Index-Based Batch Query Processing Revisited

Joel Mackenzie[1][0000−0001−7992−4633] and Alistair Moffat[2][0000−0002−6638−0232]

[1] The University of Queensland, Australia
[2] The University of Melbourne, Australia
joel.mackenzie@uq.edu.au,ammoffat@unimelb.edu.au

Abstract. Large scale web search engines provide sub-second response times to interactive user queries. However, not all search traffic arises interactively – cache updates, internal testing and prototyping, generation of training data, and web mining tasks all contribute to the workload of a typical search service. If these non-interactive query components are collected together and processed as a batch, the overall execution cost of query processing can be significantly reduced. In this reproducibility study, we revisit query batching in the context of large-scale conjunctive processing over inverted indexes, considering both on-disk and in-memory index arrangements. Our exploration first verifies the results reported in the reference work [Ding et al., WSDM 2011], and then provides novel approaches for batch processing which give rise to better time–space trade-offs than have been previously achieved.

Keywords: Batch query processing · inverted indexes · experimentation

1 Introduction

Batch processing is a general paradigm aimed at reducing computational costs by avoiding repeated or redundant computation when processing a sequence of related tasks. This idea has been explored in a range of contexts, including in relational database management systems [32], spatial-textual databases [10], caching arrangements [1, 18], and within information retrieval indexing and querying systems [4, 7, 16, 29]. Given the immense scale of commercial IR and web search systems, revisiting batch processing may be one way to achieve reductions in computational overhead, and consequently, energy consumption and carbon emissions [11, 31, 33].

In this work, we revisit the problem of batch processing over *inverted indexes.* Given a set of queries $\mathcal{Q}$, the goal is to process each of those queries minimizing the total cost according to some metric such as time taken or volume of data read. That is done by building a *query execution plan* for $\mathcal{Q}$ that involves techniques such as query reordering and partial preliminary computation of shared results into a temporary cache, so as to reduce the overall cost. In this environment individual query latency is unimportant, and it is the aggregate cost that is of interest. Batch processing over inverted indexes is motivated by a range of scenarios in which query traffic is not required to be processed under stringent service-level agreements [19], including refreshing caches [6, 17, 20, 22, 24],

internal mining and analytics tasks, collecting training data, and queries from external parties.

Despite the many possible applications of batch processing in information retrieval systems, there has been only limited experimental investigation and algorithmic development. The primary resource in this regard is the 2011 work of Ding et al. [16], who took the queries in $\mathcal{Q}$ to be sets of terms handled via Boolean conjunctions. It is reproduction of their work that forms the basis of the first experiments described here. In particular, we confirm that strategic pre-filling of a given volume of cache with postings lists results in overall savings of data transfer volumes; or, if the same amount of cache holds pre-computed list intersections, allows execution time reductions.

We then develop a new technique for the same task. Instead of selecting content with which to fill a static cache, we reorder the queries, paying careful attention to common subexpressions that can be evaluated once, at the time they are required, reused through multiple queries, and then discarded. The signal benefit of this approach is that only one intermediate list is required at any given time. With careful choice of intermediate results and a new cost estimation heuristic, we are able to outperform the Ding et al. [16] approaches using only a fraction of the cache that their methods require.

2 Background and Related Work

We first describe the problem that is considered, and then the techniques for addressing that problem that have been reported in the paper that is the basis for this reprodicibility study [16].

Definitions and Terminology. We suppose that $\mathcal{Q}$ contains n distinct queries, $\mathcal{Q} = \{Q_i \mid 1 \le i \le n\}$, which must each be resolved against a document collection $\mathcal{D}$. Each of those queries consists of a set of $q_i = |Q_i|$ distinct terms; lower-case alphabetic letters a, b, and so on from the beginning of the alphabet will be used to represent specific individual terms. The *vocabulary* V of $\mathcal{Q}$ is the complete set of all of $\mathcal{Q}$'s query terms, $V = \cup_{i=1}^{n} Q_i$. Each term $a \in V$ has an associated collection frequency $f_a \ge 0$, the number of documents in $\mathcal{D}$ that contain a, which also represents the length of the *postings list* $L(a)$ for a that records the identifiers in $\mathcal{D}$ of those f_a documents. The subset of queries that contain $a \in V$ is denoted by $T(a) = \{Q_i \in \mathcal{Q} \mid a \in Q_i\}$. That latter definition is also extended to term pairs: if $a, b \in V$, then $T(a, b)$ is the subset of $\mathcal{Q}$ in which both a and b occur as query terms.

If $L(a)$ and $L(b)$ are the postings lists for terms $a, b \in V$ with $f_a \le f_b$ then their *intersection list* $L(a, b)$ can be computed using $\omega(f_a, f_b)$ steps of computation and in $O(\omega(f_a, f_b))$ time, where $\omega(x, y) = x \log_2(1 + y/x)$. Algorithms for intersecting lists within these bounds are explored elsewhere [14]. Denote the minimum collection frequency of query Q_i by $\mu_i = \min\{f_a \mid a \in Q_i\}$. Then if $Q_i = \{a, b, c, \ldots\}$ is interpreted as a *conjunctive Boolean bag of words query* the output required is the set of at most μ_i postings that result when all of Q_i's

terms are intersected, $L(Q_i) = L(a) \cap L(b) \cap L(c) \cdots$. When we give example queries we will always list terms in increasing collection frequency. For example, in $Q_i = \{a, b, c\}$, we assume $\mu_i = f_a \le f_b \le f_c$.

In the *set-versus-set* approach to multi-way intersection the shortest list – of length μ_i – is used as a starting point, and then each other list is in turn intersected against that reducing set of candidates. If $C_0(Q_i)$ is the cost of evaluating the conjunctive query Q_i starting from the terms' individual postings lists, then $C_0(Q_i) \le \sum_{a \in Q_i} \omega(\mu_i, f_a)$. Hence, if $C(\mathcal{Q})$ is the total cost of evaluating all of the queries in $\mathcal{Q}$, with no shared processing between queries and each query evaluated in isolation, then $C(\mathcal{Q}) \le \sum_{1 \le i \le n} C_0(Q_i)$.

In addition to that computational cost there is also the cost of bringing the required lists from secondary storage into memory. A query Q_i of q_i terms requires that q_i transfer operations be initiated (seeks), and that a total of $D_0(\mathcal{Q}) = \sum_{a \in Q_i} f_a$ postings be transferred. Note that these transfer costs might also apply, albeit to a lesser degree, when a fully in-memory index is employed, with typical hardware configurations having multiple levels of memory.

Ding et al. [16] sought to explore the data transfer and computational cost benefits achievable via the complementary techniques of *list caching* and *intersection caching*, focusing on the context that has been described here – a set $\mathcal{Q}$ of queries which can be handled holistically rather than individually. Our goal in this work is to reproduce those results, and then to also extend them.

List Caching. Assume first that the system's inverted index is stored on disk, but that a known amount of main memory is available to temporarily retain selected postings lists so that they are available to future queries without seek or transfer operations being required. In the case of interactive querying, various strategies have been developed for estimating how best to employ the available memory. These are embodied as decision protocols as to whether or not a newly transferred postings should be retained; and if the decision is to retain, what list(s) to eject from the current cache so as to make space for this new entry.

In the case of non-interactive query batches it is possible to greatly assist list caching, because the queries in the batch can be evaluated in any order, and because the decision protocols can be based on *clairvoyant* knowledge [3]. For example, if a term appears only once in $\mathcal{Q}$, it should not be cached.

Query Orderings. One way in which a query set might be reordered is to first sort the terms of each query (for example, alphabetically by query term) and then sort the set of queries lexicographically [16]. This approach brings all queries with the same lexicographically least term into a consecutive group, meaning that an effective caching strategy – including the clairvoyant approach – will recognize that list and thus result in seek and transfer savings.

Ding et al. [16] also consider a second *clustering* approach to query ordering, noting the proposal of Cheng et al. [9], who study the problem of document identifier reordering for inverted indexes. In particular, Cheng et al. propose a partition-based reordering method which recursively clusters an inverted index by considering postings lists in decreasing order of their total size. Translating

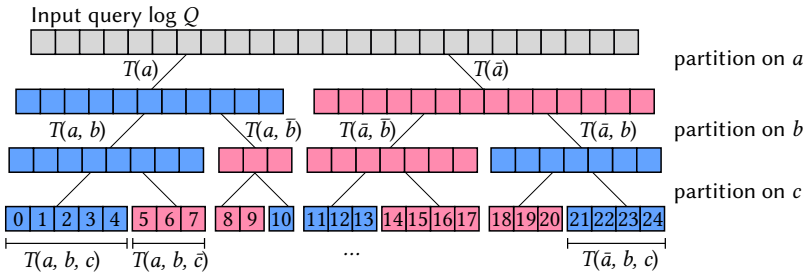

Fig. 1: Recursive Gray code-based query reordering. In this simple example terms a, b, c are ordered $|T(a)| \geq |T(b)| \geq |T(c)| \cdots$, and $T(\bar{a})$ represents the absence of term a, that is, $T(\bar{a}) = \mathcal{Q} \setminus T(a)$.

this to the set $\mathcal{Q}$, we first identify the queries in which each term $a \in V$ occurs, to establish the set $T(a)$. Then, considering those sets in decreasing size order, $\mathcal{Q}$ is recursively partitioned so that all of the queries (within each current partition of $\mathcal{Q}$) containing the next most queried term are contiguous. Each cluster is reversed relative to the previous level of the recursion, so that the cluster labels form a Gray code. Figure 1 illustrates this approach, denoted here as Partitioned.

Intersection Caching. While list caching techniques have the potential to reduce the amount of data transferred from secondary storage, they do not alter the computation required during intersection operations. Suppose now that some pair of terms $a, b \in V$ has had their joint list pre-computed, and that a query $Q_i \in T(a, b)$ is to be resolved, with $f_a \leq f_b$, and hence $|L(a, b)| \leq |L(a)| \leq |L(b)|$. Rather than using $L(a)$ and $L(b)$ in the intersection pipeline, this query should be resolved using the list $L(a, b)$. If $C_{a,b}(Q_i)$ is the computational cost of doing so, then $C_{a,b}(Q_i) \leq C_0(Q_i) - \omega(\mu_i, f_b)$, with the intersection using $L(a, b)$ requiring at most the same time as the original intersection using $L(a)$, but likely less given the avoidance of list decompression operations.

Recall that $T(a, b)$ is the subset of queries in $\mathcal{Q}$ that contain both term a and term b. An estimate of the net saving that might be achieved by precomputing $L(a, b)$ is thus given by $\sum_{Q_i \in T(a,b)} \omega(\mu_i, f_b)$, a benefit which must be debited by $\omega(f_a, f_b)$, the cost of carrying out the pre-computation. Moreover, the list $L(a, b)$ must be stored, and will require as many as $|L(a, b)| \leq f_a$ words of memory, or a fractional equivalent if stored in compressed form.

Term Pair Popularity. This then leads to a first approach to term pair caching: all co-occurring term pairs are tabulated to form the sets $T(a, b)$; the quantity $G(a, b) = -\omega(f_a, f_b) + \sum_{Q_i \in T(a,b)} \omega(\mu_i, f_b)$ is computed for each possible pair a, b, to compute the maximum gain possible by precomputing that pair, and then those values are sorted into decreasing order, to create a static popularity-based ordering of term pairs. Then, assuming that a specified amount of cache is available, pairs a, b are taken from that ordered list and given cache

Algorithm 1 Dynamic bang per byte query planning.

```
    for each pair a, b ∈ V do
2:      construct the list T(a, b) and compute B4B(a, b)
        add a, b to a priority queue PQ of pending term pairs
4:  while |PQ| > 0 and the cache limit has not been reached do
        select a, b from PQ, maximizing B4B(a, b), and exiting if B4B(a, b) ≤ 0
6:      allocate f_a units of cache to the pair a, b
        for each query Q_j ∈ T(a, b) do
8:          for each term c ∈ Q_j, with c ≠ a, b do
                adjust B4B(a, c) (or B4B(c, a), if f_c < f_a) to remove Q_j's contribution
10:             adjust B4B(b, c) (or B4B(c, b), if f_c < f_b) to remove Q_j's contribution
        remove a, b from PQ
```

allocations, iterating until the given cache limit has been reached. Term pairs for which $G(a, b) \leq 0$ should never be added to the cache.

The query processing plan fetches the postings lists for each of the selected pairs and intersects them to populate the cache; and then processes $\mathcal{Q}$ in any order, checking each Q_i for the appearance of any pairs a, b for which $L(a, b)$ is in the cache, and using pre-computed lists whenever possible. The cache of intersected lists remains constant through that execution sequence.

Static Bang Per Byte. A drawback of the simple popularity-based approach is that high-frequency pairs and low-frequency pairs might have the same value for $G(\cdot, \cdot)$, but a high-frequency pair will consume more cache and thus represent less net value. A second option is thus to normalize the projected gain according to the anticipated storage cost for the intersected list. That is, the list of term pairs is considered instead in decreasing order of $B4B(a, b) = G(a, b)/f_a$.

Dynamic Bang Per Byte. A further refinement is to note that a query Q_i might contain multiple pairs that received high $B4B(a, b)$ values, but that interactions between term pairs means that their independent gains cannot all be achieved. For example, if $Q_i = \{a, b, c\}$ and pairs a, c and b, c are both in the cache, then only one gain of $\omega(\mu_i, f_c)$ can result, making the other illusory. That consideration leads to the mechanism described in Algorithm 1, which is our interpretation of what is described in Section 4.2 (page 142, right column) of Ding et al. [16]. A priority queue PQ of pairs not yet placed in the cache is employed, with priorities given by evolving $B4B(a, b)$ values. The weights of pairs are non-increasing, meaning that a "lazy" queue can be employed (rather than an "always up to date" heap), and recomputations of $B4B(c, d)$ deferred until c, d reaches the head of PQ. A flag bit associated with each pair c, d is sufficient to record whether the current $B4B(c, d)$ value is valid or requires recomputation (and possible deferral) should it reach the head of PQ.

This mechanism employs the same query processing plan as the previous two, but should result in a more economical combination of lists $L(a, b)$ in the cache.

On the other hand, the computation described in Algorithm 1 is more complex than is calculation of static $B4B(\cdot,\cdot)$ values.

More Than Just Pairs. The ideas presented above are not limited to term pairs, and intersecting further terms (such as triples or quads) may also lead to improvements. However, Ding et al. [16] found that, due to the power-law distribution of query terms [30], selected pairs were also often overlapping with promising triples and quads, meaning that the pairs provide most of the overall benefit. Furthermore, increasing the number of candidates increases pre-processing costs. In this work, we only consider term pairs, noting that our batch processing framework can be extended to triples and beyond if required.

Other Related Work. Chaudhuri et al. [8] examined how materializing list intersections can reduce querying based on the power-law characteristics of corpus terms. Tolosa et al. [34] have also examined the benefits of caching intersections for in-memory processing scenarios, comparing a number of cache admission strategies. The broad problem of efficiently handling queries over large volumes of data has challenged researchers and practitioners for decades, with ongoing research; see Tonellotto et al. [35] for an overview.

3 Experimental Setup

Hardware and Measurement. Our experiments are conducted on a Linux server with two Intel Xeon Gold 6144 CPUs at 3.5 GHz with 512 GiB of RAM. Only a single processing core is utilized, allowing the use of processing latency as a proxy for total computing cost. The list caching experiments in Section 4 measure the volume of compressed postings lists needing to be transferred from secondary storage. Then the intersection caching experiments in Section 5 load the whole index into main memory prior to commencement of the query processing plan, and are reported as query batch execution times in elapsed seconds.

Software. One setback in reproducing the reference work is that the previous source code and experimental framework are not available. Thus, we reimplemented the algorithms based on the descriptions provided by Ding et al. [16]. All caching and query planning algorithms were implemented in C++ and compiled with GCC 7.5.0 using `-O3` optimization. Our retrieval experiments make use of a version of the efficient PISA search system [28], modified to allow the results of intersections to be stored and used during query processing. Document indexes are built using the Lucene-based Anserini system [36] and are converted to SIMD-BP128 [21] compressed PISA indexes via the common index file format [23]. Indexes are reordered using the recursive partitioning mechanism [15, 27].

Document Collections. Two public collections are used: MSMARCO-v1 and MSMARCO-v2 contain 8.8 million and 138.4 million passages respectively, drawn from English web documents [2, 13]. The *augmented* version of MSMARCO-v2 is adopted, in which passages are expanded with the document URL, title, and headings [25]. Table 1 provides statistics of the collections after indexing.

Table 1: Index statistics for the two test collections used here, and (last row) the collection employed in the reference work by Ding et al. [16], which is not publicly available.

Collection	Documents	Unique terms	Postings	Size [GiB]
MSMARCO-v1	8,841,823	2,660,824	266,247,718	0.9
MSMARCO-v2	138,364,198	16,579,071	8,629,430,400	22.8
RandomWeb [16]	10,000,000	–	–	4.2

Query Batch. Ding et al. [16] used a total of 1.16 million distinct queries from the Excite query log in their experiments. However, these queries are not likely to be temporally relevant to the MSMARCO passages used in our experimentation. Instead, we use the entire set of 10 million ORCAS queries [12]. These queries were filtered by first applying a normalization process: stemming, case-folding, and stopword removal according to the default Lucene tokenizer. All within-query duplicate terms were then removed, since we only focus on conjunctive matching. Finally, only unique queries which did not contain out-of-vocabulary terms on both MSMARCO-v1 and MSMARCO-v2 were collected into the final batch, resulting in a total of 6,761,892 queries with an average length of 3.2 unique terms.

4 Reducing Data Transfer Volume Via List Caching

The first experiment examines the gains possible as a result of in-advance knowledge of the query batch, assuming that the index resides on some form of secondary storage. In this context, data transfer time is an important factor in overall query processing time, and the aim is reduce it via list caching. Ding et al. [16] consider two factors for doing that: the order in which the queries should be processed; and the cache eviction strategy.

Query Ordering. As discussed in Section 2, the *order* in which queries are processed can improve term locality. Here, we evaluate three strategies:

- Random: A baseline measurement undertaken using a randomly shuffled log. (Ding et al. [16] started with the *natural* ordering of their query log, but we have no notion of a natural order here.)
- Sorted: Sorts the queries lexicographically, as described in Section 2.
- Partitioned: Employs the Gray code-based query reordering shown in Figure 1. We believe that this corresponds to the "agglomerative clustering" of Ding et al. [16] which is based on the work of Cheng et al. [9, Figure 7].

Ding et al. [16] comment that other clustering methods are possible. To explore that option we also implemented an approximate traveling salesman-based method [9, Figure 4] denoted TSP, which greedily traverses an induced graph

of queries, maximizing the similarity between the current query Q_i and all other unvisited neighboring queries Q_j, where neighboring queries are those sharing at least one term with Q_i. Query-to-query similarity is measured as $S(Q_i, Q_j) = \sum_{a \in Q_i \cap Q_j} |L'(a)|$, where $|L'(a)|$ is the length in bytes of the compressed postings list $L'(a)$ for term a, the goal being to maximize the retained posting volume at each query transition. Other similarities were also explored, including ones based on term overlap, but this formulation gave the best results. As an aside, both the TSP and Partitioned algorithms are employed by Cheng et al. [9], and in future work we will also apply these algorithms to document identifier reassignment [5] as a secondary reproducibility effort [26].

Cache Eviction. The second factor explored by Ding et al. was that of clairvoyant cache management. Since the whole query log is known, the cache can always evict the item not required for the longest, denoted here as CV, thereby allowing optimal behavior [3]. The *least recently used* (LRU) policy is used as a baseline mechanism.

Reproducibility Results. In the original evaluation [16] the cache size is measured in *millions* (without units); we assume those to be *millions of uncompressed postings*. Here we assume that cached lists will remain compressed and be decoded on-demand, so as to maximize the number of retained lists, and set the cache size to a sequence of fixed percentages of the original compressed index size. We then count the volume of compressed postings transferred from secondary storage through the whole of $\mathcal{Q}$. Figure 2 shows all eight combinations of log order and cache strategy for the two collections when measured this way.

The patterns of behavior observed in Figure 2 are consistent with those visible in Figure 1 of Ding et al. [16] (once the logarithmic vertical scale in Figure 2 is allowed for). In particular: the Sorted logs outperform the random (baseline) orderings; the recursively partitioned "Gray clustered" logs perform even better than the sorted logs; and in the second aspect of the experimentation, (and completely unsurprisingly, no foresight required) the clairvoyant caching approach results in less data being transferred than does the LRU strategy. Figure 2 also includes the approximate TSP-based query reordering technique, a second form of clustering; it outperforms the Gray code-based reordering process. However, the TSP ordering takes around 55 minutes to compute, whereas the Gray-code partitioning takes only 10 seconds, meaning that the slight reduction in data transferred via TSP is unlikely to be of interest.

5 Reducing Computation Via Intersection Caching

We now turn from list caching to intersection caching, seeking to confirm the outcomes reported by Ding et al. [16], again using the two MSMARCO collections.

Methods Tested. Ding et al. add materialized term pairs to a fixed-size cache, seeking to maximize the computational benefit achieved via each fixed volume

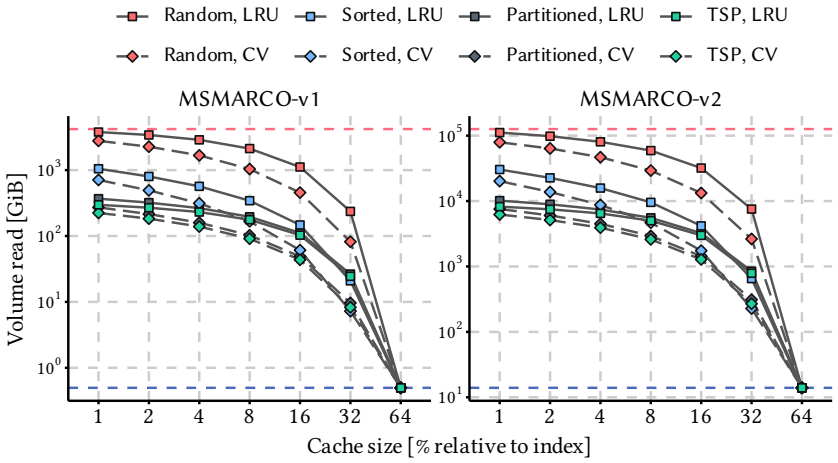

Fig. 2: Index volume transferred by LRU and CV caching for the four different orderings of $\mathcal{Q}$, with cache size set as a percentage of the compressed index size. The orange and blue dashed lines represent the worst-case (read a list from disk each time it is requested) and best-case (read each list just once) I/O performance, respectively.

of pre-computed term intersections, with the computation cost of a query Q_i estimated as $C(Q_i) = \sum_{a \in Q_i} f_a$, which is less precise than the $\omega(f_a, f_b)$ estimator presented in Section 2. They then construct an ordering of term pairs (described in their Section 4.2) using the mechanism presented in Algorithm 1, which is denoted here as Flexible-B4B. Ding et al. report average query execution speed of approximately 6.8 millisec per query (presumed to be for conjunctive Boolean queries of the kind we are also measuring here, as measured off a highly magnified screenshot of their Figure 8, with the computed percentage savings corroborated by their Figure 11); reducing to 5.0 millisec on average using a cache holding 8% of the index terms' original posting, a 26% saving; and further decreasing to approximately 4.6 millisec with a 30% term pair cache, a 33% saving.

We re-implemented the Flexible-B4B scheme, and also tested the static bang-for-byte term-pair selection process (Static-B4B), and an even simpler mechanism based purely on $G(a, b)$, denoted Popularity. Both of these approaches are also described in Section 2. In each experiment all of the postings lists were first loaded into memory; then the timing commenced; then all of the planned pre-intersections computed to populate the cache of term pair intersections; then the query batch was executed in query-sorted order (Sorted in the context of Figure 2); and finally, the timing was ended.

Reproducibility Results. Table 2 shows that the broad relationship between the three approaches can be confirmed. Stepping across each row, increasing the detail in the "merit estimation" process leads to decreased running times; within each column, increasing amounts of cache memory also lead to decreased

Table 2: Reproduced results. Total elapsed time (seconds) to execute the entire batch of queries, as a function of the volume of pre-computed intersections. Query plan generation times were all a few seconds each and are not included.

Space (%)	Popularity		Static-B4B		Flexible-B4B	
	MSM-v1	MSM-v2	MSM-v1	MSM-v2	MSM-v1	MSM-v2
0	735	18,864	735	18,864	735	18,864
1	714	18,686	701	18,523	686	18,333
2	713	18,701	684	18,291	663	18,143
4	702	17,970	661	18,113	628	17,523
8	678	18,417	622	17,606	575	16,750
16	646	18,048	575	16,880	511	15,461
32	597	17,319	516	15,956	447	13,838
64	536	16,291	453	14,598	406	12,547

execution times. Moreover, the gains achieved by the Flexible-B4B approach with a 32% space overhead of 39% for MSMARCO-v1 and 27% for MSMARCO-v2 match the gains achieved by Ding et al. [16], summarized above. Note that we are measuring cache size here as byte-based percentages of the compressed index size, and storing uncompressed intersection lists; whereas Ding et al. measure percentages of posting counts. While these are similar scales, the latter is likely to allow more postings to be stored at each percentage measurement point.

The average query execution times for the Flexible-B4B method with 32% cache are 66.1 *micro*sec (MSMARCO-v1) and 2.05 millisec (MSMARCO-v2), with the speed-up relative to Ding et al. likely the result of hardware relativities since 2011. Note also that MSMARCO-v2 is a much larger collection (Table 1).

Strategic Pair-Based Query Reordering. We now introduce a different way in which batches of queries can be expedited. Key to the new proposal is the observation that most queries are relatively short (3.2 terms on average in our test set) and hence having even one cached term pair per query represents a worthwhile target. That insight allows us to reorient the quest from being a search for the best mix of term pairs across the whole set of queries, to a query-by-query search for a good term pair. We call that connection from a query to a single term pair an *association*; and Algorithm 2 describes the process employed to identify them. An initialization phase determines which pairs $a, b \in V$ occur at least two of the queries in $\mathcal{Q}$. Three further processing phases then occur.

In Phase 1, steps 3 to 9, a best pair a', b' is tentatively associated with each query, and at the same time an estimate of the savings that might accrue is made, the latter via the computation at step 10, and credited to the associated pair using accumulator $E(a', b')$. Important new aspects of the estimation are that only one pair is allowed to claim benefit, and that it further hedges the computation by supposing that any of the other *numpairs* available in Q_i might also be used, albeit at a reduced saving. Assuming those other options to be

Algorithm 2 Associating at most one term pair with each query.

for each pair $a, b \in V$ **do** ▷ Phase 0: initialization
2: set $E(a,b) \leftarrow 0$ and set $valid(a,b) \leftarrow |T(a,b)| > 1$
for each query $Q_i \in \mathcal{Q}$ **do** ▷ Phase 1: tentative associations
4: set $numpairs \leftarrow 0$
for each term pair $a, b \in Q_i$ such that $f_a \leq f_b$ **and** $valid(a,b)$ **do**
6: set $numpairs \leftarrow numpairs + 1$
record as a', b' the a, b pair that maximizes f_b/f_a for Q_i
8: **if** $numpairs > 0$ **then**
associate the pair a', b' with Q_i
10: set $E(a',b') \leftarrow E(a',b') + \omega(\mu_i, f_{b'})/numpairs$
for each pair $a, b \in V$ **do** ▷ Phase 2: pair pruning
12: **if** $E(a,b) < \omega(f_a, f_b)$ **then**
set $valid(a,b) \leftarrow$ **false**
14: repeat steps 3 to 9 ▷ Phase 3: consolidated associations

Table 3: Query processing using Algorithm 2. All results are in total seconds, and may be compared with those in Table 2. See the text for details.

Approach	MSM-v1	MSM-v2
Phase 1 only	463	13,138
Phases 1–3, but without discounting by *numpairs*	407	11,825
Phases 1–3, including the discounting at step 10	367	10,684

uniformly spread across the range from zero to $\omega(\mu_i, f_{b'})$ means that the marginal benefit of a', b' needs to be discounted by *numpairs*.

Phase 2, steps 11 to 13, then factors in the pre-computation cost, and removes from contention (represented as the set *valid*) any pairs found to not be of benefit. That leaves a reduced set of "known to be definitely useful" pairs available; they are used in Phase 3, step 14, which recomputes the associations. Queries previously associated with term pairs that got removed must be re-associated, with any new associations formed certain to be of net benefit. That is, in Phase 3 some queries have the tentative association confirmed, some queries get a revised association, and some queries lose their associations and join a pool of "immune" queries; with every such decision definitely decreasing the estimated execution time. Figure 3 gives an example that illustrates the way in which each query has exactly one associated term pair selected, with immune queries associated with the empty term pair at the top, and benefits gained via the selected edges.

The query processing plan is then simple: each *valid* term pair is computed, all of its associated queries are resolved, and then that intersection is discarded. Only a miniscule amount of cache storage is required – in our MSMARCO-v2 experiments, the peak space needed is just 41.5 million postings.

Table 3 shows the gains resulting from this new approach, with three rows of results: first, using the Phase 1 associations alone, without the pruning step

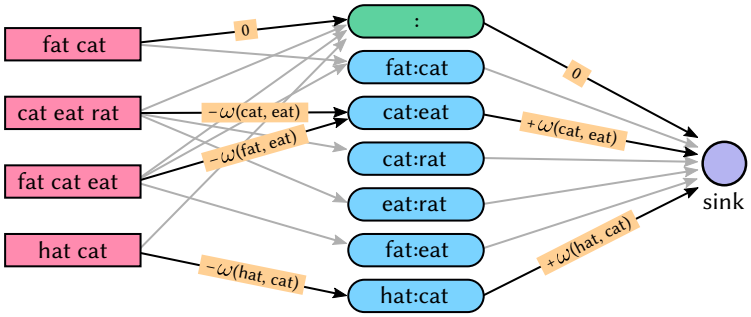

Fig. 3: Associations between queries and term pairs. Each query contains multiple term pairs, of which exactly one is selected (possibly the "empty pair"). Labeled edges represent the net saving of this possible set of associations.

Table 4: Popular term pairs in the context of the MSMARCO-v2 collection frequencies: the five with the greatest number of associations (left); and the five with the highest *match percentage* over the $|T(a,b)|$ queries containing a and b (right), where the match percentage in the rightmost column in each of the two groups is the ratio of assigned associations ("Assoc.") as a fraction of $|T(a,b)|$.

a	b	Assoc.	$\lvert T(a,b)\rvert$	%	a	b	Assoc.	$\lvert T(a,b)\rvert$	%
near	me	3,669	30,303	12.1	invent	who	1,042	1,430	72.9
icd	10	3,635	8,084	45.0	forecast	10	1,007	1,411	71.4
between	differ	3,447	7,534	45.8	fargo	well	2,610	3,734	69.9
mean	what	3,149	26,551	11.9	orlean	new	1,321	1,905	69.3
side	effect	2,977	9,632	30.9	depot	home	2,302	3,473	66.3

that is part of Phase 2; then without the discounting by *numpairs*, which is too optimistic in its estimations and retains pairs that are of marginal or negative benefit; and then, in the last row, using the mechanism that is described in Algorithm 2, including the discounting. Each variant takes less than 30 seconds to generate a plan. Not only does this approach allow batch query processing without the need for a large intersection cache, it is also notably faster than the Flexible-B4B method of Ding et al. [16], saving a further 20% and 23% of running time (MSMARCO-v1 and MSMARCO-v2 respectively) compared to the "32% cache space" row in Table 2.

Finally, Table 4 illustrates why Algorithm 2 is so effective. The important associations derived for MSMARCO-v2 do indeed correspond to term pairs that have natural relationships with each other. Note that all terms were stemmed, and that they are shown with $f_a \le f_b$ rather than in query appearance order.

6 Conclusion

We have reproduced the results presented by Ding et al. [16], and confirmed that list and intersection caching are effective techniques that can be applied when batches of conjunctive queries are to be processed in a non-interactive manner. In addition to having more precisely documented those methods, we have added a new "term pair association" mechanism to the repertoire. It allows even better batch execution times to be achieved, without requiring large volumes of cache. That means that list caching and strategic pair intersection planning can now be combined, to get the benefit of both enhancements. We hope that our reproducibility effort and public code resource will encourage further investigation into this area of research.

Acknowledgements This work was supported by the Australian Research Council's *Discovery Projects* Scheme (project DP200103136) and a University of Queensland New Staff Research Grant.

Software In the interest of reproducibility, our implementations are available at https://bitbucket.org/JMMackenzie/batch-conjunctions.

References

1. S. Albers. New results on web caching with request reordering. In *Proc. SPAA*, pages 84–92, 2004.
2. P. Bajaj, D. Campos, N. Craswell, L. Deng, J. Gao, X. Liu, R. Majumder, A. McNamara, B. Mitra, T. Nguyen, M. Rosenberg, X. Song, A. Stoica, S. Tiwary, and T. Wang. MS MARCO: A human generated MAchine Reading COmprehension dataset. *arXiv:1611.09268v3*, 2018.
3. L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
4. R. Benham, J. Mackenzie, A. Moffat, and J. S. Culpepper. Boosting search performance using query variations. *ACM Trans. Inf. Sys.*, 37(4):41.1–41.25, 2019.
5. D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. DCC*, pages 342–352, 2002.
6. B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge. A refreshing perspective of search engine caching. In *Proc. WWW*, pages 181–190, 2010.
7. M. Catena and N. Tonellotto. Multiple query processing via logic function factoring. In *Proc. SIGIR*, pages 937–940, 2019.
8. S. Chaudhuri, K. Church, A. C. König, and L. Sui. Heavy-tailed distributions and multi-keyword queries. In *Proc. SIGIR*, pages 663–670, 2007.
9. C. Cheng, C. Chung, and J. J. Shann. Fast query evaluation through document identifier assignment for inverted file-based information retrieval systems. *Inf. Proc. & Man.*, 42(3):729–750, 2006.
10. F. M. Choudhury, J. S. Culpepper, Z. Bao, and T. Sellis. Batch processing of top-k spatial-textual queries. *ACM Trans. Spat. Alg. Syst.*, 3(4):13.1–13.40, 2018.

11. G. Chowdhury. An agenda for green information retrieval research. *Inf. Proc. & Man.*, 48(6):1067–1077, 2012.
12. N. Craswell, D. Campos, B. Mitra, E. Yilmaz, and B. Billerbeck. ORCAS: 20 million clicked query-document pairs for analyzing search. In *Proc. CIKM*, pages 2983–2989, 2020.
13. N. Craswell, B. Mitra, E. Yilmaz, D. Campos, and J. Lin. Overview of the TREC 2021 deep learning track. In *Proc. TREC*, 2021.
14. J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Sys.*, 29(1):1.1–1.25, 2010.
15. 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.
16. S. Ding, J. Attenberg, R. Baeza-Yates, and T. Suel. Batch query processing for web search engines. In *Proc. WSDM*, pages 137–146, 2011.
17. T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Sys.*, 24(1):51–78, 2006.
18. T. Feder, R. Motwani, R. Panigrahy, and A. Zhu. Web caching with request reordering. In *Proc. SODA*, pages 104–105, 2002.
19. S.-W. Hwang, S. Kim, Y. He, S. Elnikety, and S. Choi. Prediction and predictability for search query acceleration. *ACM Trans. Web*, 10(3):19.1–19.28, 2016.
20. S. Jonassen, B. B. Cambazoglu, and F. Silvestri. Prefetching query results and its impact on search engines. In *Proc. SIGIR*, pages 631–640, 2012.
21. D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Soft. Prac. & Exp.*, 41(1):1–29, 2015.
22. R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. WWW*, pages 19–28, 2003.
23. 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.
24. H. Ma and B. Wang. User-aware caching and prefetching query results in web search engines. In *Proc. SIGIR*, pages 1163–1164, 2012.
25. X. Ma, R. Pradeep, R. Nogueira, and J. Lin. Document expansions and learned sparse lexical representations for MSMARCO V1 and V2. In *Proc. SIGIR*, pages 3187–3197, 2022.
26. 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.
27. J. Mackenzie, M. Petri, and A. Moffat. Tradeoff options for bipartite graph partitioning. *IEEE Trans. Know. & Data Eng.*, 2022. To appear.
28. 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.
29. M. Marín and G. Navarro. Distributed query processing using suffix arrays. In *Proc. SPIRE*, pages 311–325, 2003.
30. C. Petersen, J. G. Simonsen, and C. Lioma. Power law distributions in information retrieval. *ACM Trans. Inf. Sys.*, 34(2):8.1–8.37, 2016.

31. H. Scells, S. Zhuang, and G. Zuccon. Reduce, reuse, recycle: Green information retrieval research. In *Proc. SIGIR*, pages 2825–2837, 2022.
32. T. K. Sellis. Multiple-query optimization. *ACM Trans. Data. Sys.*, 13(1):23–52, 1988.
33. E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in NLP. In *Proc. ACL*, pages 3645–3650, 2019.
34. G. Tolosa, L. Becchetti, E. Feuerstein, and A. Marchetti-Spaccamela. Performance improvements for search systems using an integrated cache of lists + intersections. *Inf. Retr.*, 20(3):172–198, 2017.
35. N. Tonellotto, C. Macdonald, and I. Ounis. Efficient query processing for scalable web search. *Found. Trnd. Inf. Retr.*, 12(4-5):319–500, 2018.
36. P. Yang, H. Fang, and J. Lin. Anserini: Reproducible ranking baselines using Lucene. *J. Data Inf. Qual.*, 10(4):1–20, 2018.