigger, see Figure 2), and **GR** is still the most economical choice, but with the two **BR** options both close behind. In these results all of the CC-News-En queries are executed against both the 40% and the full-collections.

Figure 3 shows the growth in query processing time as update batches are added to the base collection. In this figure the queries are also stratified, with time-based subsets of the query log formed. For example, at month "5", the queries used to measure the corresponding retrieval times were the subset that corresponded to that fifth update batch. (See Mackenzie et al. [13] for details of how the topics and the queries were developed.) The relatively small number of different topics available for each temporal update period is why a degree of volatility is evident. Even so, the curves confirm the overall results shown in Table 3. Similar relative behavior was also observed when the top $k = 1000$ documents were retrieved.

Temporal Topic Effects. The heatmap in Figure 4 explores the relationship between queries and update batches more closely. Each query segment was executed against the collection at the matching stage of its growth, as already described in connection with Figure 2,

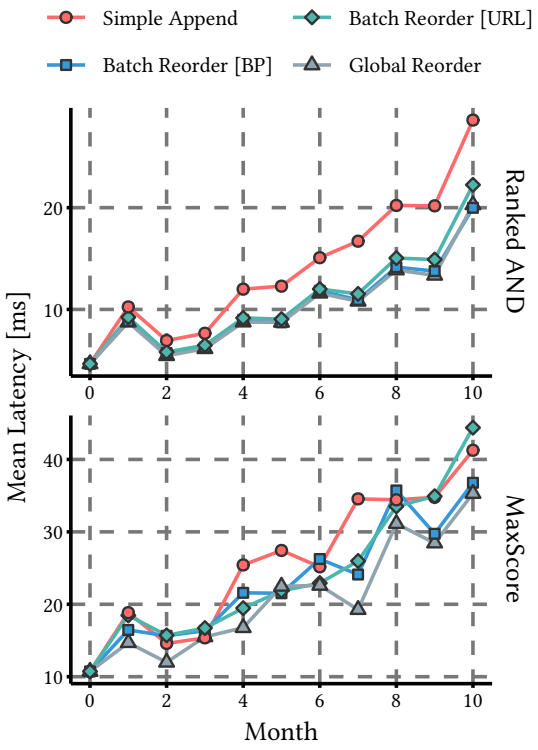

Figure 3: Mean query processing time, in milliseconds, with $k = 10$ answers retrieved for each query as the index grows over time. Note that only the queries corresponding to each batch are measured; the values shown cannot be compared to those in Table 3.

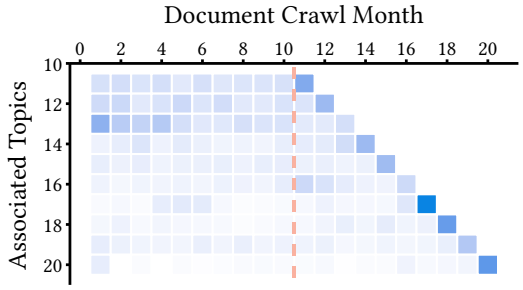

Figure 4: Fraction of top-1000 answer rankings occupied by documents of different batches, bucketed by their crawl date, and stratified across the corresponding batches of the temporal query log.

and the top $k = 1000$ documents for each query were identified. Those answer documents were then mapped back to their temporal positions in the development of the index, with the initial 40%-CC-News-En index also split into monthly acquisition batches. The total number of answers accruing from each index batch was then normalized against the document count for that batch, and plotted as an intensity level. The interesting aspect of this figure is the clear recency effect down the diagonal – queries tend to select documents from the corresponding time period. Note that the BM25 scoring regime used has no bias in favor of recency, and this pattern has arisen organically from the data.

We examined in detail the apparent anomaly that appears in month "13". The extra attention on the articles from the first batch – a little over a year prior to the documents out of which the queries were derived – was a result of the queries associated with two of the topics. One topic was related to a bombing in Yemen, the other to the price of EpiPens; both arose as "topics" associated with month 13 because in that month there were followup articles connected to the original clusters. Conversely, the strong correspondence between batch and month "17" is because of a dominance of timely events being queried by the topics used as seeds for the query solicitation process, one of which was an interview of Barack Obama on BBC radio hosted by Prince Harry, and another being the death of Rose Marie (an actor who featured in The Dick Van Dyke Show, popular on TV in the 1960s).

In future work we plan to explore this temporal locality and the effect it may have on query processing times. For example, it may be beneficial to process the most recently added index batch first, as a component of a non-sequential query processing regime of the type proposed by Mackenzie et al. [15]. West [24] has also considered temporal trends in query sequences, and trends.google.com allows these effects to be explored.

5 CONCLUSION

We have identified and examined a number of issues in connection with extensible retrieval systems, ones in which batches of documents are appended from time to time. We have described the operations that are required, and considered options applicable to each. The most compact indexes, and the fastest retrieval operations, arise if every batch of new documents is fully integrated into the previous index via a global reorganization step (**GR**), including a complete document reordering via the bipartite partitioning mechanism. But this option is computationally expensive; the cheaper **BR** batch-reordering options provide compression effectiveness that is almost as good, and support querying times that are only a little slower over the cycle of growth.

Having established this framework for describing extensible collections, we have a number of areas that we will examine next. Primary amongst these is to provide a better accounting of the relative expense of repeated full bipartite partitioning steps (for example, monthly, or weekly, or daily) compared to the CPU savings that accrue during querying, to establish balance points between document arrival rates and query arrival rates, using a "total cost of operation" model that converts hardware costs into monetary amounts for the purposes of comparison. We also plan to explore enhanced querying modalities that focus the initial evaluation effort on update batches that are likely to be temporally matched to the query, to accelerate the dynamic pruning algorithms that are integral to the MaxScore [22] and WAND-based approaches [2, 6, 16] to disjunctive query evaluation.

Acknowledgment. We are grateful to Matthias Petri (Amazon Alexa) for providing critical assistance as we were planning, executing, and in particular, finalizing this investigation. This work was funded by the Australian Research Council (project DP200103136).

REFERENCES

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

[2] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*, pages 426–434, 2003.

[3] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at Twitter. In *Proc. ICDE*, pages 1360–1369, 2012.

[4] S. Büttcher and C. L. A. Clarke. Indexing time vs. query time: Trade-offs in dynamic information retrieval systems. In *Proc. CIKM*, pages 317–318, 2005.

[5] 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.

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

[7] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Soft. Prac. & Exp.*, 41(1):1–29, 2015.

[8] N. Lester, J. Zobel, and H. E. Williams. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. In *Proc. Aust. Comp. Sc. Conf.*, pages 15–23, 2004.

[9] N. Lester, A. Moffat, and J. Zobel. Efficient online index construction for text databases. *ACM Trans. Data. Sys.*, 33(3):3.1–3.33, 2008.

[10] 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.

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

[12] 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.

[13] 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.

[14] J. Mackenzie, M. Petri, and A. Moffat. Faster index reordering with bipartite graph partitioning. In *Proc. SIGIR*, pages 1910–1914, 2021.

[15] J. Mackenzie, M. Petri, and A. Moffat. Anytime ranking on document-ordered indexes. *ACM Trans. Inf. Sys.*, 40(1):13:1–13:32, Jan. 2022. To appear.

[16] A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini. Faster BlockMax WAND with variable-sized blocks. In *Proc. SIGIR*, pages 625–634, 2017.

[17] A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel. PISA: Performant indexes and search for academia. In *Proc. OSIRRC at SIGIR 2019*, pages 50–56, 2019.

[18] 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.

[19] A. Moffat, J. Zobel, and N. Sharman. Text compression for dynamic document databases. *IEEE Trans. Know. Data Eng.*, 9(2):302–313, 1997.

[20] S. Mohammed, M. Crane, and J. Lin. Quantization in append-only collections. In *Proc. ICTIR*, pages 265–268, 2017.

[21] G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM Comp. Surv.*, 53(6):125:1–125:36, 2021.

[22] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Inf. Proc. & Man.*, 31(6):831–850, 1995.

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

[24] R. West. Calibration of Google trends time series. In *Proc. CIKM*, pages 2257–2260, 2020.

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

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