FEMTO: Fast Search of Large Sequence Collections

Michael P. Ferguson

michaelferguson@acm.org

Abstract. We present FEMTO, a new system for indexing and searching large collections of sequence data. We used FEMTO to index and search three large collections, including one 182 GB collection. We compare the performance of FEMTO indexing and search with Bowtie and with Lucene, and we compare performance with indexes stored on hard disks and in flash memory. To our knowledge, we report on the first compressed suffix array storing more than 100 GB. Even for the largest collection, most searches completed in under 10 seconds.

Keywords: FM-index, document retrieval, external memory, regular expression, compressed suffix array

1 Introduction

The FM-index of Ferragina and Manzini [8,9,10] is a remarkable structure for string searching because it supports $O(m)$ search, where m is the length of the search pattern, while typically using less space than the original data. Furthermore, the FM-index for n bytes of data can be constructed in $O(n)$ time. However, in practice, an FM-index - as well as other compressed suffix arrays - must compete with more established techniques, such as Boyer-Moore [3] or *grep*, which perform an $O(n)$ scan of the data.

There are three problems with the typical compressed suffix array that make it a poor choice for many information retrieval problems. The first problem is that most implementations are not capable of handling very large collections. They assume that the index fits into main memory or that 4 bytes are sufficient to store offsets. Furthermore, a straightforward implementation of the FM-index for hard disks will have poor performance since each search step requires millisecond-long random access to the index.

The second problem is that, when compared to a tool like *grep*, compressed suffix arrays offer limited flexibility. The implementations only return counts or offsets. At the same time, many uses of an index only need a list of matching documents. In addition, compressed suffix arrays only allow string search; there is no support for regular expressions.

The third problem is that the indexing procedure is extremely slow for large collections. Collections that cannot be indexed in main memory must be indexed

on disk. Dementiev et al. demonstrated external-memory suffix array construction at 9.88 microseconds/byte, or about 100 KB/second [6]. Compare that rate with the speed of scanning from disk - easily about 70 MiB/second [1].

We believe that we have made progress in solving these three problems. We have created a system called FEMTO: the **F**M-index for **E**xternal **M**emory with **T**hroughput **O**ptimizations. It is an FM-index with some improvements to make external memory operation faster. Our goal is to create an FM-index that can compete better with the scanning approach for large, archival collections. FEMTO minimizes seeks to perform well in external memory, supports more flexible usage including document retrieval and regular expression search, and constructs indexes in parallel and in external memory at rates exceeding 1 MiB/second with 4 machines.

2 Background

We summarize the operation of an FM-index as in [10]. First, to index any number of documents, concatenate the documents together to create an n-byte text string T. The next step in the indexing process is to perform the Burrows-Wheeler transform (BWT) on T [4]. Conceptually, form an n by n matrix from all the rotations of the n-byte input string, and sort the rows of this matrix lexicographically. This sorted matrix is called M. The BWT of the original n-byte string is the last column of M, which is called L. The original input T can be reconstructed from the L column using the *last-to-first column mapping* [4]. This mapping, denoted LF, provides a mechanism for stepping backwards in the text. That is, if row r begins with $T[i..]$, $LF(r)$ will give the index of the row beginning with $T[i-1..]$. The LF mapping can be computed entirely from information in the L column. In particular, $LF(i) = C[L[i]] + Occ(L[i], i) - 1$, where C stores, for each character, the number of characters less than that character in the entire input, and $Occ(ch, i)$ is the number of times the character ch appears at or before row i in the L column.

Note that in practice, the BWT is computed by first building the suffix array of the input string. It is straightforward to compute the L column from the suffix array [4].

An FM-index stores the L column, the C array, information to help compute $Occ(L[i], i)$, and - for some fraction of *marked* rows - offset information. The FM-index uses these structures to support two kinds of searches: *count* and *locate*. Given a pattern of m bytes, the *count* search works in $O(m)$ steps to find the rows in the conceptual matrix M that begin with that pattern. The *count* search simply returns the number of matching rows. A *locate* search finds the offset for each matching row. Both of these methods find the range of rows beginning with a particular pattern with the *backward_search* method of Ferragina and Manzini [10], which goes through the pattern backwards, maintaining the range of matching rows, and uses a variant on the LF-mapping. To find the offset for

[1] In this paper, we use the IEC binary prefixes: MB and GB mean 10^6 and 10^9 bytes; MiB and GiB mean 2^{20} and 2^{30} bytes.

a particular row, an FM-index applies the LF-mapping until a marked row is reached [10]. We call each application of the LF-mapping a *backward step*.

3 The FEMTO Index

The FEMTO index is based on a combination of methods, including the FM-index [8,10] and the Compressed Suffix Array [13]. Like other indexes, the FEMTO index is made up of blocks. A FEMTO index has a single header block and many data blocks. The header block stores the C array and the number of occurrences of each character before the start of each data block. Each data block contains buckets storing the L column and marking information as well as the number of times each character appears before each bucket in the block. The L column data is stored in a wavelet tree inside each bucket. This wavelet tree structure provides a fast way of computing $Occ(ch, i)$. We use our own implementation of the wavelet tree described by Grossi, Gupta, and Vitter [13]. FEMTO marks character offsets i with $i \pmod k = 0$, for some parameter k known as the *mark period*. FEMTO uses a succinct dictionary [13] to record which rows are marked and stores the offsets for marked rows in a separate array. Lastly, for each chunk of h rows, FEMTO stores the corresponding set of document numbers.

3.1 High Throughput with External Memory

We assume that at any given time, FEMTO is processing several queries in parallel. Each of these queries can be expressed as a sequence of *requests*. These requests find a small unit of information - such as $Occ(L[i], i)$. In order to service one of these requests, the system must read the appropriate bucket from disk. Once the bucket is loaded, the system can service any number of requests for that bucket with no extra I/O cost. The search system keeps the requests in a tree structure sorted by row. That way, it visits requests in row order in order to avoid needing to re-read parts of the index. In addition, FEMTO processes the sorted requests in multiple threads using a work-stealing strategy.

This method is new as suggested; compressed suffix array implementations typically assume that they will process a single query at a time. Two implementations use related techniques to reduce the search time for a single query, as in Grossi and Vitter [14] for reporting occurrences and in the FM-index version 2 for doing count queries with the mark character inserted [9].

Sorting requests provides an enormous reduction in the number of block cache misses. After enabling request sorting during development, we observed about a 5000-fold reduction in the amount of I/O to perform 100,000 queries.

3.2 Bi-directional Locate

During a locate operation, FEMTO steps both forward and backward at the same time to find a marked row. Backward step works in the usual manner,

but forward step uses binary search on the header block and on a data block. These forward and backward steps are continued independently until a marked row is located. As a row range is stepped forward or backward, it becomes more and more spread across the index. By stepping both forwards and backwards, FEMTO reduces this spreading, leading to better cache performance. Note that the complexity of this locate procedure remains $O(k)$ index operations, assuming that every k^{th} character is marked. Early experiments showed that bi-directional locate incurs about half the I/O of backward steps.

3.3 Improving Document Search Time

FEMTO employs a new scheme to improve the performance of queries for matching documents, which we call *locate_documents* queries. These queries do not need to return the locations of matches within the documents - only the set of matching documents. To improve these searches, FEMTO divides the index into chunks of h rows. For each chunk, it stores a list of documents present in that range of rows. When reporting the documents contained in an arbitrary range of rows, FEMTO does the normal locate procedure for the results in the partial first and last chunks. For each full chunk, it reads the set of matching documents directly. The list of document numbers stored in each chunk is compressed using standard information retrieval methods as in [21].

Assuming that each chunk can be read in a single block operation, the number of block operations to locate the documents represented by r matching rows is $O(r/h + hk)$, where h is the chunk size and k is the mark period. This is an improvement over the original $O(rk)$ search time, and in practice it can mean 10-200x speedup when returning a large number of results.

These measures to improve document search time, especially when reporting a very large number of documents, are important for applications that require the entire list of matching documents, such as Boolean query processing. An alternative approach would be to store the document numbers for each row in a wavelet tree as described in [12]. In addition, Culpepper et al. extend that result in order to directly compute the top k results, ranked by term frequency [5]. Although we were not able to perform a direct comparison, we believe that the wavelet tree approaches create larger indexes - on the order of $3n$ bytes - while the FEMTO indexes range from $n/2$ to $2n$ bytes.

3.4 Regular Expression Search

We have developed a regular expression search method for the FM-index which is an analogue of the trie automaton search of Baeza-Yates et al. [1]. First, take a regular expression and reverse it - e.g. *ab(c*d |ef)g* becomes *g(dc* |fe)ba*. Then, compile it to a nondeterministic finite automaton (NFA) using Thompson's method [20]. Next, simulate the NFA on the compressed suffix array. The simulation operates on a mapping from ranges of rows to automaton states. Each step of the algorithm proceeds in a similar manner to the *backward_search*; take

a backwards step for each character on a transition from a current NFA state and then add the resulting range of rows and state to the mapping.

```
simulate_nfa :
  add the entry ([0, n-1] -> the set of start states) to mapping
  while( the mapping is not empty ) :
    pop an entry ([first, last] -> nfa_states) from the mapping
    for every character ch reachable from nfa_states :
      new_first = C[ch] + Occ(ch, first - 1)
      new_last = C[ch] + Occ(ch, last) - 1
      new_states=states reachable from nfa_states after reading ch
      add ([new_first, new_last] -> new_states) to the mapping,
         reporting a match if a final state is set
```

This algorithm has some desirable properties. First, exact string search proceeds as in *backward_search*. Second, this algorithm is able to re-use a partial match; for example when searching for $(abc|def)xyz$, the algorithm only computes the range of rows matching xyz once. Note that this algorithm performs at least z index operations, where z is the length of the longest match; thus it is $\Omega(z)$. Furthermore, some regular expressions, such as the infinite wildcard $.*$, will cause the algorithm to visit every row in the index. Although this algorithm is not appropriate for all regular expressions, it performs well with simple regular expressions. Lastly, it is possible to modify this algorithm to support approximate queries by simulating approximate NFA search, although space does not allow us to describe it in detail here. Our implementation of approximate NFA search is based upon the approximate regular expression search of Wu and Manber [22], but interacts with the index just like *simulate_nfa* above.

3.5 Parallel External Memory Index Construction

We developed a parallel external-memory suffix array construction system, based upon the difference cover algorithm [16]. Our independent implementation combines the ideas of [17] and [6]. We observed linear performance indexing the English corpus from the Pizza & Chili website [11]. In a 10-gigabit Ethernet cluster using 3 disks per node, we measured 0.45 MiB/s on a single machine, 1.61 MiB/s on four machines, and 2.7 MiB/s on eight machines.

There are many notable alternative ways to construct an FM-index. First, there are direct index construction schemes, such as [15] and [19], that require memory space for the resulting compressed index. Also, Ferragina et al. present in [7] a many-pass method to compute the BWT in external memory. Since this many-pass method has I/O complexity $O(n^2/(MB \log n))$, it is only appropriate for collections that are not much larger than main memory. Lastly, [2] shows how to efficiently index many small documents, but their method works with fixed-size documents and is quadratic in the document size - and so not applicable to a large, varied collection. Nonetheless, we believe that future work along these lines may lead to a generally useful direct-to-BWT external memory scheme.

4 Experimental Methods

We report experiments with three different large data sets. The first is a collection of bacterial sequence data from the National Center for Biotechnology Information (NCBI) Genomes collection - totaling 3.8 GB. The second collection is 43 GB containing all textual documents from Project Gutenberg collection. The final collection is all 182 GB of sequence data from the NCBI Genomes collection. To our knowledge, this is the first time that compressed suffix arrays have been constructed for such large collections.

To construct our Gutenberg dataset, we created a local mirror of the Project Gutenberg collection (`http://www.gutenberg.org`) in November 2011 and then we selected only documents that the *file* command reported as text. We indexed this collection with Apache Lucene Core 3.5.0 (`http://lucene.apache.org`) for comparison. We modified the Lucene searching demonstration to give performance timing and to only ever return 10 results. When searching for a pattern containing multiple words, we used the phrase query mechanism in Lucene.

To construct the Genomes collection, we downloaded all .fna, .fa, .fasta, .seq, .fsa, .ffn, .faa, .frn, and gzip compressed versions of those same files, in November 2011, from ftp://ftp.ncbi.nih.gov/genomes. We then removed redundant compressed files as well as symbolic links and ARCHIVE directories and uncompressed the compressed files.

We constructed the Bacteria collection by taking files from the Bacteria subdirectory of our Genomes collection, up to 3.76 GB. We limited the size in order to compare with Bowtie2 (available at `http://bowtie-bio.sourceforge.net`) [18]. Bowtie2 cannot currently create an index for more than 4 GiB of input data. We used Bowtie 2.0.0-beta3. When searching with Bowtie, we used the arguments *–mm –sam-nosq –end-to-end -M 0 -N 0 -L 500 -i L,0,500* in addition to the *-x* argument to specify the index and the *-c* argument to specify a search pattern. We used these arguments in order to request exact matches and to request that the index be memory mapped instead of read in at program start.

FEMTO indexes were constructed with a block size of 128 MiB, a bucket size of 1MiB, a document chunk size of $h = 2048$, and a mark period of $k = 20$.

Because we are measuring external memory performance, we flushed the Linux page cache between each experiment and performed an operation to clear out hardware RAID and disk buffers. Then we performed a search for an unrelated term in order to make sure that any index headers or program images were in cache. Finally, we performed the measured search. Thus, these search performance numbers represent the time to get results when the search program is cached but almost all of the index is not.

We performed experiments on two different systems. The first has an 8-disk hardware RAID-6 array, 8-cores of Intel Xeon X5355 at 2.66 GHz, and 16 GiB of memory. The second system has 8 flash memory devices configured in a hardware RAID-5 array, 24 cores of Xeon X5670 at 2.93 GHz, and 24 GiB of memory.

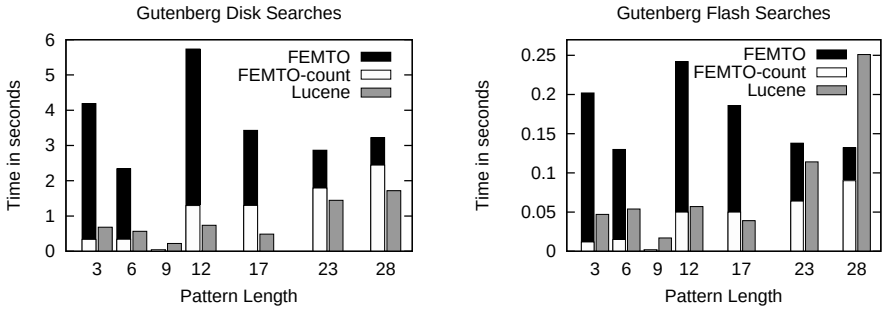

Fig. 1. Gutenberg (42.94 GB) indexed search speeds. Note that the time to search with *grep* is 559 s on disk, or 221 s on flash memory. Each query is limited to 10 results.

4.1 Results

FEMTO, Lucene, and Bowtie2 all create indexes for completely different purposes: FEMTO is meant to support large, archival indexing and search of any sequence data; Lucene indexes word data; and Bowtie2 was created for sequence alignment. Nonetheless, it is useful to compare these systems to understand, as a practical matter, in what situations a system such as FEMTO should be used. Table 1 summarize index size and indexing time.

Table 1. Data set and index construction information. Bacteria was indexed with Bowtie2, Gutenberg with Lucene. Genomes indexed with 7 machines. FEMTO indexes sizes show optional document chunk structure; see Section 4.3.

Corpus Information			Indexing time and index size	
	Size	# Docs	Bowtie2/Lucene	FEMTO
Bacteria	3.76 GB	1896	16940 s 5.26 GB	7572 s 1.91 GB - 2.34 GB
Gutenberg	42.93 GB	94471	5133 s 14.84 GB	108025 s 19.83 GB - 40.87 GB
Genomes	182.43 GB	23242528	-	158434 s* 87.62 GB - 296.68 GB

Figures 1-3 show external memory search performance. Figure 1 shows that FEMTO is not as fast as Lucene, but it is not far behind. Most patterns were one word and returned exactly 10 matches. Note that the pattern of length 9 had no matches, the pattern of length 28 had only 3 matches, and that the 23-long pattern is two words and the 28-long pattern is three words. Lucene search time depends on the number of words in the search - hence it increases for the last two - while a FEMTO search depends more directly on the number of characters. Observe that the time to identify matching rows - shown in the figure in the white area labeled FEMTO-count - increases as the pattern size grows. Searches backed by flash memory are noticeably faster for both systems, and benefit FEMTO more than Lucene, since FEMTO is limited by disk seek speed. Lastly, both systems are 100-1000x faster on disk than *grep* for this collection.

As Figure 2 shows, FEMTO is competitive with Bowtie2 for single-query exact-match searches. Bowtie2 was not designed for this kind of external memory operation, but it is a BWT-based index like FEMTO. Bowtie2 performs exact search about as quickly as FEMTO. For most of these searches, FEMTO is slightly faster. Note that for both of these systems, the search time increases as the pattern length grows. Lastly, for a collection of only 4GB, a scan of the collection with *grep* is competitive with the time to do a search with either of these indexes if the index is stored on disk; we measured *grep* to take 11.5 s which is about the time it takes to run a 256-character query.

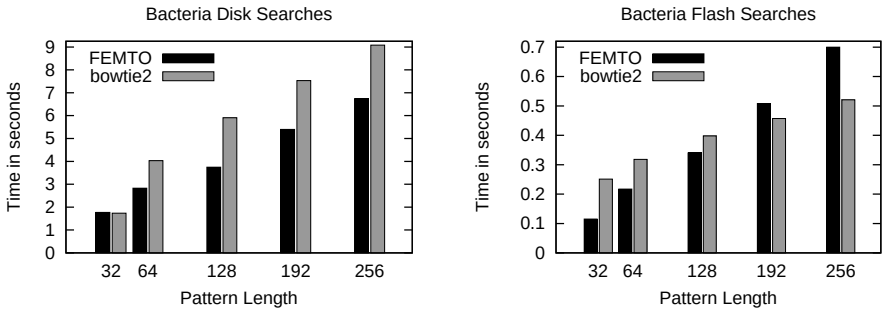

Fig. 2. Bacteria (3.76 GB) indexed search speeds. Note that the time to search with *grep* is 11.5 s on disk, or 6.12 s with flash memory. Each query returned only one result.

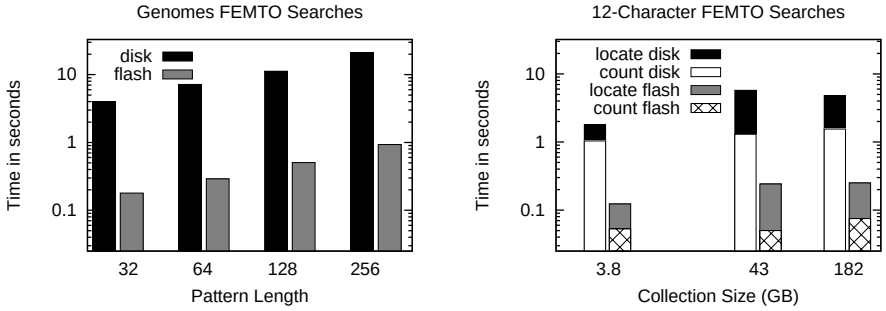

Fig. 3. Genomes (182.43 GB) indexed search speeds. Note that the time to search this collection with *grep* is 1246 s on disk, or an estimated 493 s with flash memory. Each query returned one or two results. Note that these plots are on log-log scale.

FEMTO scales to very large index sizes while still offering fast searches. Only FEMTO was able to construct a sequence index for the Genomes collection. As the left-hand side of Figure 3 shows, sequence queries still complete in seconds

and take time proportional to the length of the query. Observe on the right-hand side of Figure 3 that FEMTO search performance is approximately constant over the range of collection sizes measured here - from 3.76 GB to 182 GB.

To demonstrate the capabilities of sorting requests, we also measured the performance of serial and parallel count queries over randomly selected length-12 patterns in the Gutenberg data set. See Figure 4. The request sorting mechanism offers better throughput on disk when processing many count queries simultaneously, but does not offer much of an advantage when the index is in memory.

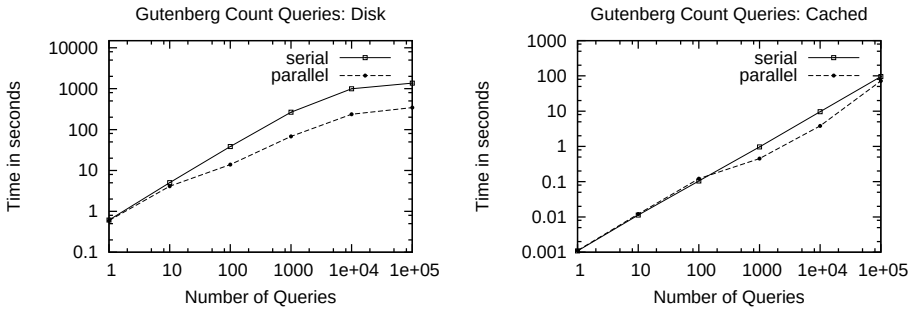

Fig. 4. Gutenberg (42.94 GB) indexed search speeds for many count queries. Note that these plots are on log-log scale.

4.2 Approximate Search

The FEMTO query "APPROX constitutional" searches for all sequences within edit distance 1 of "constitutional". That query took 37.6 s on the disk system and 2.41 s on the flash system. The analogous Lucene query, "constitutional~" took 103 s on the disk system and 96 s on the flash memory system. Thus, FEMTO is quite a bit faster than Lucene at approximate search. Approximate search is still fast even for our largest index - a 32 character search against the Genomes index took 53.2 s with the disk system and 3.34 s with flash memory.

4.3 FEMTO Index Space

Table 2 shows how the space of each FEMTO index was used. Each column shows index structures that support different features. Using only the data in the *BWT & Occs*, the index supports finding the range of matching rows for a pattern. Adding in the *Offsets* allows the index to report back the offsets of matching rows, and the mark period (20) represents a time-space trade-off for that operation. Finally, adding in the *Doc Chunks* allows the index to rapidly report a large number of matching documents for large ranges of rows. This kind of operation is important for searches which require follow-on processing,

Table 2. FEMTO index statistics. Percentages show proportion of collection size.

Collection	BWT & Occs	Offsets	Doc Chunks	Total
Bacteria - 3.76	0.96 (25%)	0.94 (25%)	0.44 (11%)	2.34 (62%)
Gutenberg - 42.93	7.74 (18%)	12.00 (28%)	21.04 (49%)	40.87 (95%)
Gutenberg' - 42.93	7.74 (18%)	12.00 (28%)	2.4594 (6%)	22.21 (52%)
Genomes - 182.43	32.97 (18%)	51.13 (28%)	209.06 (115%)	296.68 (163%)

such as Boolean queries. Note that without the document chunk information, each of these indexes would be about 50% the size of the original collection. The document chunks did not compress well in the larger collections, which have orders of magnitude more documents than the chunk size of $h = 2048$. To investigate further, we created a Gutenberg index with a $h = 131072$, listed as Gutenberg' in the table, that achieves much better compression. Thus, for consistent compression, h should depend on the number of documents.

4.4 Reporting Results

We measured the speed of finding the offset or document for a varying number of rows in the Gutenberg index. These queries really show the ability for FEMTO to minimize I/O operations. See Figure 5. With flash memory, we observed locate speeds up to 7000 results/second and document locate speeds up to 90,000 results/second. On disk, locate performed at up to 800 results/second and document locate operated at up to 70,000 results/second. We believe that the decrease in locate performance around 10,000 results comes from the working set no longer fitting into main memory. Also, note that for 10 million results, the query processing data structures occupy 8 GiB of memory.

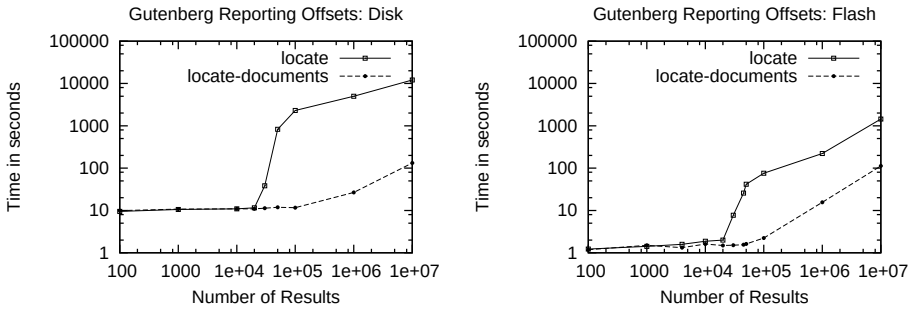

Fig. 5. Gutenberg (42.93 GB) locate speeds. Note that these plots are on log-log scale.

5 Conclusion

We have demonstrated the feasibility of using an improved FM-index for large sequence and document retrieval problems. We have demonstrated interactive search times even with our largest collection. This index structure is very quick to search - especially with flash memory - and supports regular expression and sequence queries.

Availability. FEMTO source code is available at

`https://github.com/femto-dev/femto` .

Acknowledgments. We would like to thank Aaron Marcus from Virginia Tech and Katherine Gibson from the University of Maryland Baltimore County for their help implementing regular expression search; and John Dorband, Yaacov Yesha, and Nancy Walia from the University of Maryland Baltimore County for their help with parallel suffix array construction.

References

1. R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, 1996.
2. M. Bauer, A. Cox, and G. Rosone. Lightweight bwt construction for very large string collections. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching*, volume 6661 of *LNCS*, pages 219–231, 2011.
3. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
4. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
5. J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. 18th Annual European Symposium on Algorithms*, volume 6347 of *LNCS*, pages 194–205, 2010.
6. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. In *ALENEX/ANALCO*, pages 86–97. SIAM, 2005.
7. P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, July 2012.
8. P. Ferragina and G. Manzini. An experimental study of a compressed index. *Information Sciences*, 135(1-2):13–28, 2001.
9. P. Ferragina and G. Manzini. Fm-index version 2 web page, 2005. Available at `http://roquefort.di.unipi.it/~ferrax/fmindexV2/index.html`.
10. P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
11. P. Ferragina and G. Navarro. Pizza & chili website, 2006. Available at `http://pizzachili.dcc.uchile.cl` or `http://pizzachili.di.unipi.it`.
12. T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th Symposium on String Processing and Information Retrieval*, volume 5721 of *LNCS*, pages 1–6, 2009.

13. R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004.
14. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, 2000.
15. W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, Mar. 2007.
16. J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.
17. F. Kulla and P. Sanders. Scalable parallel suffix array construction. *Parallel Comput.*, 33(9):605–612, 2007.
18. B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.
19. J. Sirén. Compressed suffix arrays for massive data. In *Proc. 16th Symposium on String Processing and Information Retrieval*, volume 5721 of *LNCS*, pages 63–74, 2009.
20. K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.
21. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2 edition, 1999.
22. S. Wu and U. Manber. Fast text searching: allowing errors. *Commun. ACM*, 35:83–91, October 1992.