In Vacuo and *In Situ* Evaluation of SIMD Codecs

Andrew Trotman
Department of Computer Science
University of Otago
Dunedin, New Zealand
andrew@cs.otago.ac.nz

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
jimmylin@uwaterloo.ca

ABSTRACT

The size of a search engine index and the time to search are inextricably related through the compression codec. This investigation examines this tradeoff using several relatively unexplored SIMD-based codecs including QMX, TurboPackV, and TurboPFor. It uses (the non-SIMD) OPTPFor as a baseline. Four new variants of QMX are introduced and also compared. Those variants include optimizations for space and for time. Experiments were conducted on the TREC .gov2 collection using topics 701-850, in crawl order and in URL order. The results suggest that there is very little difference between these codecs, but that the reference implementation of QMX performs well.

CCS Concepts

•**Information systems** → **Search index compression;**

Keywords

Search, Score-at-a-Time, Compression, Procrastination

1. INTRODUCTION

Two of the most important aspects in a search engine are the size of the index and the time it takes to search. If the size of the index can be reduced, then more documents can be indexed in the same amount of space. In a massively distributed (or replicated) search engine that stores the index in memory, a reduction in the index size has a direct payoff as a reduction in the amount of memory necessary to store the index during search time. This payoff is only worthwhile if it is not associated with an increase in search latency. An increase in latency due to a reduction in index size will, in all likelihood, result in an increase in the number of CPU cycles needed to search.

The space / time tradeoff has been a topic of investigation in the information retrieval literature for many decades. Several techniques have been proposed, one of the most common is index compression. This investigation examines three SIMD-based codecs of three different classes. TurboPackV is a bin-packer, TurboPFor is an SIMD-implementation of PFor, and QMX is in the Simple family. Four new variants of QMX are introduced. As a baseline, the non-SIMD OPTPFor [21] is used. OPTPFor has become a popular implementation of PFor [22].

Each of these codecs is tested in three experiments. The first examines compression effectiveness and decompression efficiency (hereafter, effectiveness and efficiency) using a public document-ordered dump of the TREC .gov2 collection. The second examines query-by-query search performance in the JASS impact-ordered score-at-a-time search engine [8] using TREC topics 701-850. In the final experiment, the size of the JASS index and overall latency are examined as a space / time tradeoff. All experiments were conducted on both the crawl-ordered and URL-ordered collection.

The results show that the PFor codecs do result in a smaller index, but that there is a measurable performance hit as a consequence of the added decoding complexity. On a query-for-query comparison QMX outperforms the others (including all variants). But overall the best performing codec is dependent on the document collection ordering.

ADCS '16, December 05 - 07, 2016, Caulfield, VIC, Australia

ISBN 978-1-4503-4865-2/16/12...$15.00

DOI: http://dx.doi.org/10.1145/3015022.3015023

2. POSTINGS LISTS

Modern search engines are typically based on an inverted index. This index stores a list of all the unique terms in a corpus, and for each term a list of which documents contain that term (and how many times) called a postings list. A postings list is represented:

$$< d_1, f_{t,d_1} >< d_2, f_{t,d_2} > ... < d_n, f_{t,d_n} >$$

Where d is a document ID and $f_{t,d}$ is the number of times term t occurs in document d (the so-called term frequency). As a side effect of indexing, these postings lists are ordered in increasing order of d, consequently the index is known as document-ordered.

Persin et al. [11] observe that fewer integers need be stored if the postings lists are ordered first on decreasing term frequency, then on increasing document ID. That is, the postings list is represented:

$$< f_{t,d}, d_1, d_2, ...d_n > ...$$

Where f is the term frequency and the d's are the IDs of the documents that contain that term $f_{t,d}$ times. This representation is known as frequency-ordered because the primary sort key is the term frequency.

Many common ranking functions (such as BM25 [12]) are expressed as the sum, over the query terms, of a set of constants, essentially:

$$rsv = \sum_{t \in q} f(t, d) \quad (1)$$

Where q is the query and rsv is the retrieval status value (higher is better), the sort key used in a ranking search.

Anh et al. [1] propose that $f(t, d)$ be computed at indexing time and that the only part of the ranking function performed at search time is the sum. That is, replace the $f_{t,d}$ value in a frequency-ordered index with an impact score $i_{t,d}$, and in doing so introduce impact-ordered indexes:

$$< i_{t,d}, d_1, d_2, ...d_n > ...$$

Typically the document length is used in $f(t, d)$ and so the documents in each postings segment differ between a frequency-ordered and an impact-ordered index.

Each of the index orderings is processed according to its type. For a document-ordered index document-at-a-time (DAAT) techniques such as WAND [4] and BlockMax [6] are typically used. Frequency-ordered indexes are often processed term-at-a-time (TAAT), whereas impact-ordered indexes are processed score-at-a-time (SAAT) using techniques such as the Anytime [9] algorithm.

In a recent benchmark of open-source search engines [8] the performance of impact-ordered indexes with Anytime was typically slightly better than the performance of document-ordered indexes with quasi-succinct indexes [20] (the most efficient of the DAAT approaches there). These differences could easily be caused by other factors such as parsers, stemmers, and so on. So it remains unclear whether DAAT or SAAT is the most efficient way to process postings, and that is left for further work.

For the experiments herein document-ordered postings are used to compare the effectiveness and efficiency of the codecs. Impact-ordered indexes as implemented in the JASS search engine are used to measure search latency. The JASS index is used for the space / time tradeoff experiment.

3. COMPRESSION

In early search engines the index was stored on moving-parts hard drives. Reducing index size was paramount as it reduced the amount of data read from disk, which in turn decreased latency. The typical approach is to compress each postings list independently.

In an index-on-disk environment Trotman [17] examined the efficiency of bit-aligned codes such as Elias, Golomb, and Binary Interpolative Coding along with variable byte encoding. Trotman shows that bit-aligned codes are inefficient when the disk is modelled along with the decompression characteristics. Scholer et al. [13] differently demonstrated this being the case within a search engine.

Byte-aligned codes are not as space efficient as bit-aligned codes. Anh & Moffat [2] addressed this by introducing 32-bit word-aligned codes such as Simple-9 and 64-bit word-aligned codes such as Simple-8b [3]. These codes are both space efficient and fast to decode.

Advances in hardware led to two changes in the way the index is stored. First, the availability of cheap RAM made it possible to store the entire index in memory. Second, CPUs now contain SIMD instructions and consequently a new generation of codecs has emerged.

Stepanov et al. [15] introduce varint-G8IU, which packs the maximum number of integers into an 8-byte word preceded by a selector giving details on how to unpack.

Lemire & Boytsov [7] introduce SIMD-BP128 which packs 128 integers of the same bit width into as few 16-byte words as possible. 16 of these blocks are then preceded by a 16-byte selector explaining how to decode these blocks.

Trotman [16] compares byte-aligned codes, word-aligned codes, and SIMD codes in both effectiveness and efficiency. He shows that the SIMD codecs are, indeed, more efficient than the previous methods, and goes on to introduce an effective and efficient SIMD codec called QMX.

QMX is based on the Simple family of codecs in so far as it fixed-width bin-packs as many integers as possible into a single, or sometimes two, 16-byte machine words. It is based on the SIMD family in so far as it uses SIMD instructions to unpack integers. For SIMD codecs it is novel in that it has special handling for short sequences that are common in postings lists. For Simple family it is novel that it stores the selectors at the end of the compressed sequence rather than it being part of the machine word. It is additionally novel in so far as it run-length encodes the selectors.

QMX not only performs well in theory, but in practice too. The reference implementation[1] is included in the JASS search engine where it is the preferred codec – making it a candidate for further optimisation.

4. EXTENSIONS TO QMX

The QMX codec fixed-width bin-packs as many integers as it can into a single 128-bit SIMD-word sized *payload* when integers are 1, 2, 3, 4, 5, 6, 8, 10, 16, or 32 bits wide. It fixed-width bin-packs into a two SIMD-word sized payload when they are 7, 9, 12, or 21 bits wide. It additionally packs 256 0-bit integers in a selector without a payload. These payloads are laid out first when serializing.

In total there are 15 encodings described and the encoding is stored in the high nybble of an 8-bit selector called an *eXtractor*. The low nybble of the selector is used to store a run-length (or *Multiplier*) of that selector. These selectors are laid out second when serializing.

Finally, it is necessary to find the start of the selectors so that the decoder knows how to decode the first and subsequent payload. This pointer is stored variable-byte encoded at the end of the serialized sequence. Figure 1 schematically represents this layout.

By laying out in this way, pointers to the start and to the end of the compressed sequence are needed for decoding. Both of these are typically known in a search engine as the postings are normally serialized one after the other in memory (so the start of the next postings list is the end of the previous one). In the unlikely event that the index is stored on disk, the start of the postings list and its length are needed in order to read from disk and consequently both pointers are known once the data is loaded into memory.

Four variants are discussed in the remainder of this section. Each builds on the previous by taking the previous and adding either a possible effectiveness optimization or efficiency optimization. That is, Variant 2 includes all changes for Variant 1, and Variant 4 includes all changes for Variants 1, 2, and 3.

[1] http://www.cs.otago.ac.nz/homepages/andrew/papers/QMX.zip

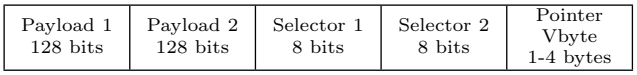

Figure 1: QMX stores all 128-bit payloads before the 8-bit selectors before the variable-byte encoded pointer to the selectors. Each selector is broken into 2 parts, a 4-bit eXtractor describing the bit-packing and a 4-bit Multiplier (or run-length) of the eXtractor.

4.1 QMX Variant 1

Around half of the vocabulary terms in an inverted index can be expected to occur in only one document [19]. Of the remaining terms, a large proportion only occur twice. In an impact-ordered index the situation is worse as a postings list is broken into segments, each of which may be singleton.

Many codecs are incapable of encoding or decoding a single integer. For example, Group Varint cannot encode or decode fewer than 4 integers. Trotman [16] outlines how QMX can encode a single integer, but it cannot decode a single integer. The reference implementation decodes a single integer into the correct integer and at least 3 over-run integers of undefined value. Problematically, when the QMX reference implementation is used to decode into an exact-sized buffer the over-run integers can (and do) overflow that buffer. Worse, an overflow also happens when reading the coded sequence from an input buffer.

QMX Variant 1 addresses this issue. That is, it can both encode and decode between 1 and 3 integers in addition to the existing encodings of 4 or more integers. It does this without overflowing either the input or output buffer.

Only 15 of the 16 possible eXtractors are used in QMX with the remaining eXtractor being reserved for future use (eXtractor F_{16}). In Variant 1 this future use is appropriated for encoding 3 or fewer integers.

As the high nybble of the selector is already taken as the eXtractor, the only remaining part of the selector available for further re-purposing is the Multiplier. There are 16 available values, this somewhat restricts the number of ways a sequence of up to 3 integers can be encoded.

In Variant 1 these integers are fixed-width bin-packed into as few bytes as possible (that is, a payload shorter than an SIMD-word). The Multiplier is then used to identify both the number of packed integers and the width of the encoding. An obvious solution is to break the Multiplier into two 2-bit *snip*s, one storing the run length and the other storing the byte width.

The details are shown in Table 1. Column 1 lists the bit pattern, column 2 gives the interpretation when seen as the width (the high 2 bits), column 3 gives the meaning when seen as the run length (low 2 bits). In this way, if the high two bits are 00_2 then all the integers are encoded in 8 bits. If the low two bits are 10_2 then there are two such integers. A Multiplier of 0010_2, therefore, represents two 8-bit integers in the short payload.

For example, the integer sequence ($0F_{16}$, $F1_{16}$) would be encoded $0F_{16}$ $F1_{16}$ $F2_{16}$ along with the QMX pointer to the start of the selectors. The sequence $0F0_{16}$ $1F1_{16}$ would be encoded as two 16-bit integers and therefore as 00_{16} $F0_{16}$ 01_{16} $F1_{16}$ $F6_{16}$ along with the QMX pointer to the start of the selectors.

An additional change is needed to the QMX reference implementation. There, a sequence is encoded based on the bit width of the largest integer in a sequence. For example, a sequence of six 16-bit integers is packed into an 8-integer payload with 2 over-run integers. Now they are packed to fill payloads wherever possible, the example is newly encoded as a 4-integer payload with the two remaining integers encoded using the re-purposed selector.

Table 1: Encoding of the F_{16} selector used in Variant 1. 2 bits are used for the byte-width and 2 bits for the run-length.

Bit Pattern	Width	Run-length
00_2	8-bits	unused
01_2	16-bits	3
10_2	24-bits	2
11_2	32-bits	1

4.2 QMX Variant 2

At the end of the encoded sequence QMX stores a variable-byte encoded pointer to the beginning of the selectors (see Figure 1). This pointer is only needed so that the selectors can be processed in a sequential manner from low memory to high memory.

QMX Variant 2 removes this pointer from the end of the compressed sequence. This is achieved by reversing the selector sequence and processing from high memory to low memory (in the reverse direction to QMX). That is, the first selector is the final byte in the encoded sequence, the second selector is the second to last, and so on.

Processing the selectors in this way could have a negative impact on decoding efficiency if the CPU is less able to predict this usage pattern or less able to pre-fetch predecessor bytes from memory than successor bytes. However, it will always have a positive effect on effectiveness as every sequence will be between 1 and 5 bytes shorter as a consequence of the loss of the pointer.

4.3 QMX Variant 3

Trotman [16] observes that the selectors can be decoded in a `case` statement. He goes on to encode the run-lengths in two's complement and uses the optimization of `case`-statement fall-though to avoid a loop. That is, if the run-length is 1 then a certain set of lines of code are needed for decoding, those are in a single `case`, and followed by a `break`. If the run-length is 2 then a repeat of those lines of code in a second `case` above the first, but without a `break`, will cause the code to be executed twice before a `break`. However, this approach does mean that other house-keeping (such as incrementing pointers) is repeated at each `case` boundary.

Variant 3 does not use `case` fall-through, but rather unwinds each loop in each `case`. This makes the decoding routine substantially longer, but means that the house-keeping only occurs once per selector. Unwinding the loops, however, is also likely to result in an increase in the number of program-cache misses. It is not clear whether the efficiency gain from reduced house-keeping is greater or less than the efficiency hit from having a larger code base.

4.4 QMX Variant 4

The memory access patterns used by QMX and variants is predictable. The payloads are always processed from low memory to high memory, they are SIMD-word aligned and SIMD-word sized. In Variant 1 the selectors are processed byte at a time and from low memory to high memory. In

Variant 2 and Variant 3 the selectors are processed byte at a time from high memory to low memory.

It is reasonable to expect the CPU to notice the access patterns at run-time and to automatically pre-fetch data into the cache before it is needed. However, during decoding the encoded sequence in the input buffer is read-only and the decoded sequence in the output buffer is write-only. Leaving parts of the coded sequence in the cache leaves less space for the decoded sequence which could result in delays during decoding (or later accesses during postings processing).

The x86-64 ISA includes memory pre-fetch instructions that shift the contents of a given memory location closer to the CPU. Of the instructions, the `PREFETCHNTA` instruction is appropriate for decoding. That instruction shifts the data on the given cache line close to the CPU in the knowledge that it will be used only once – i.e. it can be cache evicted upon first access. The Intel specification leaves room for the CPU to implement this however it chooses, but it is generally believed to move data into the L1 cache and not the other caches.

Variant 4 adds pre-fetch instructions so that every time a memory read occurs, the next word to be accessed is pre-fetched. This happens separately to the payloads and the selectors. That is, each payload is pre-fetched before it is used and each selector is pre-fetched before it is used.

If the CPU is capable of predicting the QMX access patterns then the pre-fetch instructions are unnecessary, but do make the program longer (risking program-cache misses) and must be decoded (which will take time). If the CPU is unable to predict the pattern, then an improvement in efficiency will be seen.

5. OTHER CODECS

Trotman [16] compares the efficiency and effectiveness of QMX to a multitude of prior codecs including several implementations of byte-aligned codes, words-aligned codes, and SIMD codes. Since then, more efficient implementations of some of those algorithms have become available. There have also been new SIMD implementations of these other codecs.

5.1 TurboPackV

TurobPackV[2] is an SIMD-based fixed-width bin-packer. The reference implementation computes the smallest number of bits necessary to store the largest of 128 consecutive integers and bin-packs all 128 integers into a payload that bit width.

TurboPackV is similar to SIMD-BP128 in the encoding, however it differs in how the selector is stored. SIMD-BP128 appends 16 blocks into a meta-block and prepends 16 selectors (an SIMD-word) to the meta-block. In doing so, SIMD-BP128 ensures all memory reads are SIMD-word aligned. The reference implementation of TurboPackV does not manage the selector, it returns it to the caller to store. QMX has already been compared to SIMD-BP128, this experiment was not reproduced.

For this work, the selector and the payload are serialized alternately. That is, first the first selector (a single-byte) is written, that selector describes the width of the first payload (a variable number of SIMD-words) which is written second; followed by the second selector, then the second payload, and so on. If there are fewer than 128 integers to encode TurboPackV fixed-width bin-packs those integers into a sequence of bytes.

[2]https://github.com/powturbo/TurboPFor

5.2 TurboPFor

With TurboPackV and any other bin-packer, the encoding effectiveness is largely dependent on the bit width being used, which in turn is prone to exceptions having catastrophic effects. For example, the sequence 1, 4, 2, 8 could be packed into 2 bytes using 4 bits per integer. However, by changing the 8 to a 16, it is no longer possible to store all the integers in 4 bits, as 5 bits are needed to store the 16. The sequence would now take 3 bytes.

Zukowski et al. [22] observe that by keeping an exceptions list it's possible to keep encoding effectiveness high while not forfeiting decoding efficiency.

The usual implementation is to encode, inline with the integers, a pointer to the exception in an exceptions list and to patch-up the decoded sequence from this list while decoding. The general approach used in information retrieval is to take a run of 128 integers, to place the largest at-most 10% into an exceptions list, then to fixed-width bin-pack the remainder into as few words as possible. When serializing, a selector identifying the bit width is stored first, along with details of the length of the exceptions list, then the bin-packed payload, then the exceptions list.

TurboPFor[3] differs from this standard approach. 128 integers are encoded in a block, but the exceptions are stored first and the remainder second (along with a set of flags indicating the location of the exceptions). Both blocks are fixed-width bin-packed. The exceptions are decoded into a temporary array, then the remainder is decoded and combined with the exceptions in a single pass. The decoding is performed using SIMD instructions. In the case where there are fewer than 128 integers to pack, they are likewise packed into 32-bit sequences.

5.3 OPTPFor

OPTPFor [21] includes two optimizations over PFor. In the first, the exceptions are stored differently; whereas before a pointer to the exception was stored in-line with the encoded integers, in OPTPFor the low-bits of the integer are stored in-line and the high-bits are stored in an exceptions list along with details of which integers to patch. The high-bits and patch table are themselves encoded using Simple-16.

In the second optimization the way exceptions are computed is different. Rather than targeting 10% of the integers as exceptions, OPTPFor chooses the width of the bin-packed integers to optimize on encoded size.

The implementation used herein[4] is provided by Lemire & Boytsov [7] and is not SIMD. In the case where there are fewer than 128 integers to encode (i.e. short postings lists or the end of a longer postings list), the integers are encoded using variable-byte encoding.

6. COMPARISON

Three sets of experiments were conducted. The first uses public data made available by Lemire[5] to examine the space needed to encode the document IDs in a document-ordered index along with the time needed to decode. The second

[3]also at https://github.com/powturbo/TurboPFor

[4]https://github.com/lemire/FastPFor

[5]http://lemire.me/data/integercompression2014.html

examines search time in an impact-ordered search engine. The third is the efficiency / effectiveness tradeoff.

QMX Variant 1 is expected to be negligibly slower than QMX because of additional work needed at the end of each postings list. Variant 2 is expected to be slower again if the CPU cannot predict the backwards access pattern. Variant 3 will be more efficient if the overheads of the fall-through are high. Variant 4 will be more efficient if the pre-fetch instructions are effective. TurboPackV is expected to be most efficient on moderate-sized postings lists. The PFor codecs are expected to be effective, but less efficient than the others.

6.1 Preliminaries

All experiments use the TREC .gov2 document collection of 25,205,179 web pages crawled from the .gov domain in 2004. This dataset is commonly used in efficiency experiments and was used by both Lemire & Boytsov [7] and Trotman [16] in prior work.

Experiments were conducted on a 4 CPU Intel quad-core Xeon X5550 at 2.66GHz with Linux 2.6.32-504.3.3.el6.x86_64 (Centos 6.6 (Final)). Programs were written in C/C++ and gcc/g++ version 5.3.0 with optimization set to level 3 (`-O3`). Although multiple cores were used during indexing, only a single core was used for search.

The implementations of each algorithm were verified using Lemire's dataset. Each postings list was encoded, decoded, then the decoded list was compared byte-for-byte to the original list. The implementations were considered accurate only if the original and decoded lists were identical for all postings lists in the collection. They were also verified in JASS, they were considered accurate only if all codecs produced the same results list (which differ from crawl-ordered and URL-ordered because the internal document IDs differ).

In a document-ordered index the document IDs in each postings list are, by definition, stored in increasing order. In an impact-ordered index the document IDs are normally stored in increasing order within each segment. In both cases, document IDs form a strictly monotonically increasing sequence. Each of the codecs examined is better at storing large numbers of small integers than large numbers of large integers, consequently the document IDs were converted into deltas (also known as d-gaps or differences) before being encoded. This is simply performed by subtracting the preceding integer from the current integer before encoding, and performing a cumulative sum on decoding.

Queries were derived from TREC topics 701-850 by taking the title field, removing duplicate words, and stemming.

The reader is assumed to be viewing this document in colour so as to be able to distinguish the lines in each graph.

6.2 Document ID Ordering

Lemire provides two datasets that are derived from .gov2. Both are a dump of the document IDs (without term frequencies) for all terms seen in the collection 100 times or more (document frequency $\geq$ 100). It is unclear whether any other form of stopping or stemming was performed as the terms themselves are not provided. Without the terms it is not possible to use this as an index to a search engine, but it is also not possible to reconstruct the original documents (a requirement of the nondisclosure agreement).

The two datasets differ in document order. The first is in the order the documents are seen in the collection, i.e. crawl order. For the second the documents were first re-ordered alphabetically on URL. For this particular collection URL-reordering is a known technique for both reducing the size of the index and decreasing latency while searching [14].

Each of the codecs described in Section 4 and Section 5 along with the reference implementation of QMX were tested on each of these two data sets. Each postings list in the dataset was encoded and the size (in bytes) of the encoding was stored. It was then decoded and the time taken to decode (measured in nanoseconds using the C++ `steady_clock`) was stored. The lengths this program reported were the means over all postings lists of a given number of document IDs (for example, the mean of all postings lists with 100 document IDs in them) – which does not vary from run to run. The times reported are computed in a similar fashion (mean of all lists of a given length). The experiment was conducted 25 times and the numbers reported are the medians of the means.

The results when tested on the crawl-ordered collection are shown in Figure 2 and Figure 3. Figure 2 presents the encoding effectiveness of each of the codecs on a log / log scale. On this set of data there is very little difference seen in the size required to store postings lists of any length examined. Figure 3 presents, on a log / log scale, the decoding time taken for postings lists of a given length. There it can be seen that the PFor family takes longer to decode than the others – the exceptions lists take time to process.

There is very little difference in the others, except that TurboPackV is faster than the others when the lists are short. This difference is likely to be due to the number of selectors that must be decoded. When there are 128 integers TurboPackV needs only 1 selector, but the QMX variants require more. However, when the lists get denser (i.e. longer) QMX is able to encode 256 integers in a single selector that itself can have a run-length of up to 16. In other words, QMX can encode as many as 4096 integers in a single selector. There is an overhead to processing each selector and this is reflected in the decoding times.

Figure 4 and Figure 5 show the results when tested on the collection in URL order. A similar pattern emerges; it is not obvious that any one scheme is more effective than any other – the codecs are highly dependent on the sequences of integers in the postings lists. The QMX-based schemes all show far more chaotic behaviour than the others. To explore this, all postings lists segments with more than 78,000 document IDs and that QMX encoded into fewer than 100 bytes were examined. Each and every one was the name of an American county[6] which are likely to be sorted beside each other in URL order. Given that a single 1-byte QMX selector encoding 0-bit integers (without payloads) can encode 4,096 document IDs, and that 78,000 IDs could, therefore, be encoded in as few as 20 selectors and 0 payloads (i.e. 20 bytes), URL ordering this collection does appear to be highly effective for some terms and QMX. With respect to efficiency, the PFor-based codecs take longer to decode than the others, which appear to behave similarly.

6.3 Comparison in JASS

Section 6.2 suggests that the effectiveness of PFor codecs is not substantially better than the QMX variants and TurboPackV. It also suggests that decoding is less efficient.

[6]"marinette" was the longest with 78,103 document IDs encoded into 98 bytes.

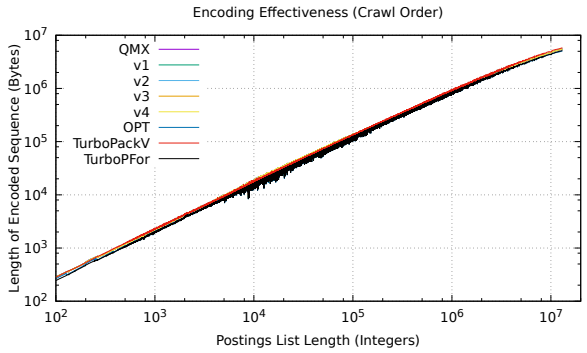

Figure 2: Encoding effectiveness when tested on TREC .gov2 in crawl order. Bytes for a list of a given length is shown on a log / log scale.

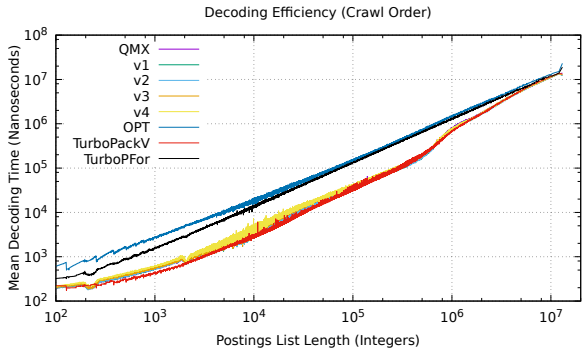

Figure 3: Decoding efficiency when tested on TREC .gov2 in crawl order. Nanoseconds to decompress a list of a given length is shown on a log / log scale.

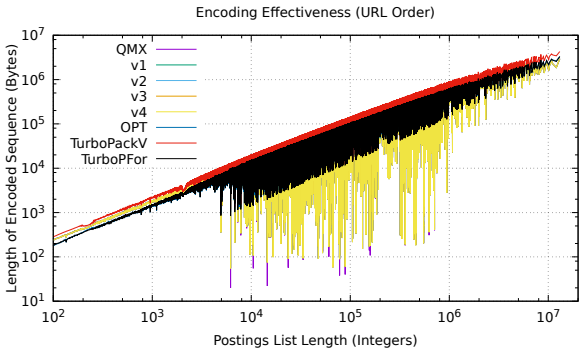

Figure 4: Encoding effectiveness when tested on TREC .gov2 in URL order. Bytes for a list of a given length is shown on a log / log scale.

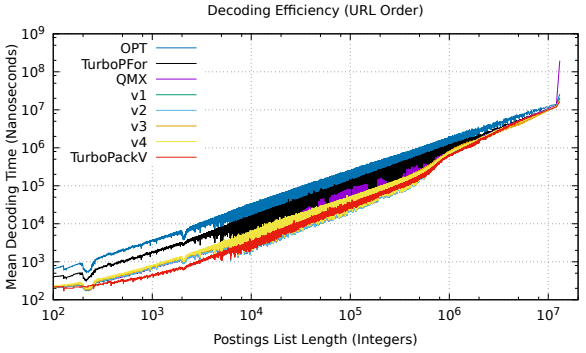

Figure 5: Decoding efficiency when tested on TREC .gov2 in URL order. Nanoseconds to decompress a list of a given length is shown on a log / log scale.

For a search engine, there are two issues being traded-off. The first is index size. The second is search latency. Naturally, a smaller and faster index is better than a larger and slower index.

The .gov2 collection was pre-processed then indexed using the ATIRE [18] search engine. In this case ATIRE loads the documents, cleans them by replacing all non-ASCII characters with spaces, removing all HTML tags (but not their content), and s-stemming.[7] ATIRE produces an impact-ordered index, in this case BM25 with $k_1 = 0.9$ and $b = 0.4$ was chosen as the ranking function (the ATIRE default), the impact values were computed as 9-bit integers using the techniques of Crane et al. [5] and stored in 16-bit integers in the index.

ATIRE uses a parallel indexing pipeline which concurrently reads from multiple input channels, parses documents in parallel, then merges these parsed documents into a centralized index. Document IDs are assigned to documents at the merge stage, consequently the exact order of the documents in the index is non-deterministic. Unfortunately, this non-determinism is a realistic indexing model as it is fast. Even though the exact results are non-reproducible, similar results will be seen each time the collection is re-indexed.

The ATIRE index was then converted into a JASS index using the JASS tool `atire_to_jass_index` – this is the standard way of building a JASS index, an approach the JASS authors took to avoid having to build an indexing pipeline.

[7]removes suffixes 's' and 'es', and turns suffix 'ies' into 'y'.

Each JASS index is a translation of the exact same ATIRE index and is not subject to further non-determinism.

For the codecs tested, JASS pads the index so that each and every postings list and postings list segment starts on an SIMD-word aligned boundary (16-bytes). On modern CPUs unaligned SIMD reads are just as fast as aligned SIMD reads on aligned boundaries, but they are slower on unaligned boundaries. The JASS authors clearly chose to trade off effectiveness for efficiency in this case.

Each experiment was conducted 25 times and the reported results are the medians of those 25 runs. Search, using TREC topics 701-850, was performed to completion with no early termination.

Figure 6 shows, on a log / linear scale, the time to search the crawl-ordered collection. It is ordered from highest to lowest latency according to QMX. From visual inspection there is no material difference between any of the codecs.

The ATIRE indexer can be configured (with compile-time flags) to index documents in collection order. This is unrealistic as it takes substantially longer to index sequentially than it takes to index in parallel, however indexing .gov2 in URL order has become common practice due to effectiveness and efficiency gains seen by doing so. So, the documents in the .gov2 collection were sorted into URL order, indexed sequentially using ATIRE in deterministic order, then converted to a JASS index. The same document cleaning, stemming, and ranking parameters were used as before.

Figure 7 shows, on a log / linear scale, the time to search the URL-ordered collection. It is ordered from highest to

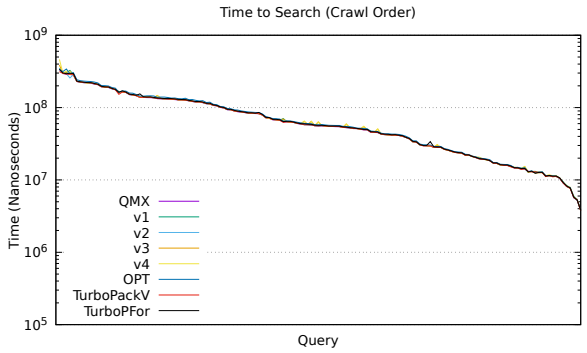

Figure 6: Time to search crawl-ordered .gov2 with queries ordered from highest to lowest latency.

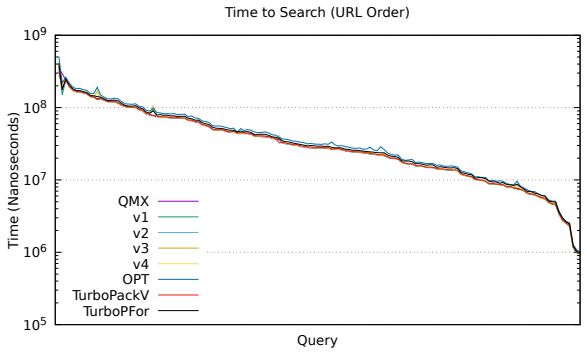

Figure 7: Time to search URL-ordered .gov2 with queries ordered from highest to lowest latency.

lowest latency according to QMX. It is clear from this figure that, as seen in Section 6.2, the PFor codecs are slower than the others.

Figures 6 and 7 are presented on log / linear scales emphasizing differences for all query latencies, however little can be deduced about the efficiency of QMX, QMX variants, or TurboPackV because the lines are indistinguishable.

Table 2 presents the tournament matrix of which codec outperforms which other codec when the collection is in crawl order. The first row presents the name of the codec being compared to the first column. The numbers in the other cells report the number of times that codec is no worse (i.e. time $\leq$). For example, the 17 in the cell in column "v2" row "QMX" reports that in 17 of the 150 queries Variant 2 was better or equal to QMX. The last row of the table presents the total number of "wins" for the given codec. Table 3 shows the same table for the URL-ordered collection.

QMX outperforms the others in both cases. TurboPackV is effective in crawl order but not in URL order – in URL order the difference in the number of selectors is likely to be smaller. Variant 1 outperforms the other variants on both collections, suggesting that reading the selectors from low-memory to high-memory is more efficient that reading the other way around. Variant 3 generally outperforms Variant 2, suggesting that the `case` statement fall-through overheads are measurable. Variant 4 is effective on the URL-ordered collection but not on the crawl-ordered collection, suggesting that pre-fetch is only sometimes worthwhile. In both cases, the PFor codecs did not perform as well as the others.

The Friedman test shows a significant difference between the codecs ($p < 0.001$). The Wilcoxon-Nemenyi-McDonald-Thompson post hoc analysis shows that QMX is significantly ($p < 0.05$) faster than all the others except TurboPackV (crawl order) and Variant 1 (URL order).

If efficiency is important then QMX is the best option. The changes made to prevent overflow and to reduce the size of the index also add decoding overheads. However, if effectiveness is more important then TurboPFor is the best.

Table 2: Tournament matrix showing how many times the codec in column i results in a no-slower ($\leq$) search time than the codec in row j for crawl-ordered .gov2 using topics 701-850. Larger is better.

	QMX	v1	v2	v3	v4	OPT	TPackV	TPfor
QMX	-	17	17	26	14	9	42	8
v1	133	-	51	64	39	12	105	20
v2	133	99	-	82	58	10	123	32
v3	124	86	68	-	39	17	110	21
v4	136	111	92	111	-	29	129	66
OPT	141	138	140	133	121	-	144	137
TPackV	108	45	27	40	21	6	-	7
TPfor	142	130	118	129	84	13	143	-
TOTAL	917	626	513	585	376	96	796	291

Table 3: Tournament matrix showing how many times the codec in column i results in a no-slower ($\leq$) search time than the codec in row j for URL-ordered .gov2 using topics 701-850. Larger is better.

	QMX	v1	v2	v3	v4	OPT	TPackV	TPfor
QMX	-	55	17	29	5	1	19	2
v1	95	-	14	7	47	1	10	2
v2	133	136	-	81	93	3	47	4
v3	121	143	69	-	98	3	46	4
v4	145	103	57	52	-	2	43	7
OPT	149	149	147	147	148	-	150	145
TPackV	131	140	103	104	107	0	-	1
TPfor	148	148	146	146	143	5	149	-
TOTAL	922	874	553	566	641	15	464	165

7. THE SPACE TIME CONTINUUM

Section 6.3 examined how the user experiences the search engine – a direct query-for-query comparison. An alternative way to examine the utility of the codecs is the system view. That is, if a given codec is chosen, how much storage space is needed and how much CPU will be consumed?

This question is examined using the very same JASS runs seen in Section 6.3. However, rather than examining on a query-by-query basis, this section examines the total overall CPU usage and the size of the postings file.

The median of total search time (of 25 runs) was plotted against the index size in Figure 8 (with a non-zero origin) when the collection is in crawl order and in Figure 9 when in URL order (again, a non-zero origin). These figures show that TurboPackV, TurboPFor, and Variant 1 (URL order) or Variant 2 (crawl order) lie on the Pareto frontier – these codecs offer the best choice between search latency, index size, and the tradeoff of the two. It can also be observed from these figures that the QMX variants result in index sizes that are not substantially different from each other, TurboPackV can be smaller, and that the PFor codecs result in small indexes that take longer to process.

Care should be taken when drawing latency conclusions from these two graphs as the total search time is dominated by a small number of slow queries. This can be seen at the left hand side of Figure 6 and Figure 7, which show (on a log / linear scale) that a small number of queries take a substantial amount of the time. For example, in crawl order the quickest query using QMX took ≈4ms, the median took

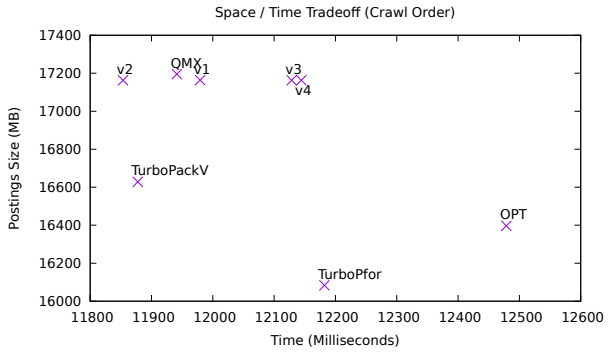

Figure 8: Space / time tradeoff when tested on TREC topics 701-850 on .gov2 in crawl order.

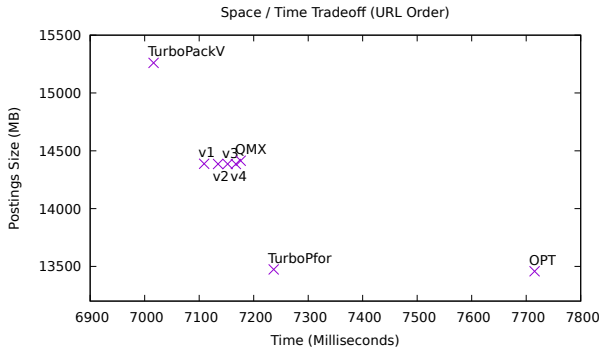

Figure 9: Space / time tradeoff when tested on TREC topics 701-850 on .gov2 in URL order.

≈57ms and the slowest took ≈339ms (88 times longer than the quickest and 6 times longer than the median). These high latency searches dominate the total execution time.

8. DISCUSSION AND CONCLUSIONS

This investigation compared OPTPFor with several SIMD-based codecs including TurboPFor, TurboPackV, and QMX. It introduced four variants of QMX, two supposed effectiveness improvements and two efficiency improvements. Experiments on public data in document order showed little difference between the codecs. Experiments in the JASS open-source score-at-a-time search engine with impact-ordered indexes suggest little difference between the codecs with QMX slightly outperforming the others.

This work has some limitations, prior work has shown that results depend on hardware [16], document collection, and processing strategy – none of which were examined. SAAT does not use skipping so Elias-Fano [10] was not tested, nor were previously compared codecs such as variable-byte encoding.

In conclusion, no evidence was found to suggest that the QMX variants improved efficiency, even though they improved effectiveness. Although PFor codecs were shown to take less space than the others, they also performed unfavorably against QMX and variants when decoding. TurboPackV, a bin-packer, was less effective, but more efficient on longer queries and unsorted data.

9. REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-Space Ranking with Effective Early Termination. In *SIGIR 2001*, pages 35–42.

[2] V. N. Anh and A. Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Inf. Retr.*, 8(1):151–166, 2005.

[3] V. N. Anh and A. Moffat. Index Compression Using 64-bit Words. *Softw. - Pract. Exp.*, 40(2):131–147, 2010.

[4] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient Query Evaluation Using a Two-Level Retrieval Process. In *CIKM 2003*, pages 426–434.

[5] M. Crane, A. Trotman, and R. O'Keefe. Maintaining Discriminatory Power in Quantized Indexes. In *CIKM 2013*, pages 1221–1224.

[6] S. Ding and T. Suel. Faster Top-k Document Retrieval Using Block-Max Indexes. In *SIGIR 2011*, pages 993–1002.

[7] D. Lemire and L. Boytsov. Decoding Billions of Integers Per Second Through Vectorization. *Softw. - Pract. Exp.*, 45(1):1–29, 2015.

[8] J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. Macdonald, and S. Vigna. Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge. In *ECIR 2016*, pages 408–420.

[9] J. Lin and A. Trotman. Anytime Ranking for Impact-Ordered Indexes. In *ICTIR 2015*, pages 301–304.

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

[11] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered Document Retrieval with Frequency Sorted Indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, 1996.

[12] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. *Proc. 3rd Text Retr. Conf.*, pages 109–126.

[13] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of Inverted Indexes for Fast Query Evaluation. In *SIGIR 2002*, pages 222–229.

[14] F. Silvestri. Sorting out the Document Identifier Assignment Problem. In *ECIR 2007*, pages 101–112.

[15] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-Based Decoding of Posting Lists. In *CIKM 2011*, pages 317–326.

[16] A. Trotman. Compression, SIMD, and Postings Lists. In *ADCS 2014*, pages 50–57.

[17] A. Trotman. Compressing Inverted Files. *Inf. Retr.*, 6(1):5–19, 2003.

[18] A. Trotman, M. Crane, and X.-F. Jia. Towards an Efficient and Effective Search Engine. In *SIGIR 2012 Workshop on Open Source Inf. Retr.*, pages 40–47.

[19] A. Trotman, X.-F. Jia, and M. Crane. Managing Short Postings Lists. In *ADCS 2013*, pages 113–116.

[20] S. Vigna. Quasi-Succinct Indices. In *WSDM 2013*, pages 83–92.

[21] H. Yan, S. Ding, and T. Suel. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *WWW 2009*, pages 401–410.

[22] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE 2006*, page 59.