The Feasibility of Brute Force Scans for Real-Time Tweet Search

Yulu Wang[1] and Jimmy Lin[2]

[1] Department of Computer Science, University of Maryland
[2] David R. Cheriton School of Computer Science, University of Waterloo

ylwang@umd.edu, jimmylin@uwaterloo.ca

ABSTRACT

The real-time search problem requires making ingested documents immediately searchable, which presents architectural challenges for systems built around inverted indexing. In this paper, we explore a radical proposition: What if we abandon document inversion and instead adopt an architecture based on brute force scans of document representations? In such a design, "indexing" simply involves appending the parsed representation of an ingested document to an existing buffer, which is simple and fast. Quite surprisingly, experiments with TREC Microblog test collections show that query evaluation with brute force scans is feasible and performance compares favorably to a traditional search architecture based on an inverted index, especially if we take advantage of vectorized SIMD instructions and multiple cores in modern processor architectures. We believe that such a novel design is worth further exploration by IR researchers and practitioners.

Categories and Subject Descriptors: H.3.4 [Information Storage and Retrieval]: Systems and Software—Performance evaluation

Keywords: search architectures; multi-core processors

1. INTRODUCTION

Real-time tweet search exemplifies a class of retrieval problems that requires dealing with a host of architectural challenges compared to "traditional" web search. Many of these were identified by Busch et al. [5], including the need for rapid data ingestion of high velocity data streams and ensuring that documents are immediately searchable. As they discussed, one major performance bottleneck is the construction of the inverted index in an incremental fashion. For performance considerations, indexes and related data structures must be kept completely in main memory (which is a standard assumption today). Nevertheless, inverted indexing requires non-regular data structures and data access patterns that are at odds with modern processor architectures. This paper explores the following question: What if we abandoned document inversion? Can we envision a retrieval architecture that addresses the architectural challenges of real-time tweet search in a completely different way?

We explore the feasibility of a simple retrieval architecture based on brute force scans of document representations. The sketch of the approach is as follows: whenever we encounter a new document (tweet), we convert it into arrays of integers. "Indexing" simply means appending this representation to an existing in-memory buffer. For retrieval, we perform a *brute force scan* of these representations and compute query–document scores according to a scoring model. We can build on this simple idea by exploiting vector SIMD instructions and modern multi-core processors to further increase performance. Note that although this approach may seem reminiscent of bit signatures [9], the critical difference is that we work with *exact* document representations since there is no hashing involved.

Surprisingly, the retrieval performance of such an architecture is quite reasonable compared to a search engine built on a standard inverted index. We experimentally verified our techniques on data from the TREC Microblog evaluations from 2011 and 2012. Compared to the open-source search engine Lucene (a variant of which Twitter deploys for real-time search in production), we achieve query latencies that are within 30% when exploiting SIMD instructions and intra-query parallelism to its fullest.

ICTIR'15, September 27–30, Northampton, MA, USA.

ACM 978-1-4503-3833-2/15/09 ...$15.00.
DOI: http://dx.doi.org/10.1145/2808194.2809489.

2. BACKGROUND AND RELATED WORK

Initially, query evaluation based on brute force scans seems impractical, perhaps even outlandish. Nevertheless, we argue that this approach is worth considering for a couple of reasons. One challenge in designing software for modern architectures is the so-called "memory wall" [3]: increases in processor speeds have far outpaced improvements in memory latency. Today, memory latencies are hidden by hierarchical caches, but cache misses remain expensive. Another salient property of modern CPUs is pipelining, where instruction execution is split between many stages. Modern *superscalar* CPUs add the ability to dispatch multiple instructions per clock cycle. Pipelining suffers from two dangers or "hazards": *Data hazards* occur when one instruction requires the result of another, such as when manipulating pointers. Subsequent instructions cannot proceed until we first compute the memory location and the processor stalls. *Control hazards* are instruction dependencies introduced by branches. Such hazards are alleviated, but not completely eliminated, by branch prediction techniques.

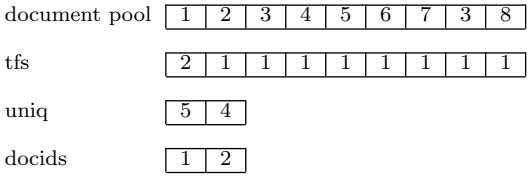

Figure 1: Document representations for two tweets.

The upshot is that branches and irregular memory accesses such as pointer chasing significantly reduce a processor's maximum performance. Unfortunately, inverted indexing and query evaluation algorithms are rife with exactly these inefficiencies. In contrast, brute force scans yield predictable memory accesses and code with minimal branching, yielding higher instructions-per-clock-cycle throughput. We wonder: has the latter caught up with the former?

The case for brute force scans is further bolstered by vector instructions that are common in today's processors. Advanced Vector Extensions (AVX) are extensions to the x86 instruction set architecture that support SIMD (single instruction multiple data) processing. AVX provides a number of instructions for operating on special 128-bit registers and 256-bit registers; AVX2 expands most vector integer AVX instructions to operate on 256-bit registers. The simplicity of the brute force scan approach allows us to exploit these instructions to achieve high instruction throughput.

3. SYSTEM DESIGN

3.1 Document Representations

We begin with a dictionary that provides a mapping from terms to 32-bit integer term ids. For simplicity, newly-encountered terms are assigned the next available term id. Whenever a new document (i.e., tweet) is encountered, it is first converted into an array of unique term ids, which is then appended to the end of a large array called the document pool. A parallel array of 8-bit integers stores corresponding term frequencies (tfs). We keep track of the number of unique terms in each document in a third array of 8-bit integers (uniq), which provides pointers into the otherwise unsegmented document pool. A final array keeps track of document ids. Figure 1 illustrates these data structures for two tweets: "BBC News: The BBC cuts budget" and "Just watched The Rite" (tokenized to "bbc new the bbc cut budget" and "just watch the rite"). To work with AVX2 instructions, we pad the document pool and the tf array to the nearest multiple of eight to align document boundaries with SIMD instructions. Thus, we have unpadded and padded variants of the data structures. To facilitate the computation of document scores, we also need to store document lengths and collection frequencies (but these data structures are not specific to our approach).

One aspect of our design is worth discussing: we decided *not* to compress the document pool for two reasons. First, our approach to assign term ids incrementally means that, on the whole, the ids are relatively large, and hence the document pool does not compress well. In an initial experiment, we tried compression using SIMD-BP128 [6], which yielded slower performance at only a modest saving in memory; we decided the tradeoff was not worthwhile. Although it is possible to assign term ids to facilitate compression (e.g., ordering terms based on frequency), this would complicate dictionary construction in a real-time scenario. Second, leaving the document pool uncompressed makes it straightforward to exploit SIMD instructions (more later); compression would significantly complicate matters.

Algorithm 1 Scan1

```
1: procedure SCAN1(Q, pool, tfs, uniq, docids)
2:     b ← 0
3:     for i ← 1 to N do
4:         s ← 0
5:         for j ← 1 to uniq[i] do
6:             for k ← 1 to |Q| do
7:                 if Q[k] = pool[b + j] then
8:                     s ← s + S(·)
9:                 end if
10:            end for
11:        end for
12:        if s > 0 then
13:            heap.add(docids[i], s)
14:        end if
15:        b ← b + uniq[i]
16:    end for
17: end procedure
```

3.2 Query Evaluation

We explored a number of query evaluation approaches based on a brute force scan of the document representations to compute query-document scores (query-likelihood in our case). The top k are retained in a heap and returned after all documents are processed. Four different implementations are described below:

Scan1. Our first approach (Algorithm 1) operates on the unpadded document pool and is a straightforward implementation consisting of three nested loops: over all documents, over all unique terms in each document, and over all query terms. The notation $S(\cdot)$ is shorthand for computing the score contribution of the current query term.

Scan2. In our second approach, which also uses the unpadded document pool, we unroll the innermost loop of the first approach (i.e., lines 6–10). This is accomplished by creating a separate query evaluation function for every query length (a relatively small number); such a design allows us to hard code the number of query terms and avoid branch mispredicts in the innermost loop.

AVXScan1. This approach replaces the two inner loops of Algorithm 1 with SIMD instructions. The key is an AVX2 instruction that performs element-wise comparison of two vectors of eight 32-bit integers concurrently (in 256-bit registers); the result is a mask indicating which of the integers match. Our approach is as follows: for each query term, we replicate its term id eight times in a 256-bit register, and then apply the AVX2 vectorized comparison to eight integers at a time from the document pool. Based on reading the result mask, if there is a match, we then compute the query term score contribution and add it to the current score. Thus, we scan eight unique term ids within a document at a time. As with the Scan2 algorithm, we implement different query evaluation functions for queries of different lengths to reduce branching when scanning the document pool (however, branches involved in checking the mask are unavoidable). Note that this technique requires padding the document pool and tf array with zeros to the nearest multiple of eight so that the AVX2 instructions do not cross document boundaries. We lack the space to provide de-

tailed pseudo-code, but we welcome the reader to examine our open-source code for more details.

AVXScan2. In this approach, we build on the previous algorithm by partially unrolling the outer loop that iterates over the documents. Instead of considering one document at a time, we consider six documents at a time, and use the vectorized comparison operation described in the AVXScan1 approach to score each document. Partial scores are held in a six-document array, and once all query-document scores have been computed, we examine each element in the array and insert non-zero scores into the heap; this is accomplished in a fully-unrolled loop to avoid branches.

The intuition behind this approach is to reduce loop overhead, and is similar to vectorized processing in databases [4]. The number of documents to process at once requires balancing two factors: a value too small does not significantly reduce loop overhead, while a value too large might cause cache churn. In our application, the value of six was heuristically determined. As with before, we lack sufficient space to provide the pseudo-code for this approach, but our source code is available online.

3.3 Exploiting Parallelism

Modern processors contain multiple cores, so it makes sense to explore how we can exploit parallelism in our brute force scan approach. The two obvious strategies are inter-query parallelism and intra-query parallelism.

With inter-query parallelism, we simply run our query evaluation algorithm on multiple threads, each of which operates on one query independently. In this approach, as the number of threads grows, query throughput increases up to a certain point, after which performance no longer increases (or may suffer) due to resource contention.

With intra-query parallelism, we split the collection into equal portions and scan the document pool in parallel on multiple threads; each thread maintains its own heap. There is a synchronization point in waiting for the threads to finish, and then to merge all the heaps to produce a final top k ranking. With this approach, we still evaluate one query at a time, but parallelism decreases latency. However, we eventually run into diminishing returns: beyond a certain point, the costs of synchronization outweigh the benefits of increased parallelism.

4. EXPERIMENTAL SETUP

We implemented our system in C and compared against the open-source search engine Lucene (version 4.3.1). All code used in our experiments is released under an open-source license.[1] The use of Lucene, which is written in Java, might strike some as an odd baseline, but we provide the following rationale: Lucene has been proven to be "industrial strength", with numerous deployments in production settings. Twitter itself runs a variant of Lucene for real-time search, and numerous other companies including LinkedIn and Bloomberg rely on Lucene in production. Thus, it is substantially more mature than research systems that may be implemented in C/C++. More importantly, though, Lucene has two features lacking in many research systems: First, near-real-time indexing, where ingested documents are available for search within a short amount of time. This is a key feature of our brute force scan approach and it would be unfair to compare our techniques against a system that does not have this feature. Second, Lucene has robust support for multi-threaded retrieval, which provides a point of comparison to parallel versions of our brute force scan approach. To provide a fair comparison, all Lucene indexes were loaded into main memory at startup. For both Lucene and brute force scan, we retrieved the top 1000 hits, per usual practice in IR research.

Experiments were conducted on a machine with two Intel Xeon E5-2680 v3 processors at 2.5 GHz. Each processor has 12 cores, with a total of 24 physical cores capable of supporting 48 virtual cores via hyperthreading. Our machine has 768 GB of RAM, although our experiments used only a small fraction of the total. The machine runs RedHat Enterprise Linux (release 6.6) and we used gcc version 4.9.1. All reported results represent the average of three trials.

We used test collections from the recent Microblog tracks at TREC [7, 8]. The 2011 and 2012 evaluations used the Tweets2011 collection (16 million tweets). We first verified with the TREC topics that our system generates exactly the same document scores and ranking as Lucene; subsequent experiments focused exclusively on efficiency. For these experiments, we took the first 1000 queries from the TREC 2005 terabyte track efficiency queries (we needed more queries for reliable measurements). The TREC Microblog queries are associated with timestamps, which are missing from these efficiency queries; thus, we searched over the entire collection (without any early termination).

5. RESULTS

After processing, the size of the document pool was 162 million; all data structures (but not the vocabulary) occupy 1026 MB memory. For the padded AVX2 representations, the document pool expanded to 225 million, or an increase of 39%. The padded data structures occupy 1341 MB memory in total. For reference, the Lucene index occupies 4.8 GB (although it stores positions and other metadata).

5.1 Single-Threaded Experiments

Results for the single-threaded experiments are shown in the first row of Table 1. We report average query latency (in milliseconds) of Lucene and our brute force scan approaches (with 95% confidence intervals). We see that the Scan1 approach is approximately 4.3× slower than Lucene, but each successive improvement to the basic brute force scan approach improves query latency. We get a big gain from unrolling the inner loop over query terms and writing a query evaluation function for queries of different lengths (Scan2). Applying SIMD instructions in the inner loop (AVXScan1) yields small improvements, but processing six documents at a time (AVXScan2) makes a noticeable contribution. Bottom line: the best brute force scan approach is within a factor of two of the performance of Lucene.

5.2 Multi-Threaded Experiments

Multi-threaded performance exploiting intra-query parallelism is shown in the rest of Table 1. We report query latency (in milliseconds) of Lucene versus the four brute force scan approaches (with 95% confidence intervals).

The results show that Lucene achieves maximum performance with 24 threads, which equals the number of physical cores in our machine—in other words, hyperthreading doesn't help. On the other hand, we get a small gain in

[1] `https://github.com/lintool/c-bfscan`

Threads	Lucene	Scan1	Scan2	AVXScan1	AVXScan2
1	178±25	769±4	417±11	408±13	361±3
2	109±37	610±23	358±56	297±16	261±15
4	86±38	456±24	270±6	228±7	189±1
8	76±13	297±7	168±7	141±5	135±1
12	73±8	217±3	126±5	107±8	109±7
24	62±1	156±3	92±1	87±3	88±7
48	81±1	134±2	89±2	81±2	84±3

Table 1: Latency (in milliseconds) with 95% confidence intervals of Lucene versus brute force scan, exploiting intra-query parallelism.

going from 24 to 48 threads with our brute force scan approaches. Interestingly, we noticed a different relative performance ordering of the brute force scan approaches compared to the single-threaded results. Even though AVXScan2 achieves the fastest single-threaded performance, it is (slightly) slower than AVXScan1 beyond 12 threads. Overall, the maximum performance we achieve with brute force scan (AVXScan1 with 48 threads) is about the same speed as Lucene under the same parallelism setting, but still a bit slower than the absolute best Lucene results (24 threads). Bottom line: the best brute force scan approach is only 30% slower than the best Lucene performance.

Multi-threaded performance exploiting inter-query parallelism is shown in Table 2. We report the throughput of Lucene versus the four different brute force scan approaches (with 95% confidence intervals). We see that maximum throughput with Lucene is achieved with 24 threads. This suggests that the benefits of additional parallelism are outweighed by resource contention that reduces overall throughput. Unfortunately, none of our brute force scan approaches are able to achieve anywhere close to the throughput of Lucene. Interestingly, however, just as in the intra-query parallelism case, we noticed a different relative performance ordering of the brute force scan approaches compared to the single threaded results. The Scan2 approach achieves the highest throughput with maximum parallelism, even though it has significantly lower performance on a single thread. Bottom line: inter-query parallelism does not appear to be the best way to exploit multiple cores using brute force scan techniques in today's processors.

6. FUTURE WORK AND CONCLUSION

What do we make of these results? Although none of the brute force scan approaches beat Lucene in terms of performance, we were surprised that the results are even this close. To recap: on a single thread, retrieval using the best brute force scan approach is within a factor of two of using an inverted index, and when exploiting intra-query parallelism to the fullest extent, the performance gap drops to 30%. We believe that we are a couple more optimizations away from achieving performance parity, and invite other researchers to help us get there.

The obvious advantage of our approach is that it eliminates the complexities of document inversion, which we do not show with indexing experiments here. In real-time search, there is another important advantage: users most often care only about the latest results. With an inverted index, it is desirable to traverse postings lists "backwards" (from most recent) and early exit when enough results have been accumulated [2]. Most systems are not designed this way, which foregoes optimization opportunities; adapting traditional query evaluation algorithms to operate in this

Threads	Lucene	Scan1	Scan2	AVXScan1	AVXScan2
1	7.5 ±0.1	1.3±0.1	2.5±0.3	2.5±0.1	2.9±0.3
2	20.7 ±1.5	2.6±0.1	4.8±0.6	5.1±0.2	5.9±0.1
4	38.2 ±0.7	5.0±0.1	9.4±0.7	9.6±0.3	11.0±0.1
8	52.6±12.0	9.3±0.2	17.1±1.3	17.0±0.3	19.2±0.5
12	104.0 ±4.6	13.5±0.3	24.5±2.2	23.8±1.2	25.8±1.5
24	188.7±21.0	25.3±1.2	41.6±5.1	38.3±1.9	40.0±0.1
48	164.4±10.3	30.3±0.5	53.1±2.7	49.3±2.5	48.4±4.2

Table 2: Throughput (query per second) with 95% confidence intervals of Lucene versus brute force scan, exploiting inter-query parallelism.

manner is non-trivial. In contrast, with an approach based on brute force scans, it is easy to factor in temporal constraints: since the documents are arranged chronologically, we simply stop scanning once we've reached the proper time.

We see an additional advantage beyond optimizations for real-time search: many web search engines today adopt a multi-stage architecture [1] that begins with a simple ranker (e.g., BM25) followed by one or more complex (machine-learned) rankers. Later stages typically access document-level features (e.g., static priors) that are not stored in the inverted index, thus requiring a separate "document store" from which those feature vectors are fetched or computed. This requires storing two separate structures (in essence, both a forward and an inverted index). With the brute force scan approach, the document store is no longer necessary, as document-level features can be directly stored in the document pool (suitably encoded). This yields a simpler and potentially more efficient end-to-end architecture.

To conclude: we show that the simplest possible approach to document retrieval—one based on simply scanning all documents in the collection—is surprisingly feasible given today's hardware. We hope to inspire future work that further explores this novel line of inquiry.

7. ACKNOWLEDGMENTS

This work was supported by the U.S. National Science Foundation under IIS-1218043 and CNS-1405688. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsor.

8. REFERENCES

[1] N. Asadi and J. Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. *SIGIR*, 2013.

[2] N. Asadi, J. Lin, and M. Busch. Dynamic memory allocation policies for postings in real-time Twitter search. *KDD*, 2013.

[3] P. Boncz, M. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *CACM*, 51(12):77–85, 2008.

[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. *CIDR*, 2005.

[5] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at Twitter. *ICDE*, 2012.

[6] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[7] I. Ounis, C. Macdonald, J. Lin, and I. Soboroff. Overview of the TREC-2011 Microblog Track. *TREC*, 2011.

[8] I. Soboroff, I. Ounis, C. Macdonald, and J. Lin. Overview of the TREC-2012 Microblog Track. *TREC*, 2012.

[9] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.