

# Intersection in Integer Inverted Indices

Peter Sanders\*

Frederik Transier†

## Abstract

Inverted index data structures are the key to fast search engines. The predominant operation on inverted indices asks for intersecting two sorted lists of document IDs which might have vastly varying lengths. We compare previous theoretical approaches, methods used in practice, and one new algorithm which exploits that the intersection uses small integer keys. We also take different data compression techniques into account. The new algorithm is very fast, simple, has good space efficiency, and is the only algorithm that performs well over the entire spectrum of relative list length ratios.

## 1 Introduction

One of the main (positive *and* negative) effects of the information society is that we are inundated with textual data in the form of emails, papers, web pages, annotations to data base entries, . . . While we can cheaply store these huge amounts of text, we often cannot afford to scan them exhaustively when we are searching for something. Therefore, *full text indices* are a crucial data structure. Because of its simplicity, versatility, and space efficiency, the *inverted index* is the most widely used full text index data structure: Each of the  $U$  documents is identified using an integer document ID (docID for short) in the range  $\mathcal{U} := 0..U-1$ . For each term  $t$ , the inverted index stores a sorted list of docIDs  $L(t)$  which specifies the documents containing this term. For space efficiency, these list are usually compressed. One common feature of successful compression algorithms is  $\Delta$ -*encoding*— they encode only (or mostly) the differences between subsequent docIDs.

The most common type of query specifies a small set of terms  $t_1, \dots, t_k$  and asks for all documents that contain all these terms. The task is now to intersect the sets (represented by)  $L(t_1), \dots, L(t_k)$ . In most cases, it is best to do this using pairwise intersection. We start with the two shortest lists, intersect their intersection with the third-shortest list, etc. Thus, it suffices to study the problem of intersecting two lists  $M$  and  $N$  with  $m := |M| \leq n := |N|$ .

Of course, this problem has been studied extensively in the past. If  $m \approx n$ , a simple and efficient solution is *zipper* — binary merging by scanning both lists. Zipper needs time  $\mathcal{O}(m+n)$  and is also quite cache efficient. However, zipper is wasteful if  $m \ll n$ . In the comparison based model, the asymptotically best approach is a clever use of binary search which needs time  $\mathcal{O}(m \lg \frac{n}{m})$  [1, 2, 3]. Several papers have looked at improving the constant factors, exploiting regularities in the input or devising average case efficient methods e.g. [4, 5]. The focus of these papers is on reducing the number of comparisons for the general case of intersecting  $k$  sequences at once. Unfortunately, many of these papers study comparisons rather than running time. But the considerations below illuminate that comparisons are not a good predictor of running time.

In practice, binary merging based methods have several disadvantages. First, in real implementations, binary search based algorithms beat zipper only when  $n > 20m$  [3] although at  $n \approx 20m$  binary merging based algorithms performs several times fewer comparisons than zipper. Furthermore, binary search conflicts with compression using  $\Delta$ -encoding which enforces sequential scans of the input. A practical compromise cuts the lists into  $\Delta$ -encoded pieces and stores additional information that allows skipping pieces that do not contain docIDs from  $M$ . Note that this compromise will reduce the possible advantage over zipper when  $m \ll n$ . Experiments with simple variants of this approach are reported in [6]. However, we are not aware of running time studies that use both space efficient representation and asymptotically optimal algorithms. In Section 2 we explain the algorithms that we use in our study.

One subject of this paper is to engineer practical set intersection algorithms that are *not* (purely) comparison based but exploit that docIDs are small integers and that we are even free to choose the mapping of documents to docIDs. We are not aware of papers with this specific purpose. Of course, we could also represent lists by space efficient hashing based dictionaries which would yield execution time  $\mathcal{O}(m)$ . However, we are not aware of practical variants of the above approaches which are comparably space efficient as  $\Delta$ -encoding and are similarly fast as zipper when  $m \approx n$ . For example, even if we would relax the requirement of

\*Universität Karlsruhe (TH), 76128 Karlsruhe, Germany, sanders@ira.uka.de

†SAP AG, 69190 Walldorf, Germany, transier@ira.uka.de

space efficiency, we cannot use conventional hash tables since this would incur a large number of cache misses during intersection.

Therefore, we will consider a setting where docIDs are chosen randomly. This can be implemented using a *pseudorandom permutation*  $\pi$  that assigns  $\pi(i)$  to the  $i$ -th document in the data base. In Section 3 we explain our algorithm *lookup*. We show that lookup runs in expected time  $\mathcal{O}(n)$  and needs space close to the information theoretic minimum of  $\lg \binom{U}{n}$  bits for storing a set of  $n$  random integers from  $\mathcal{U}$ .

In Section 4 we compare efficient implementations of previous approaches and our algorithm using real world inputs. Our focus is on a good compromise between speed and space consumption that is particularly relevant for main memory data bases. This scenario is from an application at SAP: The TREX engine combines full-text search with the search on structured data supporting basic operations of relational databases. Typically, its indexes maintain a huge amount of business objects in main memory. But the text fields within these business objects are small and contain only a few words or sentences. Examples are memos or product descriptions. However, we believe that our results are equally relevant for the more traditional applications with larger documents. Even if the full text index resides on disk, it is quite likely that internal processing time plays an important role since many lists will be cached and since using parallel disks we can get I/O bandwidths not too far away from main memory bandwidth even on low cost machines [7]. Section 5 summarizes the results and discusses possible generalizations and further research.

## 2 Comparison Based Intersection

### 2.1 Basic Compression Techniques

**Uncompressed Format.** The simplest representation for a list  $N$  is a sorted array of  $n$  docIDs each stored in a machine word of its own. Since this is also the most efficient representation for the smaller list in an intersection operation, this uncompressed format is also the *output* format of all intersection algorithms discussed in this paper. A search engine using our approach will therefore have to distinguish between the case of a compressed and uncompressed smaller list respectively.

**$\Delta$ -Encoding, Bit-Compression.** A straight forward approach to compression is *bit-compression*—use only the bits actually needed to represent the integers under consideration. Applied to an uncompressed list of docIDs, this reduces the space consumption of a list  $N$  to  $n \lceil \lg U \rceil$  bits rather than  $n$  machine words. The first nontrivial step is  $\Delta$ -encoding—we store  $\langle d_1, d_2 - d_1, d_3 -$

$d_2, \dots, d_n - d_{n-1} \rangle$  to represent the list  $N = \langle d_1, \dots, d_n \rangle$ . If the elements of a list  $N$  were uniformly spread over the range  $0..U - 1$ , bit-compressed  $\Delta$ -encoding could reduce space consumption to  $n \lg \frac{U}{n}$  bits. Of course, differences can get as big as  $U - n + 1$ . Thus, in the worst case,  $\Delta$ -encoding with bit-compression is no better than bit-compression alone.

**Variable Bit Length Encoding** addresses the above mentioned drawback of bit-compression. Suppose we could encode an integer  $d$  using  $\lg d$  bits. Then  $\Delta$ -encoding would need  $\sum_{i=1}^n \lg d_i - d_{i-1} \leq n \lg \frac{U}{n}$  bits (defining  $d_0 := 0$ ).<sup>1</sup> Of course, actual encoding schemes cannot reach this overoptimistic situation. There is a multitude of encoding schemes with different tradeoffs between space consumption and (de)coding cost. One compromise that has turned out to be useful is *escaping*. Consider a parameter  $b$ . A  $k$ -bit integer is split into  $K = \lceil \frac{k}{b-1} \rceil$  pieces of  $b-1$  bits each and encoded into  $K$  blocks of  $b$  bits each. All but the last block are marked by a 1 in the most significant bit.

### 2.2 Algorithms

**Zipper.** The simplest intersection algorithm scans both lists as in a binary merging operation. It needs time  $\mathcal{O}(n + m)$  with a quite small constant factor. Hence, zipper is best if the lists have about equal size. Zipper harmonizes well with  $\Delta$ -encoding and any compression scheme. However, when lists are stored in main memory, the decompression overhead outweighs the gains in memory access time.

**Variants of Binary Search.** A simple, (suboptimal) algorithm that is sublinear in the size of the longer list  $N$  scans the shorter list  $M$  and locates each element  $d$  of  $M$  in  $N$  using binary search. This takes time  $\mathcal{O}(m \log n)$  with constant factors worse than in algorithm zipper. Binary search can be improved to an asymptotically optimal  $\mathcal{O}(m \lg \frac{n}{m})$  by exploiting that  $M$  is sorted [1]. If  $M[i-1]$  was located at position  $j$  in  $N$ , i.e.,  $j = \max \{k : N[k] \leq M[i-1]\}$ , then  $M[i]$  must be located at a position  $j' > j$  in sequence  $N$ . If  $j' = j + \Delta_i$ , we can find an upper bound on  $\Delta_i$  in time  $\mathcal{O}(\log \Delta_i)$  using *exponential search*—we double an estimate  $\Delta'$  of  $\Delta_i$  until  $N[j + \Delta'] \geq M[i]$ . We can then locate  $M[i]$  exactly using binary search in time  $\mathcal{O}(\log \Delta_i)$ . Overall, we get execution time  $\sum_{i=1}^m \mathcal{O}(\log \Delta_i) \leq \mathcal{O}(m \log \frac{n}{m})$ . Since we perform two searches for each element, the constant factors are even worse than for binary search. A somewhat faster variant of binary search therefore uses the

<sup>1</sup>The ' $\leq$ ' indicates that one can actually profit from the nonuniformity present in real world inputs. Indeed, Blandford demonstrates in [8] that by carefully choosing docIDs to make lists more clustered, further space can be saved.

*divide-and-conquer* principle [2, 3]. The middle element  $N[\lfloor n/2 \rfloor]$  of  $N$  is located in  $M$  using binary search and then intersection proceeds recursively on the respective left and right halves of  $M$  and  $N$ . All these algorithms have the disadvantage that binary search is incompatible with  $\Delta$ -encoding.

**Using a Two-Level Representation.** To overcome the limitation of binary search to non- $\Delta$ -encoded data, we have implemented a two-level representation. In addition to a  $\Delta$ -encoded compressed list, a *top-level* data structure  $t$  stores every  $B$ -th element of  $N$  together with its position in the main list ( $B$  is a tuning parameter). We can now run any binary search based algorithm on  $t$  and then scan only the pieces of the main list that might contain an element to be located.

**Skipper.** Using the above two-level representation one can also run a two-level variant of zipper that scans the top level data structure and delves down into the lower level whenever necessary. This avoids the complicated control structure of binary search and allows us to use  $\Delta$ -encoding also for docIDs and references stored in the top level data structure. A similar scheme where all the information is embedded into a single bit sequence has been proposed and implemented in [6].

### 3 Randomized Inverted Indices

**Pseudo random permutations.** For simplicity, we explain the algorithm for the case that  $U = 2^{2u}$  for some integer  $u$ . The general case is explained in [9, 10]. Consider the *Feistel permutation*  $\pi_i(u \circ v) := v \circ (u \oplus f_i(v))$  where  $\circ$  denotes the concatenation of  $u$ -bit bitstrings,  $\oplus$  denotes bit-wise exclusive-or, and where  $f$  is a random mapping  $0..2^u - 1 \rightarrow 0..2^u - 1$ . Note that  $f$  can be efficiently implemented by filling a lookup table of size  $\sqrt{U}$  with random values. Luby and Rackoff [9] have shown that chaining 4 Feistel permutations gives a permutation which is pseudorandom even in a cryptographic sense. Our implementation chains only two Feistel permutations.

**Representation.** Consider a list  $N = \langle d_1, \dots, d_n \rangle$ . The basic idea of our algorithm is to split the range of docIDs into *buckets* based on their most significant bits. Let  $B$  denote a parameter we are free to choose. Roughly,  $B$  controls the average bucket size. Let  $k_N = \lceil \lg \frac{UB}{n} \rceil$ . Bucket  $b_i^N$  stores the sequence of docID residues  $\langle d_j \bmod 2^{k_N} : d_j \gg k_N = i \rangle$ . Due to randomization, the average size of  $b_i^N$  will be between  $B/2$  and  $B$ . There will be smaller and larger buckets also, but basic balls-into-bins theory guarantees with high probability, that no bucket size will be very much larger than  $B$ . Each bucket can be stored as in the comparison based approach. Uncompressed or using  $\Delta$ -encoding with bit-compression or various variable

**Function** *lookup*( $M, N$ )

```

 $O := \langle \rangle$  // output
 $i := -1$  // current bucket in  $N$  (now a dummy)
foreach  $d \in M$  do // unpack  $M$ 
   $h := d \gg k_N$  // new bucket in  $N$ 
   $\ell := d \text{ bitAnd } 2^{k_N} - 1$  // least significant bits
  if  $h > i$  then // move to start of new bucket
     $i := h$  // new current bucket in  $N$ 
     $j := t^N[i]$  // current position in  $b^N$ 
     $e := t^N[i + 1]$  // start of next bucket in  $b^N$ 
    while  $j < e$  do //  $b_i^N$  not yet exhausted
       $\ell' := b^N[j]$  // unpack if necessary
      if  $\ell \leq \ell'$  then
        if  $\ell = \ell'$  then append  $d$  to  $O$ 
        break loop
       $j++$ 
return  $O$ 

```

Figure 1: High level pseudocode for our intersection algorithm.

bit-length coding schemes. The encoding of all buckets can be stored consecutively in one array  $b^N$ . A top level data structure  $t^N$  provides the information leading from the most significant bits  $i = d \gg k_N$  of a docID  $d$  to the bucket  $b_i^N$ . In the simplest case,  $t^N$  is an uncompressed array of indices into array  $b^N$ .

**Intersection.** Figure 1 gives pseudocode for our intersection algorithm. The smaller list  $M$  is traversed by the outermost loop. The current element  $d$  of  $M$  is disassembled into its most significant bits  $h$  and its least significant bits  $\ell$  such that  $h$  can be used as an index into the top level data structure of  $N$ .<sup>2</sup> Thus  $h$  determines the bucket of  $N$  where we have to look for elements matching  $\ell$ . We now have to scan bucket  $b_h^N$  for an element  $\ell'$  that coincides with  $\ell$ . If  $b_h^N$  is not the bucket we have been scanning before, we have to scan it from the start. Otherwise, we continue scanning from where we left off. There are three ways that scanning  $b_h^N$  can stop: If we reach the end of  $b_h^N$  or if we encounter an element  $\ell < \ell'$ , we know that  $d$  cannot be in  $N$  and go to the next iteration of the outer loop. Otherwise, we have found  $d$  in  $N$  and thus output it.

#### 3.1 Analysis

**THEOREM 3.1.** *Algorithm lookup runs in expected time  $\mathcal{O}(m + \min\{n, Bm\})$ .*

<sup>2</sup>In the special case  $k_N = k_M$  we can save a few bit operations by using the pieces stored in the data structure for  $M$  directly.

*Proof.* (Outline) The total time spend outside the inner loop is  $\mathcal{O}(m)$ . Since each bucket of list  $N$  is scanned at most once, there is also a bound of  $\mathcal{O}(m+n)$  on the time spend in the inner loop. Another bound for the time in the inner loop is  $\sum_{d \in M} |b_{d \gg k^N}^N|$  which makes the conservative assumption that for each element  $d \in M$  the entire bucket of  $N$  corresponding to  $d$  has to be scanned. Using linearity of expectation it thus suffices to bound the expected size of a bucket  $b_i^N$  of list  $N$ .

Note that we get time  $\mathcal{O}(n)$  if  $B = \mathcal{O}(1)$ .

**THEOREM 3.2.** *Using bit-compressed  $\Delta$ -encoding and assuming a truly random permutation, our representation needs  $n(\lg \frac{U}{n} + \lg \min \{B, \log n\} + \mathcal{O}(1 + \frac{\log n}{B}))$  bits for storing a list of size  $N$  with high probability.*

*Proof.* (Outline) There are  $\Theta(n/B)$  buckets. Assume each bucket begins at a machine word of size  $\Theta(\log n)$  bits. This means  $\mathcal{O}(\log n)$  unused bits at the end of a bucket and  $\mathcal{O}(\log n)$  bits for storing pointers to bucket starts. The largest difference between two bucket entries can be bounded in two ways—the bucket width  $\Theta(UB/n)$  and the largest difference between any two consecutive list entries. The latter quantity is well known from probability theory: We throw  $n$  balls into  $U$  (different) pots and ask for the longest consecutive stretch of unoccupied pots. Using standard techniques, it can now be shown that the largest difference is  $\mathcal{O}(\frac{U}{n} \lg n)$  with high probability. The two bounds can be combined to a bound of  $\lg \min \{\Theta(UB/n), \mathcal{O}(\frac{U}{n} \lg n)\} = \lg \frac{U}{n} + \lg \min \{B, \log n\} + \mathcal{O}(1)$  bits per list element.

From this bound and its proof we can learn that using bit-compression,  $\Delta$ -encoding is actually only useful when  $B = \Omega(\log n)$ . For smaller  $B$ , it is quite likely that there are some empty bins and thus also bins with only one element which will determine a maximum difference close the the width of an entire bucket. We can harness the full power of  $\Delta$ -encoding using variable bit length encoding which will replace the term  $n \lg \min \{B, \log n\}$  by  $\mathcal{O}(n)$ . The term  $n \lg \frac{U}{n}$  is unavoidable since it already shows up in the information theoretic lower bound  $\lg \binom{U}{n} \approx \lg \frac{U^n}{n!} = n \lg \frac{U}{n} + \Theta(n)$ . The term  $\mathcal{O}(n \frac{\log n}{B})$  can be made arbitrarily small by choosing a sufficiently large bucket size. However, this comes at the cost of increased running time when intersecting lists of asymmetric size. Still, for all but the longest lists the term  $n \lg \frac{U}{n}$  will dominate space consumption.

For very long lists which need only a few bits for encoding each entry of a bucket, it might pay off to avoid aligning the beginning of a bucket with a word boundary

and to replace our two-level data structure with a three-level data structure. We could use *meta buckets* of size  $\mathcal{O}(\log n)$  and bottom level buckets of size  $\mathcal{O}(1)$  whose position within a meta bucket can be encoded using  $\mathcal{O}(\log \log n)$  bits with high probability. In practice, this might boil down to using a single byte for encoding the relative position of a bottom level bucket.

### 3.2 Further Issues

**Incremental Document Insertion.** So far, we have assumed a static set of  $U$  documents. This situation can be dynamized using a number of rather standard techniques. Newly inserted documents can first be inserted into a small buffer data base that need not be compressed and supports dynamic insertion. Periodically, this data base can be merged (in the background) with the main data base. New documents arriving during this reconstruction go to a tertiary data base that becomes the secondary data base once merging is completed. Furthermore, it may make sense to replace the permutation  $\pi : \mathcal{U} \rightarrow \mathcal{U}$  by a permutation on the larger range  $0..4^{\lceil \log_4 U \rceil} - 1$ . This permutation is easier to evaluate and yields an pseudorandom injective mapping from the range  $0..U - 1$  to  $0..4^{\lceil \log_4 U \rceil} - 1$ . Unless  $U$  is already a power of four, we can now introduce a new docID  $\pi(U)$  by simply incrementing  $U$ .

**Load Balancing.** Any large full text data base will nowadays use multiple processors. Currently, a typical configuration is a cluster of low cost servers each equipped with one or two multi-core processors. The easiest way to exploit  $p$  processing elements (PEs) is to assign about  $U/p$  documents to each PE. Now queries simply go to all the PEs. Doing this naively, based on nonrandomized docIDs, will lead to poor load balancing however because document sizes, and term appearances will be correlated with the docID. Randomization largely dissolves all these problems. We can just map docID  $i$  to PE  $i \bmod p$  (round robin). We get strong probabilistic guarantees not only that every PE is responsible for a similar overall amount of data but also that *every* list of the inverted index is split almost evenly. Note that trying to do this deterministically would be very complicated. Using balls-into-bins arguments one can see that a given list of length  $n$  gets split into sublists of size  $\frac{n}{p} + \mathcal{O}(\sqrt{\frac{n}{p} \lg p})$  with high probability. Note that in particular long lists, which cause the largest query times, will get split very evenly.

Table 1: Properties of the used text corpora.

	WT2g	WT2g.s	BibTex
#documents	239 492	10 749 669	1 195 624
volume [KB]	1 483 926	1 493 788	70 783
max $ L $	212 974	5 742 167	387 794
#terms	1 532 125	1 703 342	164 345

## 4 Experiments

We have implemented all algorithms using (fairly low level) C++ and the STL, in particular, making heavy use of `vectors`. We use macros to get an implementation that is at the same time efficient and allows us to flexibly combine different basic representations, compression schemes, and intersection algorithms. Both the comparison based and the randomized representation use a two level representation. The bottom level may be uncompressed, compressed by bit-compression only, or use  $\Delta$ -encoding combined with bit-compression, or escaping.<sup>3</sup> Our escaping implementation decides for each list separately, what number of bits per piece gives the best compression. The top level can be uncompressed or bit-compressed.  $\Delta$ -encoding and escaping are not implemented because only skipper would be able to work with this representation.

The experiments were done on one core of an Intel Xeon processor clocked at 3.2 GHz with 4 GB main memory and  $2 \times 512$  KByte L2 cache, running SuSE Linux Enterprise Server 9 (kernel 2.6.5). The program was compiled by the gnu C++ compiler 3.3.3 using optimisation level `-O3`. Timing was done using `clock_gettime`. Since time measurements on our system are noisy, and since we measure instances with very different sizes, we smoothed Bezier curves rather than individual data points.

Table 1 summarizes the properties of the real world instances we have used. WT2g<sup>4</sup> is a standard benchmark for web search. Although WT2g is by now considered a ‘small’ input, it seems to be the right size for the amount of data assigned to *one processing core* of a main memory search engine based on a cluster of low cost machines. WT2g is our default benchmark underlying the subsequent experiments if not otherwise mentioned. WT2g.s is derived from the documents of WT2g by viewing every sentence of the text as a document of its own. This leads to longer lists and might better reflect the application at SAP. BibTeX contains the titles of the bibliography data base

<sup>3</sup>At one point we also tried Golomb coding which turned out to be inferior to escaping both with respect to size and even more to execution time.

<sup>4</sup>[http://ir.dcs.gla.ac.uk/test\\_collections/access\\_to\\_data.html](http://ir.dcs.gla.ac.uk/test_collections/access_to_data.html)

<http://liinwww.ira.uka.de/bibliography/>. Let  $\mathcal{L}$  denote the sorted sequence of all lists in decreasing order of their lengths. We generate ‘difficult’ queries involving long lists and widely varying length ratios as follows: We investigate the range of length ratios  $[0.001, 1]$  by subdividing it into 100 subintervals that all have the same width on a logarithmic scale. We determine up to  $s = 10$  pairwise queries falling into each interval set  $I(\cdot)$  using the following algorithm:

```

foreach  $N \in \mathcal{L}$  do
  foreach  $M \in \mathcal{L}$  with  $|M| \leq |N|$  do
    if  $|I(|M|/|N|)| < s$  then
       $I(|M|/|N|) := I(|M|/|N|) \cup \{(M, N)\}$ 

```

Figure 2 compares the space consumption of our two-level data structures for different encoding schemes. Interestingly, escaping is not so useful for the randomized representation—only for very large bucket sizes we see a tiny improvement over bit-compressed  $\Delta$ -encoding. As predicted by our analysis of the randomized representation, bit-compression does not need  $\Delta$ -encoding for small bucket sizes. Since the overhead of  $\Delta$ -encoding is negligible however, from now on we use bit-compressed  $\Delta$ -encoding as our default implementation of the randomized representation. As to be expected, the deterministic variant does not work well with bit-compressed  $\Delta$ -encoding. For large bucket sizes, escaping can exploit the nonuniformity of the input to get smaller space consumption than the randomized implementation. For small bucket sizes, the deterministic implementation needs more space in order to store the the last entry in each bucket.<sup>5</sup> From now on  $\Delta$ -encoding with escaping is our default deterministic implementation. Figures 10 and 12 in the appendix show that the results for other problem instances are similar. The main difference is that these instances need somewhat more space per list entry. BibTeX because the lists are shorter and WT2g.s because the universe is bigger.

Running time of lookup for different bucket sizes is shown in Figure 3. As to be expected, large buckets are bad for small length ratios and good for nearly equal lengths. However, we quickly reach a point of diminishing return so that bucket size eight looks like a good compromise for all length ratios.

Algorithm skipper behaves differently than lookup for varying bucket sizes as can be seen in Figure 8 in the appendix. In particular, for medium length ratios, neither very large nor very big buckets give the best choice. This is because we have a delicate compromise between two evils—scanning big buckets or scanning

<sup>5</sup>This disadvantage would be smaller in a representation optimized for algorithm skipper because it could actually work with a top level using  $\Delta$ -encoding and escaping.

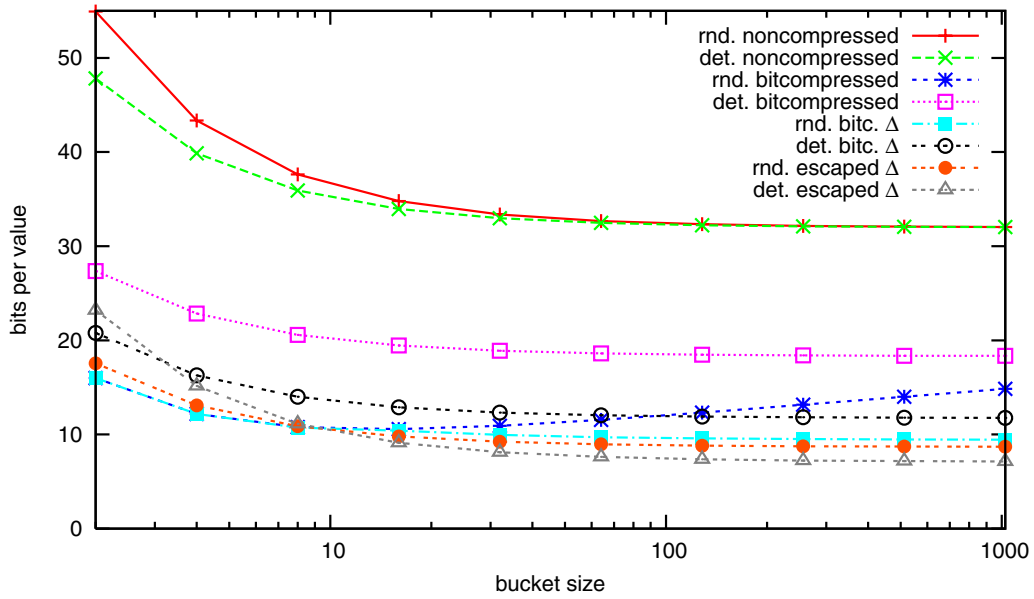


Figure 2: Space consumption of different encoding schemes.

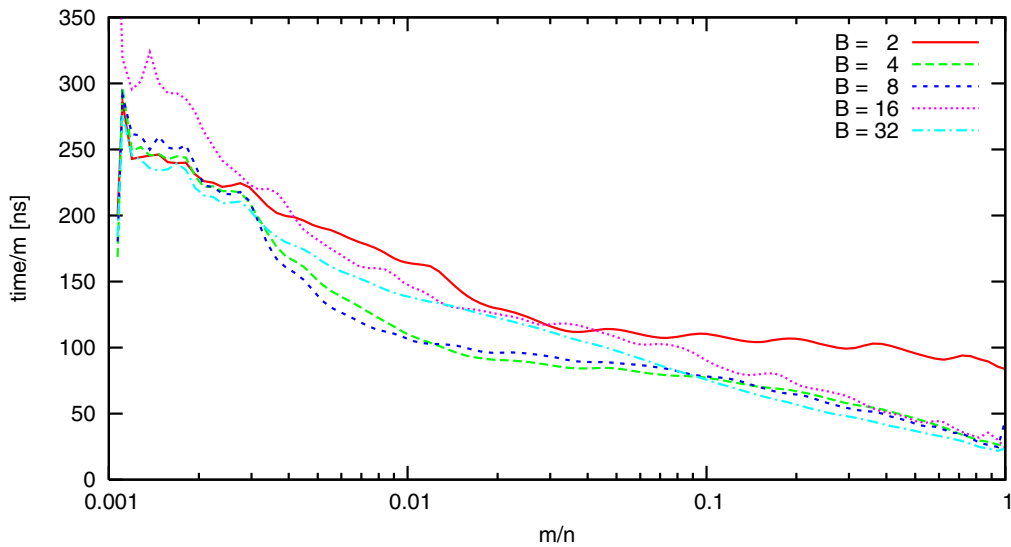


Figure 3: Performance of algorithm lookup (using bit-compressed  $\Delta$ -encoding) for different bucket sizes.

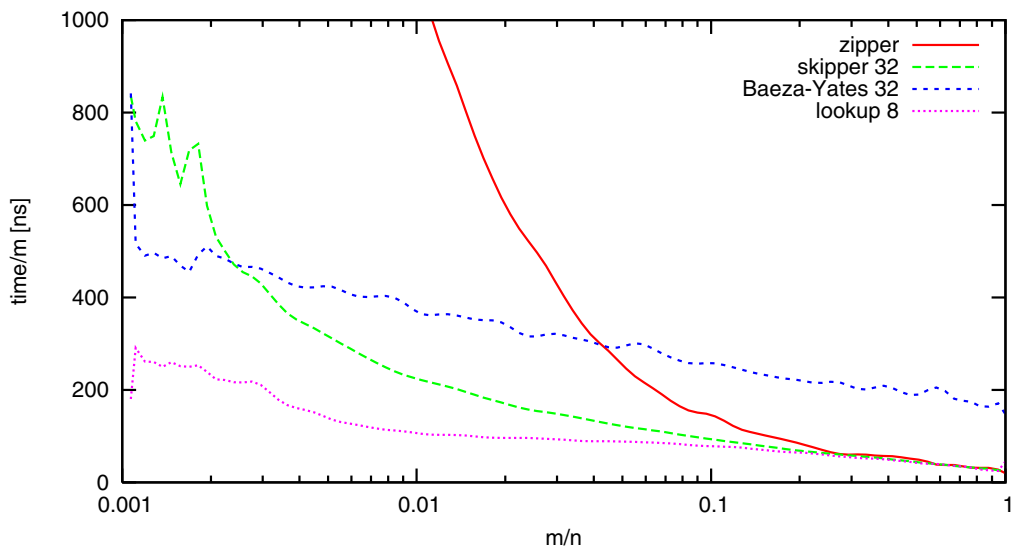


Figure 4: Performance of different intersection algorithms.

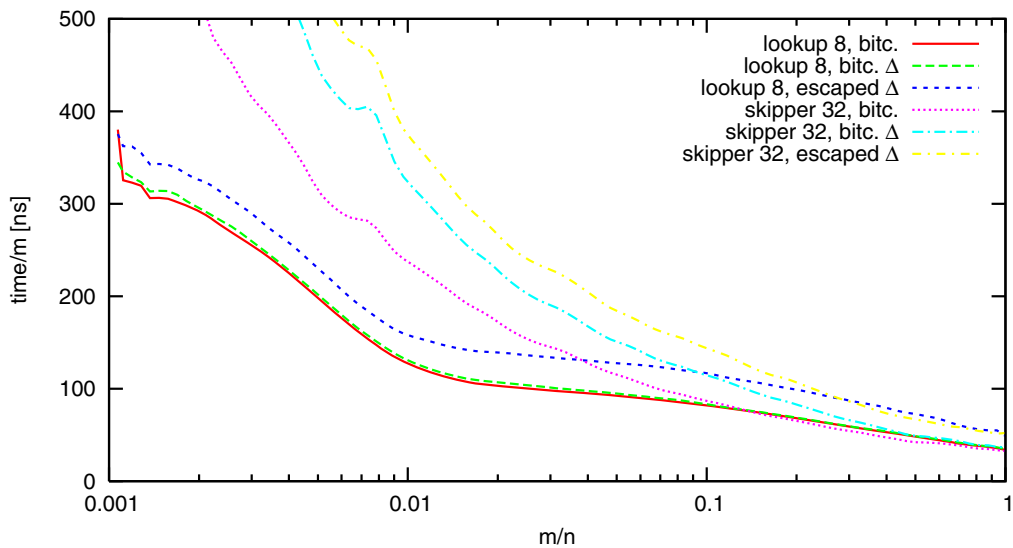


Figure 5: Space-time tradeoff using different encoding schemes.

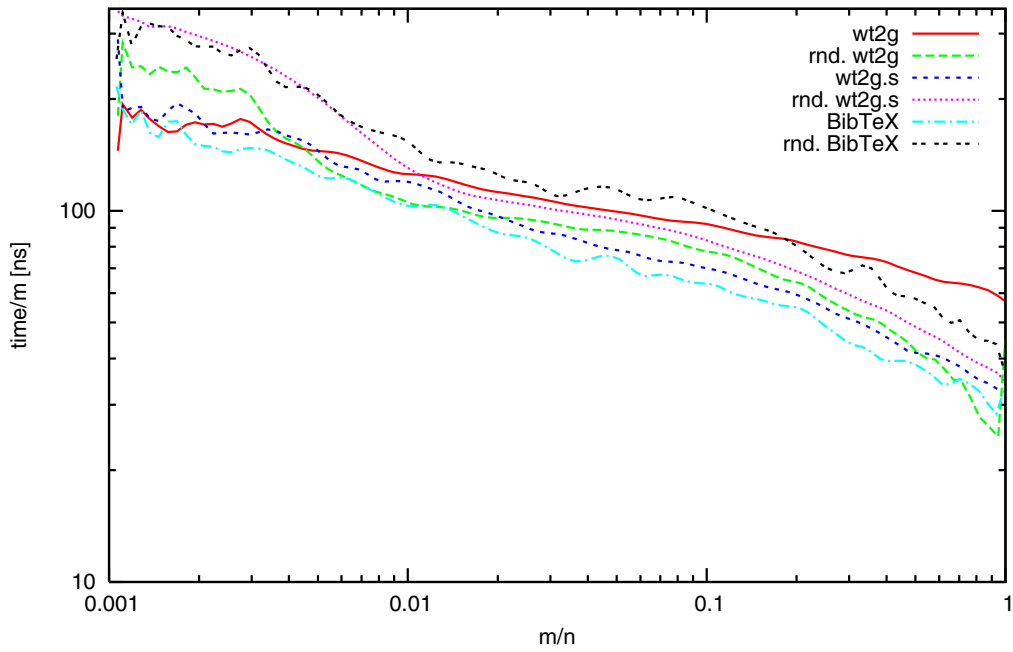


Figure 6: The impact of randomization on algorithm lookup.

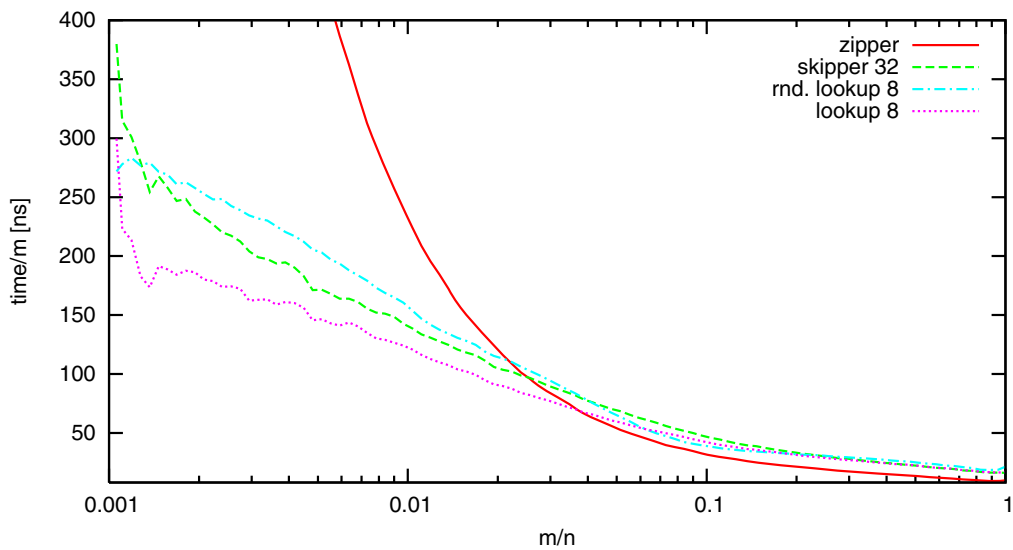


Figure 7: Performance of different algorithms running on non-compressed data.

many entries on the top level. We use  $B = 32$  as a good compromise.

The same bucket size is also the best choice for algorithm Baeza-Yates. Figure 9 in the appendix shows that here performance is less dependent on bucket size, since larger buckets increase bucket scanning overhead but also require more of the expensive levels of recursion.

We are now in the position to compare all four main algorithms in Figure 4. The most important observation is that lookup is the best algorithm up to a length ratio close to one. For lists of similar length, zipper, skipper, and lookup are all very good. Still, it might be a good idea to implement a version of lookup optimized for lists of similar length. Skipper outperforms Baeza-Yates for all but very small length ratios. Figures 11 and 13 in the appendix show that the same basic tendencies also hold for our other problem instances.

Obviously, the used compression scheme has influence on the running time of the algorithms. Figure 5 shows this space-time tradeoff for lookup and skipper running on instance WT2g.s. As mentioned above, the performance loss caused by the  $\Delta$ -encoding in the lookup data structure is negligible low. By contrast, the escaped encoding requires perceivable more time for unpacking the docIDs. But in this case it is not rewarded by any reduction of space (cp. figure 2). For skipper all the encodings have clearly different running times. As expected, using bit-compression requires the longest time followed by bit-compressed  $\Delta$ -encoding and  $\Delta$ -encoding with escaping. However, the performance gain of the less sophisticated encoding schemes is bought dearly by a clearly worse compression (cp. figure 2).

In Figure 6, we investigate the impact of randomization. The results are somewhat astonishing. For instance WT2g, randomization helps for small list ratios and harms for large ones. However, the performance difference is not damning. For instances BibTeX and WT2g.s, omitting randomization always helps. Thus, algorithm lookup is also a good heuristic for nonrandomized data.

The *negative* practical performance impact of randomization is even stronger for non-compressed data. Figure 7 shows that using this representation skipper *outperforms* randomized lookup, versus the latter algorithm still wins if randomization is not used.

## 5 Discussion

The outcome of our study is a little bit like election day in Germany. All the parties usually claim that they have won in some respect.

Zipper is the uncontested best algorithm for nearly equal sequence lengths. But we should keep in mind

that this difference is not so big for compressed inputs and that the length ratio where we have break even with skipper or lookup is more like 1 : 5 than 1 : 20. Hence, one conclusion is that it is worth looking for an alternative algorithms for small  $m/n$ .

Our new algorithm lookup is at the same time simple and among the best algorithm over the entire spectrum of length ratios. For compressed indices and small  $m/n$  it can claim considerable speedups over all other algorithms in our experiments. Randomization allows interesting performance guarantees on both execution time and space requirement. However it seems that it is not so good in practice to destroy the dependencies present in real world inputs. Even lookup itself sometimes profits from these dependencies. Even randomized load balancing can be achieved by other means than a complete random permutation of the data.

Algorithm skipper is as simple as lookup and can exploit dependencies in data more directly than lookup. Its party can argue that the lower performance for small  $m/n$  is partly due to the different space time tradeoff. In particular, what we have not tried yet is a skipper with non-compressed top level data structure. With this, skipper would still remain space competitive with lookup and for small  $m/n$  it would greatly profit from faster scanning of the top level data structure.

The only clear losers in our study seem to be asymptotically efficient algorithms like Baeza-Yates. For all  $m/n$  there are other algorithms that are considerably faster. However, Baeza-Yates and its successors are comparatively complicated and get even more complicated when combined with compression. Rather than seeing this as another disadvantage, one could argue that a clever implementation could considerably improve on our results. Hence, proponents of such algorithms could view our study as a stimulating challenge to make studies with large real world inputs and compression.

**Acknowledgements** We would like to thank Holger Bast, Franz Färber and Günter Radestock for valuable discussions and Thomas Worsch for making the BibTeX data available.

## References

- [1] Hwang, F.K., Lin, S.: Optimal merging of 2 elements with  $n$  elements. *Acta Informatica* **1** (1971) 145–158
- [2] Baeza-Yates, R.: A fast set intersection algorithm for sorted sequences. In: 15th Symposium on Combinatorial Pattern Matching. Volume 3109 of LNCS. (2004) 400–408
- [3] Baeza-Yates, R.A., Salinger, A.: Experimental analysis of a fast intersection algorithm for sorted se-

- quences. In: String Processing and Information Retrieval, 12th International Conference. Volume 3772 of LNCS., Springer (2005) 13–24
- [4] Barbay, J., López-Ortiz, A., Lu, T.: Faster adaptive set intersections for text searching. In Álvarez, C., Serna, M.J., eds.: Experimental Algorithms, 5th International Workshop, WEA. Volume 4007 of LNCS., Springer (2006) 146–157
- [5] Demaine, E.D., López-Ortiz, A., Munro, J.I.: Experiments on adaptive set intersections for text retrieval systems. In: Algorithm Engineering and Experimentation, 3rd International Workshop, ALENEX. Volume 2153 of LNCS., Springer (2001) 91–104
- [6] Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems* **14** (1996) 349–379
- [7] Dementiev, R., Sanders, P.: Asynchronous parallel disk sorting. In: 15th ACM Symposium on Parallelism in Algorithms and Architectures, San Diego (2003) 138–148
- [8] Blandford, D.: Compact Data Structures for Fast Queries. PhD thesis, CMU (2005)
- [9] Luby, M., Rackoff, C.: How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing* **17** (1988) 373–386
- [10] Dementiev, R., Sanders, P., Schultes, D., Sibeyn, J.: Engineering an external memory minimum spanning tree algorithm. In: IFIP TCS, Toulouse (2004)

A More Figures  
A.1 Instance WT2g

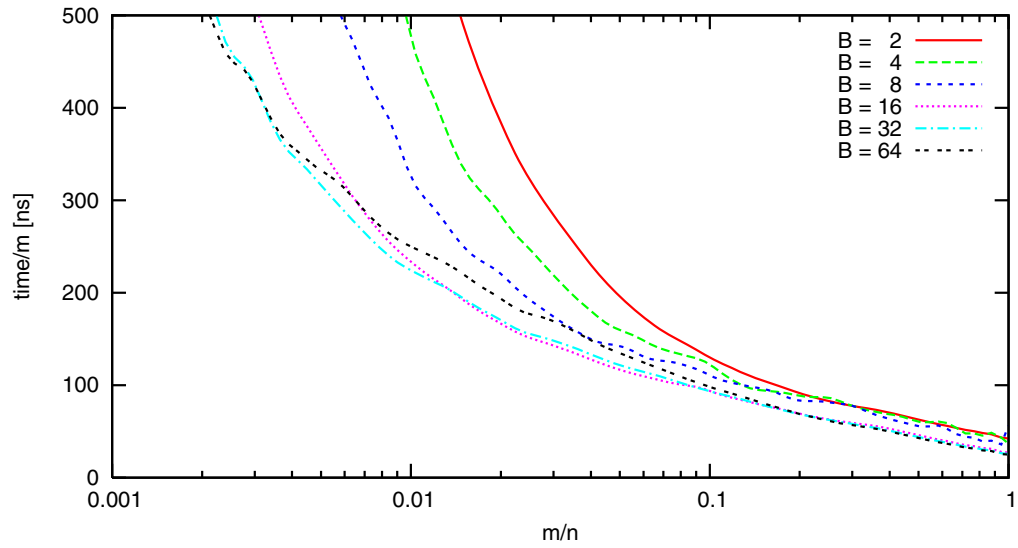


Figure 8: Performance of algorithm skipper (using  $\Delta$ -encoding with escaping) for different bucket sizes.

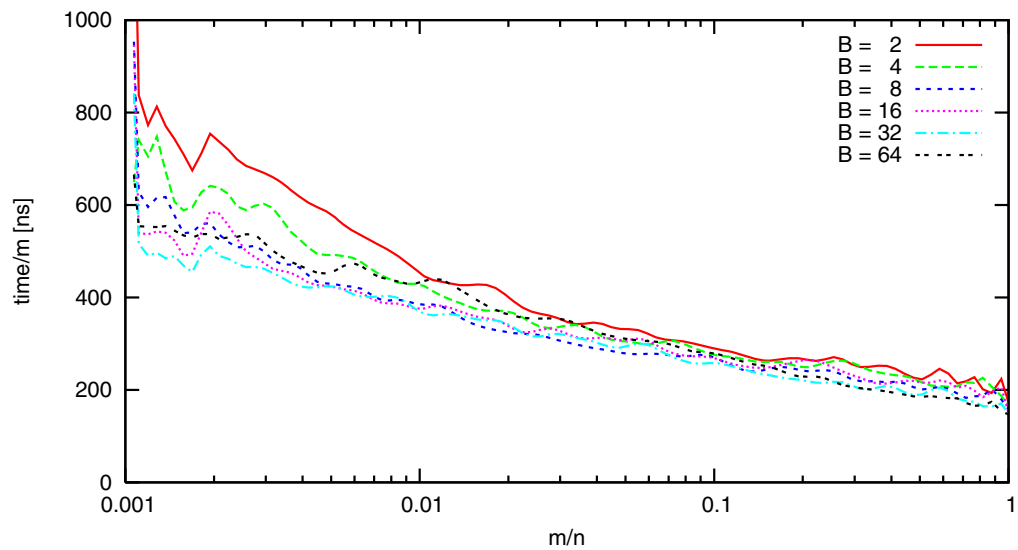


Figure 9: Performance of algorithm Baeza-Yates (using  $\Delta$ -encoding with escaping) for different bucket sizes.

## A.2 Instance WT2g.s

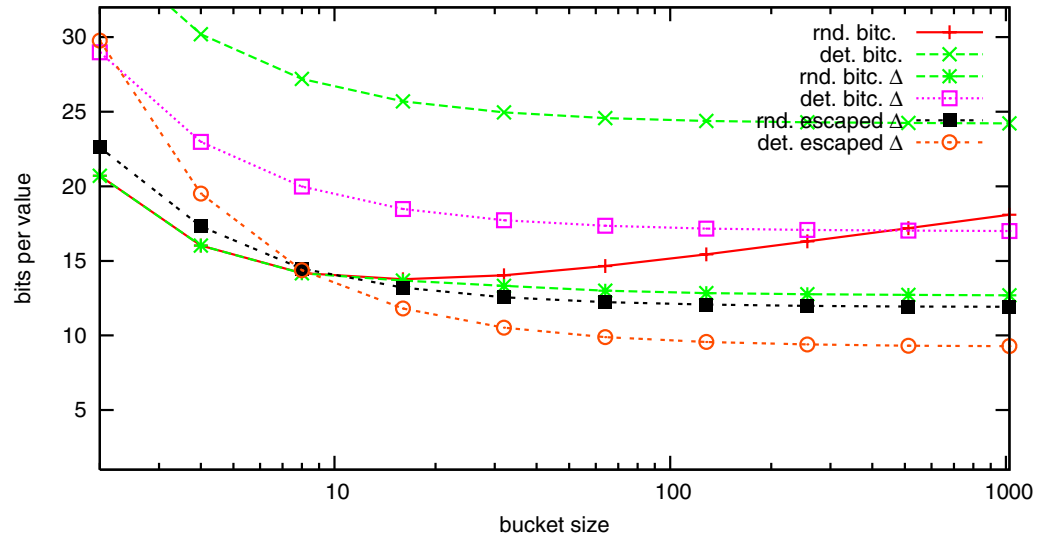


Figure 10: Space consumption of different encoding schemes.

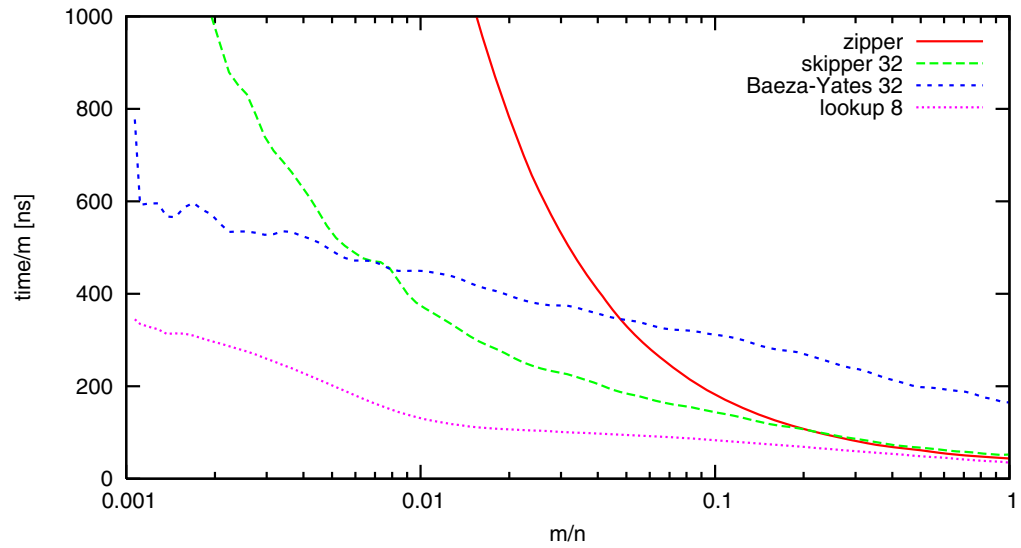


Figure 11: Performance of different intersection algorithms.

### A.3 Instance BibTeX

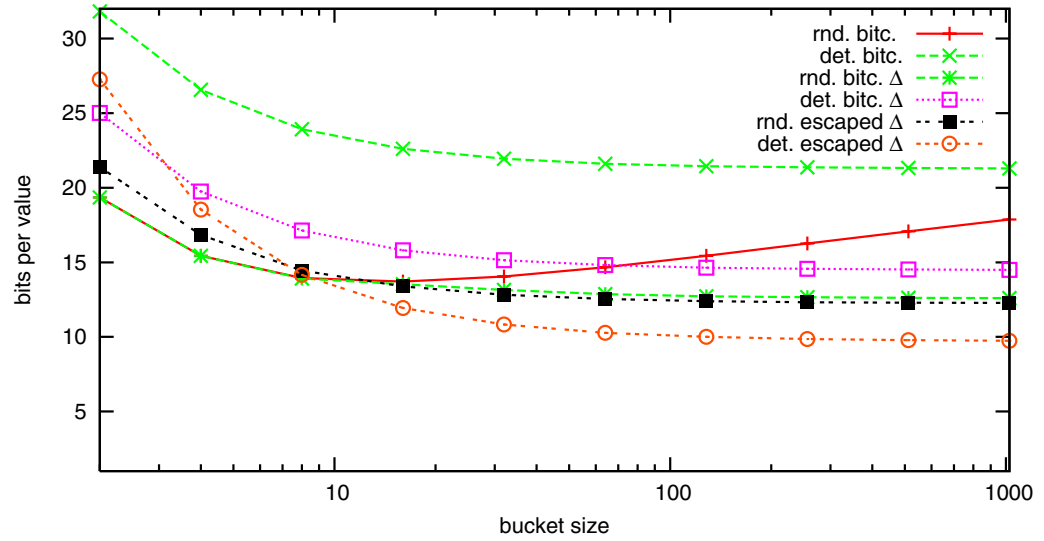


Figure 12: Space consumption of different encoding schemes.

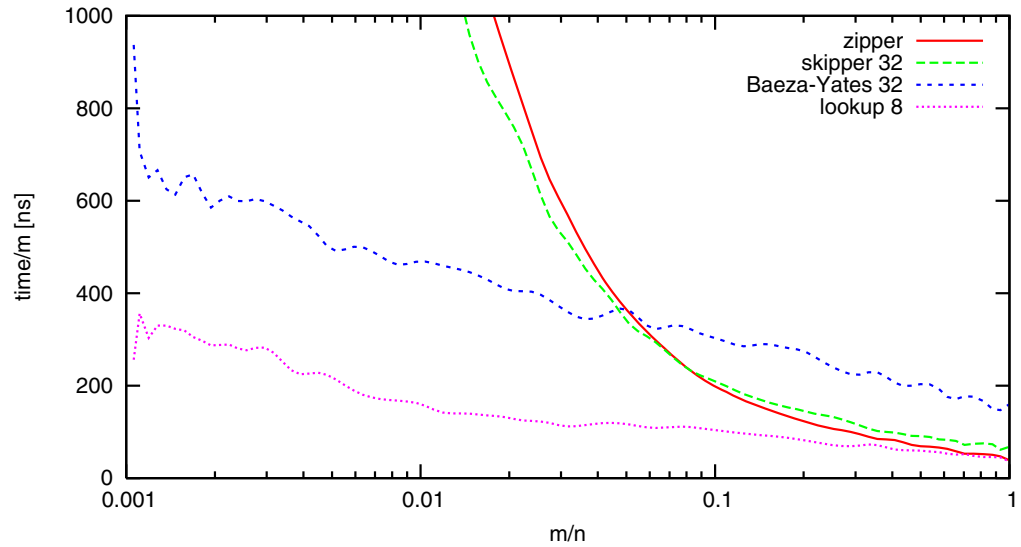


Figure 13: Performance of different intersection algorithms.