R3D3: A Doubly Opportunistic Data Structure for Compressing and Indexing Massive Data

Máté Nagy, János Tapolcai, Gábor Rétvári

MTA-BME Lendület Future Internet Research Group, MTA-BME Information Systems Research Group
Department of Telecommunications and Media Informatics, BME, Email: {nagym, tapolcai, retvari}@tmit.bme.hu

***Abstract*—Opportunistic data structures are used extensively in big data practice to break down the massive storage space requirements of processing large volumes of information. A data structure is called (singly) opportunistic if it takes advantage of the redundancy in the input in order to store it in information-theoretically minimum space. Yet, efficient data processing requires a separate index alongside the data, whose size often substantially exceeds that of the compressed information. In this paper, we introduce doubly opportunistic data structures to not only attain best possible compression on the input data but also on the index. We present R3D3 that encodes a bitvector of length n and Shannon entropy H_0 to nH_0 bits and the accompanying index to $nH_0(1/2 + O(\log C/C))$ bits, thus attaining provably minimum space (up to small error terms) on both the data and the index, and supports a rich set of queries to arbitrary position in the compressed bitvector in $O(C)$ time when $C = o(\log n)$. Our R3D3 prototype attains several times space reduction beyond known compression techniques on a wide range of synthetic and real data sets, while it supports operations on the compressed data at comparable speed.**

***Index Terms*—succinct and compressed data structures, compressed self-indexes, big data, packet forwarding**

I. Introduction

Recently, the exponential growth of available electronic information has created new challenges in data mining, machine learning, pattern analysis, and networking, as the sheer volume of data to be stored, transferred, and processed online has greatly surpassed the increase in memory, disk, and link capacities of current computers and computer networks [1], [2]. Space reduction for massive data processing applications is an attractive choice to tackle these challenges, as storage space is fundamentally related to the time it takes to process data [3]. In fact, by making better use of cache and memory levels closer to the processor, waiving the painful cost of disk accesses, and utilizing processor–memory bandwidth more efficiently, space reduction techniques can make processing of unprecedentedly large quantities of data feasible even in resource-constrained environments. Ultimately, the goal is to *store data in memory in a compact or compressed format* and *still operate directly on it* without any major performance hit compared to a naive, uncompressed representation [4].

Succinct and compressed data structures are a relatively new development in theoretical computer science that promise with substantial decrease in the memory footprint of big data operations, by storing sequential or structured static data in a compressed but readily accessible, queryable, and manipulable format [5]. Applications encompass essentially the entire field of computer science, from space-efficient encodings of ordered sets, sparse bitmaps, partial sums, binary relations, range queries, and arbitrary sets supporting predecessor and successor search [6]–[9], ordinal and labeled trees [6], [9]–[13] and general graphs [6], [14], indexing massive textual data [5], [12], [15]–[19], top-k document retrieval, suffix trees, arrays, and inverted indexes in information retrieval systems [12], [16], [19]–[22], point grid queries in computational geometry [22] and genome compression in computational biology [23], [24], all the way to key-value stores, log analytics, machine learning, data mining, and big data applications [4], [25].

The cornerstone of these schemes is a *compressed bitvector representation* that encodes an arbitrary bitmap in very small space and, at the same time, implements some simple operations, namely access, rank, and select queries (see later), right on this compactified format [6], [15], [17], [26]–[30]. Such compressed bitvectors can then be used to build composite data structures and construct complex queries on them [6]. As recently shown, for instance, such compressed bitvectors can be used to construct a space-efficient representation for Internet routers' forwarding tables (FIBs) [31]. The resultant compressed IP FIBs have been shown to squeeze the routing table of a contemporary IPv4 router, counting beyond 500,000 prefixes, to a mere 70–200 kbytes of memory, while supporting wire-speed longest-prefix matching right on the compressed form.

Space usage of any queryable data structure boils down to two elementary components: the space for storing the data itself, plus some additional space for an index into the data that guarantees fast access [25]. In this setting, the data component constitutes the useful information and the index is pure redundancy, whose size should be minimized as much as possible. The first technique to attain worst-case-optimal storage space on both the data and the index components was the *succinct bitvector and ordered-tree data structures* due to Jacobson [6] (but see also [15]). The memory footprint was further reduced by Ferragina and Manzini, who introduced *opportunistic data structures that attain information-theoretically minimal entropy-constrained storage space on the data component* [20]. Their data structures are called (singly) opportunistic in that they can take advantage of the compressibility of the input by decreasing the space occupancy beyond the worst-case limit, at no significant slowdown in

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0

Figure 1: A sample bitvector.

query performance. A good example for such an opportunistic compression approach is the RRR compressed bitvector scheme due to Raman, Raman, and Rao [12], attaining nH_0 bits on the data and $O(\frac{n \log\log n}{\log n}) = o(n)$ bits on the index, where n is the length of the input and H_0 is the zero-order empirical entropy [32], while supporting access, rank, and select queries in optimal $O(1)$ time. Today, the RRR scheme serves as the major building block for space-efficient data processing techniques, enjoying wide-scale use throughout the entire spectrum of compressed information processing [12], [16], [18], [19], [27], [29].

A major shortcoming of compressed information processing is, however, that the storage size of the index can significantly outweigh (up to and beyond 8 times, [4]) that of the data, taking a huge toll on the storage efficiency of data compression and hindering engineering applications [27]–[30]. To address this limitation, in this paper we introduce the concept of *doubly opportunistic data structures*, which, as opposed to conventional opportunistic schemes that compress only the data component, *achieve information-theoretically minimal entropy-constrained space both on the data and the index at the same time*. We present R3D3 ("RRR–Developed Data structure for big Data"), which combines the storage scheme of RRR for encoding the index and the Elias-Fano compression method [15] for block-encoding the data, to attain $nH_0 + nH_0(\frac{1}{2} + O(\frac{\log C}{C}))$ bits of space and random access and rank queries in $O(C)$ time and select in $O(\log n)$ when $C = o(\log n)$ constant. R3D3 thusly not only attains provably maximum compression (up to small error terms) on both the data and the index, and hence qualifies as the first doubly-opportunistic bitvector compression scheme, but it also allows to realize many interesting engineering trade-offs between storage space and query time by fine-tuning the constant C. By comprehensive evaluations on synthetic data sets and a real data corpus we show that R3D3 achieves from 2 up to 10 times smaller space than RRR while supporting queries in similar, or slightly worse, performance.

The rest of the paper is structured as follows. In Section II we review bitvector compression, in Section III we introduce R3D3 and give a detailed space–time analysis, in Section IV we present the results of our benchmarks, and finally we conclude our work in Section V.

II. Compressed Bitvector Indexing

In this section we give an overview on succinct and compressed data structures and we describe the RRR and the Elias-Fano coding schemes in some detail.

A. Notations and Definitions

Let t be a bitvector with length n. The number of bits set to 1 in t is called the population (or popcount) and the ratio

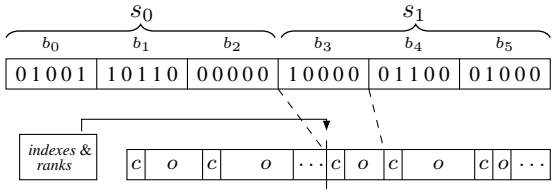

Figure 2: Sketch of the RRR encoding scheme.

of the population and n is the empirical probability p of 1s in t. Our aim is to build a compact representation for t that supports the following queries efficiently:

- $\mathrm{access}(t, i)$: return the i-th bit of t;
- $\mathrm{rank}_q(t, i)$: return the number of occurrences of symbol q in $t[1, i]$;
- $\mathrm{select}_q(t, i)$: return the position of the i-th occurrence of symbol q in t.

Consider the example in Fig. 1. Here, $n = 16$, $p = 3/16$ and $\mathrm{popcount}(t) = 3$, the query $access(t, 5) = 1$ tells that the bit at position 5 is set, $rank_1(t, 8) = 2$ gives the number of bits set to 1 up to and counting the 8-th position, and finally $select_1(t, 2) = 7$ indicates that the second set bit occurs at position 7. Notice that rank and select are "dual" in that if $\mathrm{select}_1(t, i) = m$ then $\mathrm{rank}_1(t, m) = i$. Further, $\mathrm{rank}_0(t, i) + \mathrm{rank}_1(t, i) = i$ but the same does not hold for select.

A *succinct encoding* of t will store t on worst-case minimum $n+o(n)$ bits of space (the uncompressed representation would need n bits and the error term $o(n)$ vanishes asymptotically) and implement access, rank, and select "fast" (preferably in $O(1)$). The naive "bitmap" representation is not succinct in this sense since it fails the second requirement; rank and select would need a linear sweep through the bitmap, taking $O(n)$ time. A *compressed encoding* of t, on the other hand, reduces the memory footprint beyond the worst-case limit, if the input is compressible, to $nH_0+o(n)$ bits, where H_0 is the zero-order empirical entropy (or the Shannon entropy) of t:

$$H_0 = p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p} \leq 1 \ ,$$

without any performance penalty on the performance of queries. Note that all our logarithms are base 2. For brevity's sake, we shall mostly omit rounding our logarithms to integers in the forthcoming analyses wherever this does not affect the validity of the results.

B. A Scheme due to Raman, Raman, and Rao

Raman, Raman, and Rao introduced the first compressed data structure for bitmaps, usually referred to as *RRR*, that solves access and rank queries in constant time [12]. In this paper, we describe a modified encoding due to Navarro and Providel [30], which, although needs slightly worse $O(\log n)$ time for queries, proved much more space- and time-efficient in practical implementations [33].

RRR comprises a block-coding component to encode the useful data and an indexing scheme to support queries to the blocks [27]–[30]. The structure partitions t into blocks

$b_1, b_2, \ldots$ of size $b = \frac{\log n}{2}$ bits (see Fig. 2 for an illustration). Each block b_i is encoded with a pair (c_i, o_i), where $c_i = \text{popcount}(b_i)$ is the *class* of b_i and o_i is the offset, or the *combinatorial rank*, of b_i, defined as the sequence number of b_i in some fixed enumeration (e.g., lexicographic order) of all combinations of exactly c_i occurrences of 1s on b bit positions [34]. Storing c_i needs $\log(c_i + 1)$ bits and o_i needs $\log \binom{b}{c_i}$, so the block codes (the *data component*) take $\sum_i \log(c_i + 1) + \log \binom{b}{c_i} = nH_0 + O(\frac{n}{\log n})$ bits overall [35].

The *indexing scheme* in turn groups every consecutive $\log n$ blocks into a superblock. Then, for each superblock the index stores the starting positions for the block-codes inside it and the cumulative rank up to the superblock's beginning, plus, for each block, the corresponding block-code's starting position and the rank at the block's beginning, both relative to the superblock that contains it. Cumulatively, this indexing structure needs $O(\frac{n \log \log n}{\log n}) = o(n)$ bits of space.

Answering $\text{access}(t, i)$ works as follows. As superblocks and blocks span constant number of bits in t, i uniquely determines the superblock and the block that contain position i. We follow first the superblock pointer and then the block pointer to reach the block-code for the corresponding position, this can be done in $O(1)$ time. From this point, decoding a block (the so called *combinatorial unranking* operation) takes $O(b) = O(\log n)$ time [30], [34], [36]. Solving rank goes similarly, but this time we also add up the superblock's and block's rank counters along the way, which, together with the time to unrank the block, takes $O(\log n)$ time. Finally, select binary-searches over superblock and block ranks, again in $O(\log n)$ time.

Experimental studies show that the $O(\frac{n \log \log n}{\log n})$ bits size of the index, although asymptotically small, may outweigh the data components' size nH_0 substantially, especially for low-entropy input [27]–[30]. Correspondingly, many schemes eliminate block-code pointers and rank counters from inside the superblocks, which tends to save a lot of space at the cost of degrading block access and rank to a linear search over the blocks of the superblock, making queries slow. This scheme is usually referred to as, somewhat confusingly, the *unindexed* version of RRR, to distinguish it from the above described version (with explicit block pointers and ranks inside superblocks) that is called *indexed* RRR.

Today, RRR is a popular tool amongst theoreticians and practitioners and constitutes a fundamental building block for compressed indexes of complex structured and unstructured types of information, like trees [13], strings (wavelet trees, [16]), or IP forwarding tables [31]. Practice has shown, nevertheless, that RRR exhibits a brittle space–time trade-off: meaningful storage space reduction can only be realized at the price of sacrificing precious query performance, like adopting larger block sizes [30] or swapping indexed-RRR to the much slower unindexed version [33].

C. The Elias-Fano scheme

Elias-Fano coding has been proposed in [15] to store a bitvector t in $nH_0 + o(n)$ space and answer select_1 queries

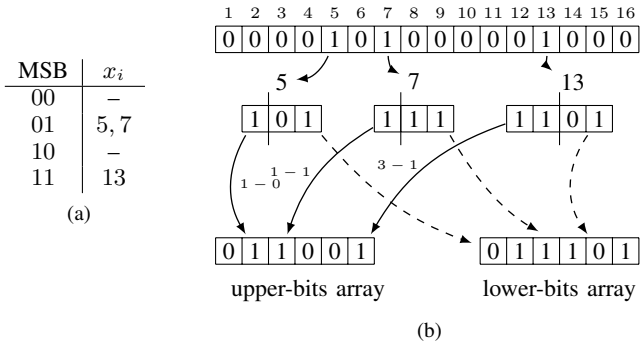

Figure 3: Elias-Fano encoding scheme: (a) MSB bucketing on the characteristic vector $(5, 7, 13)$, and (b) EF encoding.

$O(1)$ time, with no support for rank and access. Herein, we describe an alternative scheme *EF* that attains $nH_0 + o(m)$ bits of space and needs $O(m)$ for access, select, and rank, where $m = \text{popcount}(t)$ (see also [12], [22], [25], [27], [37]).

The idea of EF is to encode the characteristic vector $\{x_1, x_2, \ldots, x_m\}$ of t, where $x_i = select_1(t, i) : i \in \{1, \ldots, m\}$, instead of t itself. EF uses a technique called MSB bucketing: group x_is according to the most significant $\log m$ bits into buckets, store the $l = \log n - \log m = \log \frac{n}{m}$ lower-order bits for each x_i verbatim in an array (called the Lower-bits Array, *LBA*), and store the significant bits as a sequence of unary encoded gaps in another array (the Upper-bits Array, *UBA*) as follows: for each bucket write down as many 1s as there are x_is in the bucket followed by a 0.

Perhaps an example is in order here. In Fig. 3, $x_1 = 5$, $x_2 = 7$ and $x_3 = 13$, $n = 16$ and $m = 3$, so $l = \lfloor \log {}^{16}/_{3} \rfloor = 2$. This means that the LBA will contain the lower $l = 2$ bits of each x_i verbatim. Further, the bucket size is $2^l = 4$, and so the number of x_is in each bucket is $0, 2, 0, 1$, whose unary encoding gives the UBA: 0110010 (the last 0 can be omitted).

Storing the m elements of the LBA takes $m \log \frac{n}{m}$ bits while the UBA needs $2^{\log m} + m = O(m)$ bits, as there are as many 0s as there are buckets plus m bits set to 1 for each x_i in $i \in \{1, \ldots, m\}$. Finally, we need an additional $\log m$ bits to store m, which we omit here for reasons that will be made clear later. The overall size of EF is $m \log \frac{n}{m} + 2^{\log m} + m = nH_0 + O(m)$ bits, where the data component (the LBA) takes nH_0 bits and the index (the UBA) takes another $O(m)$.

Now, answering $\text{access}(t, i)$ goes as follows. First, we find the bucket q that contains position i: $q = \frac{i}{2^l}$, then we find the run of 1s in the UBA that corresponds to the q-th bucket: $z = \text{select}_0(UBA, q)$; we observe that there were exactly $z - q$ occurrences of 1s in the UBA *before* position z so we scan the LBA *leftward* from position $(z - q)l$, decoding at most 2^l elements x_j of the characteristic vector; if for some $x_j = i$ then the result of the query is 1, otherwise 0. For instance, $\text{access}(t, 6) = 0$ in Fig. 3, as position 6 is in the second bucket thus the MSB is 01, $q = \text{select}_0(UBA, 2) = 4$ so up until the end of the second bucket there were $4 - 2 = 2$ bits set to 1, and decoding the LBA from the second entry leftward, combined with the MSB 01, yields first 7 and then 5, at which point

we know the answer is 0. This goes in $O(m)$ time, as just answering the first select query may require a linear search on the UBA in the worst case. Note that adding another $O(m)$ bits would guarantee $O(1)$ random access [22], [37], but we disregard this option here as it would double the index size. Solving rank goes similarly, while select is by binary search over the UBA and the LBA, again in $O(m)$ time.

When compared to RRR, EF usually yields larger encoded size. At the extreme, for $p = 0.5$ EF uses $1.5n$ bits, a whopping 50% overhead. Furthermore, the somewhat rigid structure of EF does not provide too much in the way of the space–time trade-off like the one we have seen for RRR. Then again, EF can be very fast depending on the input t, as queries take only $O(\text{popcount}(t))$ steps; this can be a massive win, e.g., for small-entropy input. Our compressed bitvector data structure, R3D3 to be presented next, heavily builds on this property.

III. A Doubly Opportunistic Data Structure

In summary, both RRR and EF are opportunistic data structures that realize significant space savings in the data encoding, with EF yielding potentially faster but larger encodings than RRR. Could we somehow combine RRR and EF into a compressed bitvector scheme that would somehow display the advantages of both simultaneously?

In this section we answer this question in the affirmative. We propose R3D3, a combination of RRR and EF that, in contrast to conventional singly-opportunistic encodings that compress only the data component, attains entropy-constrained size on both the data and the index. Thus, we call R3D3 a *doubly opportunistic* data structure.

A. R3D3

So how can we combine the advantages of RRR and EF? First, RRR's indexing scheme gives very fast $O(1)$ access to block-codes and block-ranks, so we definitely want to keep it. It also offers an elegant way to tune the space–time trade-off: The RRR index size is chiefly shaped by the block size b; the larger the block size the fewer blocks we need, and hence the fewer the costly block pointers and block-ranks. Since these dominate the size (taking $O(n/\log n)$ bits when $b = \log n$), increasing blocks will go to great lengths to save memory on indexing. Unfortunately, this cannot be done with RRR for free, as the access and rank execution times are dominated by the block-coding component's running time $O(b)$.

But what if we substitute the block-coding component with EF? After all, decoding a block b_i requires only $O(c_i)$ steps with EF where, recall, c_i is the class of b_i: $c_i = \text{popcount}(b_i)$, in contrast to the $O(b)$ time complexity of combinatorial unranking; in other words, *EF's efficiency depends fundamentally on the number of* 1*s in a block and not the block size itself*. Hence, we can safely increase the block size b to save space on RRR's indexing until we reach block-coding execution-time parity with RRR, which occurs when $\text{popcount}(b_i) = O(\log n)$. At this point our larger blocks will contain as many 1s as the default block size $O(\log n)$ of RRR and so both will need $O(\log n)$ steps for block-decoding, but we gain significant space on the indexing, thanks to the large blocks. Then again, EF-coded blocks will be slightly larger than in RRR, but the gross space reduction we earn on the index will, hopefully, amply compensate for this loss.

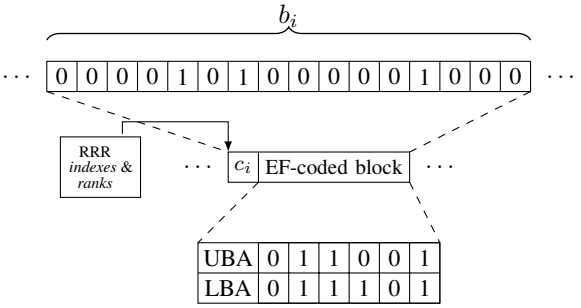

Figure 4: Sketch of the R3D3 encoding scheme, with a single 16-bit block and the corresponding EF block-code. The pointers and rank counters and the block classes are encoded in the RRR index, while the blocks are encoded with EF.

This is the main idea of R3D3: we keep the indexing structure of RRR but we swap the block-coding component for the much more efficient EF. Then, we can increase blocks way beyond what RRR would admit, without major penalty on query times. The basic structure of R3D3 is given in Fig. 4.

Building the R3D3 encoding goes very similarly to how it happens with RRR, just the block-coder is now EF instead of combinatorial ranking/unranking. First, we divide the input t into superblocks of size s and blocks of size b (we set these parameters later), build the RRR index, encode the class c_i for each block b_i directly and then invoke EF to encode b_i. Note that the input to EF is now the block b_i and the length equals b. Additionally, the number of 1s in b_i (the input parameter m) is exactly c_i, so we do not need to store it separately in EF. To control c_i and get better compression we do the usual trick that if $\text{popcount}(n) > \frac{n}{2}$ then we encode the inverse of t instead of t. In fact, in our implementation we do this trick block-wise [33], which yields $p \leq \frac{1}{2}$ and $c_i \leq \frac{b}{2}$.

In fact, R3D3 adopts a scheme we call *duplicate indexing*; it first invokes the RRR indexes to find the starting position for each block, then looks up the UBA to index the relevant entries in the LBA, and finally only a few LBA entries need to be directly decoded. As the analysis in next section reveals, this duplicate indexing scheme yields a highly space- and time-efficient compressed bitvector data structure.

B. Analysis

We fix the superblock size at $s = b \log n$, like in RRR (see the proofs in the Appendix for the reason); the block size b will be determined later. With this parameter setting, the result below gives the storage space and the query times for R3D3.

Theorem 1. *Let t be a bitvector of length n, let $p = \text{popcount}(t)/n$, let H_0 be the zero-order empirical entropy of*

t, and fix the block size at b. Then, encoding t with R3D3 needs at most

$$nH_0 + np + \frac{n}{b}\left(2 + 3\log b + 2\log\log n\right)$$

bits and supports access *and* rank *queries in expected $O(pb)$ time and* select *queries in $O(\log n)$ if $pb = o(\log n)$.*

We give the proof of Theorem 1 through a sequence of technical Lemmas; for clarity the proofs of the Lemmas in turn will be relegated to the Appendix.

The below Lemma characterizes the encoded size M_I of the RRR index structure that we embed into R3D3.

Lemma 1. *The RRR index needs $M_I = \frac{n}{b}(2 + 3\log b + 2\log\log n)$ bits.*

M_I is of course the redundancy in R3D3. Next, we give the size of the EF-coded blocks, M_D.

Lemma 2. *The EF-encoded data needs $M_D = nH_0 + np$ bits.*

Finally, the query execution times stated below for R3D3 are as follows: for $\mathrm{access}(t, i)$ locating the beginning of the EF-coded block that contains position i and identifying the class take $O(1)$ time, to which block-decoding adds another $O(pb)$ for the "average" block. The same holds for $\mathrm{rank}(t, i)$, while $\mathrm{select}(t, i)$ goes with binary-searching superblock and block ranks in $O(\log n)$ time and then decoding the block, again in expected $O(pb)$ time. The total time $O(\log n) + O(pb)$ is dominated by the binary-search as long as $pb = o(\log n)$.

Lemma 3. *Answering* access *and* rank *queries on the R3D3 representation needs expected $O(pb)$ time and* select *goes in $O(\log n)$ as long as $pb = o(\log n)$.*

This completes the proof of Theorem 1. What remains to be done is to fine-tune the block size b. This needs to be done very carefully; increasing b makes for smaller index M_I and hence smaller overall size (the data part M_D is by and large independent of b), but increasing b too much deteriorates query time. We need to strike a fine balance between space and time here, one that results in entropy-constrained size for both M_D and M_I but still does not ruin query performance.

We introduce a new parameter $C = pb$, which can be broadly interpreted as the "average" popcount of blocks. Of course, $C \geq 1$ to ensure that there is at least one bit set in each block. Thus, $b = C/p$ and we immediately get the execution times for access and rank as $O(pb) = O(C)$. Then again, C must not be too large, that is, beyond $O(\log n)$, otherwise select suffers. This gives the useful range $C \geq 1$, $C = o(\log n)$. The following result summarizes these findings.

Theorem 2. *Let t be a bitvector of length n and entropy H_0, and let $C \geq 1$, $c = o(\log n)$. Then, encoding t with R3D3 needs at most*

$$nH_0 + nH_0\left(\frac{1}{2} + O\left(\frac{\log C}{C}\right)\right) \tag{1}$$

bits and supports access *and* rank *in expected $O(C)$ time and* select *in $O(\log n)$.*

Again, consult the Appendix for the proof.

C. Discussion

We close this Section with some remarks on R3D3.

First, R3D3 achieves entropy-constrained space on both the data and the index (up to a small error term for the index): the RRR index and the UBA components in the block-codes, which, as per duplicate indexing, together make up the R3D3 index, need $nH_0\left(1/2 + O\left(\log C/C\right)\right)$ bits of storage space, while the data component (the LBAs) uses another nH_0 bits. As far as we are aware of, R3D3 is the first such doubly opportunistic compressed data structure.

Second, the above space bounds are strictly of worst-case nature, in that there are much tighter upper bounds than what we used in Theorem 2. Since $nH_0 + np \ll \frac{3}{2}nH_0$ when p is sufficiently small, the space bounds can be improved to $nH_0 + nH_0 O\left(\log C/C\right)$ bits if $p < 0.169$, a substantially tighter space characterization for low-entropy input.

Third, tuning constant C opens the door to a wide spectrum of space–time trade-offs. At one extreme, when $C = 1$, i.e., when there is only a single bit set per block on average, we get very fast $O(1)$ access and rank at the cost of a somewhat largish $nH_0\left(3/2 + O(1)\right)$ bits memory footprint, an overhead of $\sim$ 50%. This is because EF-coded blocks are slightly larger than RRR's blocks. On the other hand, increasing C will result larger blocks and less overhead for indexing; when $C = O(\log n)$ we get execution-time parity with RRR with much smaller $nH_0(\frac{1}{2} + O\left(\log\log n/\log n\right)$ bits indexes.

Finally, we observe that our results are in line with the lower bounds of [26], stating that we need $\Omega(\frac{\log\log n}{\log n})$ bits index to implement rank in $O(1)$. R3D3, however, gives $O(1)$ index size in this setting.

IV. Numerical Evaluations

Next, we turn to present a comprehensive set of experimental results to evaluate the space- and time-efficiency of R3D3. For this purpose, we created a proof-of-concept prototype on top of the Succinct Data Structure Library (SDSL, [33]), a powerful C++ template toolkit with comprehensive support for the state-of-the-art in compressed data structures. Stock SDSL offers only the unindexed version of RRR, therefore we created 3 additional C++ template classes on top of SDSL: indexed RRR plus indexed and unindexed versions of R3D3. In the rest of this section RRR and R3D3 will refer to the indexed versions. The R3D3 block-coding routines, furthermore, use the EF optimizations as described in [37]. The code is available at [38].

The two dimensions of interest are the compressed size and performance of queries for RRR and R3D3. We used the CPU's `RDTSC` register, holding the actual snapshot of the program counter, to measure execution times with (close-to) cycle-level precision. The experiments were conducted on a Linux PC, Intel Core i3 CPU @ 3.3GHz with 4Gbyte of RAM.

Block-coding. The goal of our first experiment is to validate our choice for EF instead of RRR's combinatorial ranking/unranking scheme to encode blocks. Recall, this choice

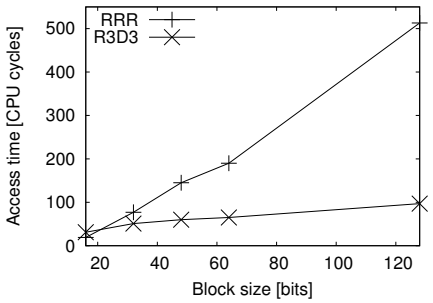

Figure 5: Average time to access a random position in RRR and EF block-codes as the function of the block size, on random bitmaps, $p = 0.1$.

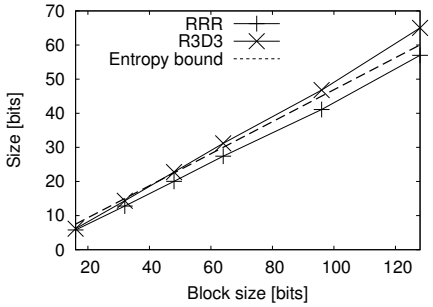

Figure 6: Average size of RRR and EF block-codes and the zero-order entropy limit (dashed line) as the function of the block size, on random bitmaps, $p = 0.1$.

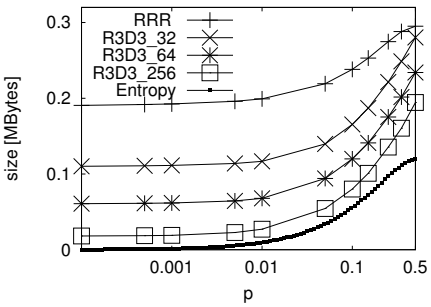

Figure 7: Average size of random bitmaps compressed with RRR and R3D3, and the corresponding entropy limit, as the function of p.

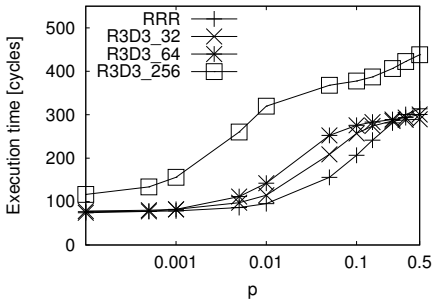

Figure 8: Average time of a random access query on random bitmaps.

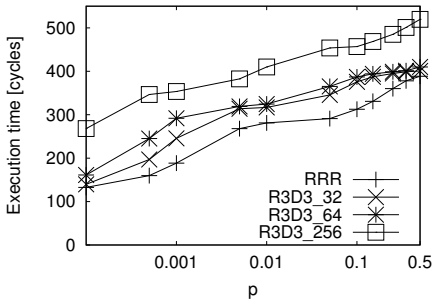

Figure 9: Average time of a random rank query on random bitmaps.

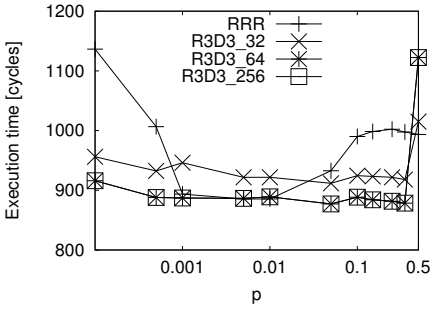

Figure 10: Average time of a random select query on random bitmaps.

was made because EF supports all basic block-operations in $O(\text{popcount}(b))$ time as opposed to $O(b)$ for RRR, where b is the block size, at the cost of slightly bigger block-codes. Note that the population of the block does not alter the relation between EF and combinatorial encodings.

We made 1 million trials, each time with a new random block generated by setting each bit to 1 independently with probability $p = 0.1$. Fig. 5 gives the average time to make a random access to the encoded blocks and Fig. 6 compares the average size of block-codes, as the block size increases from 16 to 128.

We observe that EF block-coding is indeed much less sensitive to the block size; while R3D3 needs only 3 times as much time to access a 128-bit block as for a 16-bit block, this factor is 25-fold with RRR. Furthermore, R3D3 produces only slightly larger blocks than RRR and both are comfortably close to the entropy bound (that RRR beats the entropy limit is not surprising, as combinatorial ranks are a maximally space-efficient universal code, plus our results do not account for the storage size of the class bits c_i). This seems a price it is well worth paying for more efficient block-(de)coding at higher block sizes as the reduced indexes will greatly compensate for this loss, as revealed in our next experiments.

Random synthetic bitmaps. For this experiment we stay at random bitmaps as input, but now we evaluate RRR and R3D3 *en bloc*, not just the block-coding components as previously. We generated 1 Mbit random bitmaps with increasing p from 0 to $1/2$ and we evaluated space and time characteristics of our compressed bitvectors; Fig. 7 gives the size and Fig. 8, Fig. 9, and Fig. 10 give the execution time for access, rank, and respectively select queries to random positions, averaged over 10 trials. We repeated the experiments for R3D3 at different settings for the block size: $b = 32$, $b = 64$, and $b = 256$, while for RRR we used the default setting $b = 16$.

On the storage size front, R3D3 exhibits huge gains over RRR. Even at $b = 32$ we already see two-fold reduction, while the setting $b = 64$ yields fourfold and $b = 256$ a whopping 4–10-fold improvement. At this point, R3D3 compresses very close to the entropy limit. On the other hand, the performance figures are slightly worse with R3D3; access is at most 20% and rank is at most 23% faster with RRR than with R3D3 when the block size is 32, the figures are 30% for access and 35% for rank at $b = 64$, and 30–70% on access and 10–60% on rank at $b = 256$. The performance difference manifests itself only on a limited regime of inputs and in the majority of the examined cases RRR and R3D3 produced remarkably similar performance figures. Finally select times are slightly better with R3D3, especially at larger block sizes.

Real data. We repeated the previous experiment, but this time over real data taken from real-life applications. For the first experiment we collected *bitmaps* from various sources of everyday engineering practice:

Table I: Comparison of RRR or R3D3 on real bitmaps: sample name, size, and entropy bound (nH_0); and compressed size and average execution time of random access and rank queries. Sizes are in Mbytes (MiB) and times in number of CPU cycles.

			RRR			R3D3_32			R3D3_64			R3D3_256		
Name	Size	Entropy	Size	Access	Rank	Size	Access	Rank	Size	Access	Rank	Size	Access	Rank
fax	0.49	0.19	0.86	106	205	0.56	112	256	0.37	125	267	0.2	210	328
bmp1	4.15	1.16	7.1	90	142	4.40	96	160	2.68	97	176	1.23	149	238
bmp2	4.15	1.69	7.46	121	223	4.99	141	278	3.34	150	290	2.06	236	357
zip	0.011	0.011	0.025	223	300	0.023	240	336	0.019	265	348	0.017	365	456
caida_4	3.38	0.19	5.6	82	145	3.28	87	195	1.84	95	243	0.62	146	328
caida_8	1.08	0.14	1.81	89	176	1.08	97	238	0.63	102	279	0.25	172	345
caida_16	0.34	0.09	0.58	104	211	0.37	114	282	0.23	124	297	0.11	209	354

Table II: Comparison of RRR or R3D3 on real textual data: sample name, size, and entropy bound (nH_0); and compressed size and average execution time of random access and rank queries. Sizes are in Mbytes (MiB) and times in number of CPU cycles.

			RRR			R3D3_32			R3D3_64			R3D3_256		
Name	Size	Entropy	Size	Access	Rank	Size	Access	Rank	Size	Access	Rank	Size	Access	Rank
shakes	0.119	0.068	0.18	3860	1495	0.17	3402	1614	0.14	3473	1639	0.115	4477	1802
scifi	0.733	0.401	1.04	3619	1523	0.97	3219	1625	0.79	3280	1658	0.65	4201	1809
bible	3.86	2.06	5.26	3451	1504	4.83	3094	1632	3.99	3156	1655	3.26	4021	1806
chr7	10	2.5	6.61	1518	771	5.9	1427	825	4.79	1441	836	3.88	1840	915
chr22	3.73	0.92	2.45	1523	778	2.17	1427	829	1.77	1433	841	1.42	1825	913
coli	4.42	1.11	2.75	1482	769	2.57	1379	822	2.14	1403	839	1.77	1813	923
euler	1.91	0.79	2	2623	1146	1.83	2378	1210	1.51	2404	1238	1.23	3077	1368
pi_1M	0.95	0.39	1.02	2640	1159	0.93	2401	1208	0.76	2430	1241	0.62	3107	1360
pi_10M	9.54	3.96	10.23	2643	1161	9.28	2403	1229	7.62	2429	1242	6.19	3103	1357

Table III: Comparison of RRR or R3D3 on routing tables: sample name, number of prefixes, and entropy bound as of [31]; and compressed size and average execution time of random FIB lookups. Sizes are in Kbytes (KiB) and times in CPU cycles.

			RRR		R3D3_32		R3D3_64		R3D3_256	
Name	#Prefixes	Entropy	Size	Lookup	Size	Lookup	Size	Lookup	Size	Lookup
hbone-szeged	453,685	70.1	172.7	11468	145.8	9092	116.7	10444	93.2	14724
access_d	403,245	149.1	226.1	10828	193.7	9576	155.4	10268	123.8	13492
access_v	2,970	1.08	7.6	5672	7.4	5448	6.6	6772	6.4	7796
mobile	4,391	1.32	3.7	6760	3.8	6488	3.6	7260	3.5	7588
hbone-vh1	453,741	222.6	418.5	9248	362	7600	293.7	7556	238.1	9264

- fax: 1728x2376 bitmap image of text and diagrams from the Calgary Corpus [39];
- bmp1, bmp2: bilevel bitmap images scanned at 600dpi;
- zip: US ZIP codes in bitmap format, 1 marks a valid and 0 marks an invalid ZIP code;
- caida_4, caida_8, caida_16: adjacency matrices of the 4, 8, and 16-core of the Internet AS-level map in bitmap format, as obtained from CAIDA on 2014-06-01.

The results are given in Table I. The first surprising observation is that not just that RRR does not reach the entropy limit but it completely fails even the uncompressed size. This is due to the excessive size of the index that we need to store to allow queries into the compressed data. R3D3, on the other hand, attains at least the uncompressed size at $b = 32$, improving on RRR by a factor of 2 in most cases. Increasing the block size to 64 then decreases the size by another factor of 2, while at $b = 256$ R3D3 gets very close to the entropy limit, improving over RRR by around a factor of 8. Meanwhile, the performance of queries with R3D3 remains comfortably close to that for RRR: at $b = 32$ the access execution times are on par and rank is at most 30–40% slower, while at $b = 256$ we get roughly half the performance of RRR. Recall, this is in return to about 8 times smaller size.

We repeated the experiments with *textual data*, this time compressing the input using Huffman-shaped wavelet trees [16]. Since a wavelet tree is essentially just a collection of bitmaps organized into a tree structure and access and rank queries translate to those on these bitmaps, wavelet trees nicely exercise the underlying bitvector encoders. The inputs:

- shakes, scifi and bible: excerpts from Shakespeare's plays, a science-fiction novel, and the Bible, all in English;
- chr7, chr22, and coli: genome sequences from the human Chromosome 7 and 22, and E-coli bacteria, downloaded from the UCSC Genome Browser [40];
- euler, pi_1m, and pi_10m: first 2 million digits of the Euler constant, and 1 and 10 million digits of π.

The results are in Table II. It seems that *on real text inputs random access is consistently faster with R3D3 than with RRR at moderate block sizes* while rank performance is similar, and

even at $b = 256$ we see only a minor performance hit. This is most probably due to the larger inputs and thereby CPU cache performance dominating query times. Interestingly, access ran slower than the more complex rank queries.

This experiment spectacularly highlights the benefits of data compression for operating on large quantities of data: it simultaneously delivers significant space savings over an uncompressed representation *and* implements fast operations on the content, permitting powerful queries of the type "How many times digit 5 occurs in π until the $500,000$-th position?" ($rank_5(\pi, 500000)$) or "Which is the 500-th valid ZIP code?" ($\mathrm{select}_1(\mathrm{zip}, 500)$), which a naive uncompressed representation does not even support out of the box.

These observations are further confirmed by our experiments on real Internet forwarding tables (see Table III). We used the *XBW* scheme of [31], a pair of a bitvector and a wavelet tree that together encode a prefix tree, to compress real FIB instances taken from operational Internet routers. Again, R3D3 approaches the entropy at larger block sizes and beats RRR multiple times, and it supports longest-prefix matches faster than the RRR-based encoding at roughly 5 times the speed as reported in [31].

V. Conclusion

Throughout the recent years, compressed data structures have gained wide-spread adoption in information retrieval, computational geometry, bioinformatics, networking, and big data. This is on the one hand due to their potential for making it possible to operate on unprecedentedly huge instances of data and, on the other hand, because they support much more complex queries to the compressed data, like rank and select, with zero performance impact. In many cases compression creates a win-win situation, as the memory footprint of large bodies of information can be freely decreased and meanwhile processing may even get faster, thanks to the data drifting closer to the CPU in the cache hierarchy.

In this paper, we have proposed R3D3 as a new tool for compressing and indexing bitvectors. R3D3 is, in contrast to previous work, doubly opportunistic, in that it realizes substantial space savings on the compressed data and the index alike. Furthermore, it allows to strike a fine space–time balance as required by the application at hand, with a smooth transition between the extremes. We have shown that most benefits already manifest themselves at moderate block sizes, realizing several times smaller encodings at only a slight performance impact compared to the state-of-the-art compressed bitvector scheme, RRR. At the extreme, for very large blocks R3D3 may provide 10-fold space reduction over uncompressed data and over RRR, in exchange of at most 50% performance penalty. Notably, on real data R3D3 proved faster than RRR. And because underlying most data indexing schemes, like compressed text indexes or compressed labeled trees, there is a bitvector data structure behind the scenes, the benefits of R3D3 also appear when compressing complex information, like small entropy textual data or genomes.

References

[1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big data: The next frontier for innovation, competition, and productivity," McKinsey Global Institute, Tech. Rep., June 2011.

[2] O. Trelles, P. Prins, M. Snir, and R. C. Jansen, "Big data, but are we ready?" *Nat Rev Genet*, vol. 12, no. 3, p. 224, 2009.

[3] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, ser. Series in Computer Science. Addison Wesley, 1998.

[4] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15, 2015, pp. 337–350.

[5] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Comput. Surv.*, vol. 39, no. 1, 2007.

[6] G. Jacobson, "Space-efficient static trees and graphs," in *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, ser. SFCS '89, 1989, pp. 549–554.

[7] W.-K. Hon, K. Sadakane, and W.-K. Sung, "Succinct data structures for searchable partial sums," in *Algorithms and Computation: 14th International Symposium*, 2003, pp. 505–516.

[8] C. K. Poon and W. K. Yiu, "Opportunistic data structures for range queries," in *Computing and Combinatorics: 11th Annual International Conference*, 2005, pp. 560–569.

[9] J. Barbay, M. He, J. I. Munro, and S. R. Satti, "Succinct indexes for strings, binary relations and multilabeled trees," *ACM Trans. Algorithms*, vol. 7, no. 4, pp. 1–27, 2011.

[10] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao, "Representing trees of higher degree," *Algorithmica*, vol. 43, no. 4, pp. 275–292, 2005.

[11] R. F. Geary, R. Raman, and V. Raman, "Succinct ordinal trees with level-ancestor queries," *ACM Trans. Algorithms*, vol. 2, no. 4, pp. 510–534, 2006.

[12] V. R. R. Raman and S. R. Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets." *ACM Transactions on Algorithms*, vol. 3(4), 2007.

[13] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Compressing and indexing labeled trees, with applications," *J. ACM*, vol. 57, no. 1, pp. 1–33, 2009.

[14] F. Claude and G. Navarro, "A fast and compact web graph representation," in *String Processing and Information Retrieval: 14th International Symposium*, 2007, pp. 118–129.

[15] P. Elias, "Efficient storage and retrieval by content and address of static files," *J. Assoc. Comput. Mach.*, vol. 21, pp. 246–260, 1974.

[16] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '03, 2003, pp. 841–850.

[17] A. Golynski, J. I. Munro, and S. S. Rao, "Rank/select operations on large alphabets: A tool for text indexing," in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, ser. SODA '06, 2006, pp. 368–373.

[18] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "Compressed representations of sequences and full-text indexes," *ACM Trans. Algorithms*, vol. 3, no. 2, 2007.

[19] T. Gagie, G. Navarro, and S. J. Puglisi, "New algorithms on wavelet trees and applications to information retrieval," *Theor. Comput. Sci.*, vol. 426-427, pp. 25–41, 2012.

[20] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, ser. FOCS '00, 2000.

[21] K. Sadakane, "Succinct data structures for flexible text retrieval systems," *J. of Discrete Algorithms*, vol. 5, no. 1, pp. 12–22, 2007.

[22] S. Gog, "Compact and succinct data structures: From theory to practice," available online: http://es.csiro.au/ir-and-friends/20131111/anu_gog_seminar.pdf, 2015.

[23] M. C. Brandon, D. C. Wallace, and P. Baldi, "Data structures and compression algorithms for genomic sequence data," *Bioinformatics*, vol. 25, no. 14, pp. 1731–1738, 2009.

[24] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel, "Iterative dictionary construction for compression of large DNA data sets," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 9, no. 1, pp. 137–149, Jan 2012.

[25] R. Raman, "Succinct data structures for data mining," Workshop on Algorithms for Large-Scale Information Processing in Knowledge Discovery, 2014.
[26] P. B. Miltersen, "Lower bounds on the size of selection and rank indexes," in *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '05, 2005, pp. 11–12.
[27] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, 2007, pp. 60–70.
[28] S. Vigna, "Broadword implementation of rank/select queries," in *Proceedings of the 7th International Conference on Experimental Algorithms*, ser. WEA'08, 2008, pp. 154–168.
[29] F. Claude and G. Navarro, "Practical rank/select queries over arbitrary sequences," in *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, ser. SPIRE '08, 2009, pp. 176–187.
[30] G. Navarro and E. Providel, "Fast, small, simple rank/select on bitmaps," in *Experimental Algorithms: 11th International Symposium*, 2012, pp. 295–306.
[31] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: towards entropy bounds and beyond," in *ACM SIGCOMM 2013*, 2013, pp. 111–122.
[32] T. M. Cover and J. A. Thomas, *Elements of information theory*. Wiley-Interscience, 1991.
[33] S. Gog, "Succinct Data Structure Library," https://github.com/simongog/sdsl-lite.
[34] D. E. Knuth, *The Art of Computer Programming: Combinatorial Algorithms*, ser. Series in Computer Science. Addison-Wesley, 2011.
[35] V. Makinen and G. Navarro, "Dynamic entropy-compressed sequences and full-text indexes," *ACM Transactions on Algorithms*, vol. 4, 2008.
[36] A. Majdán and G. Rétvári, "Development and performance evaluation of fast combinatorial unranking implementations," in *EUNICE*, 2014.
[37] S. Vigna, "Quasi-succinct indices," in *ACM International Conference on Web Search and Data Mining, WSDM*, 2013, pp. 83–92.
[38] M. Nagy, "Github homepage," https://nmate.github.io, 2019.
[39] I. Witten, T. Bell, and J. Cleary, "The Calgary Corpus," 1987, http://corpus.canterbury.ac.nz/descriptions/#calgary.
[40] UCSC Genome Bioinformatics, "The UCSC Genome Browser," http://hgdownload.cse.ucsc.edu/downloads.html.

APPENDIX

Proof of Lemma 1: The following metadata are stored in the RRR index for each superblock i and each block j:

- P_i: the address of the ith superblock;
- R_i: cummulative rank up to the ith superblock;
- L_{ij}: relative address of block j inside superblock i;
- Q_{ij}: relative rank of block j inside superblock i;
- K_{ij}: the block class $c_j = \text{popcount}(b_j)$.

Both P and R require $\frac{n}{s}\log(n)$ bits, K needs $\frac{n}{b}\log(b)$ bits, while L and Q, both holding values relative to the containing superblock, can use $\log(s)$ bits per block. In total

$$M_I = 2\frac{n}{s}\log n + \frac{n}{b}\log b + 2\frac{n}{b}\log s \quad (2)$$

$$= 2\frac{n}{b}\left(\frac{b}{s}\log n + \frac{\log b}{2} + \log s\right) \ . \quad (3)$$

Now, (2) gives a useful hint on how to select the superblock size: introducing the notation $x = \frac{b}{s}$ we see that (2) is minimal where $\frac{\partial}{\partial x}\frac{n}{b}\left(x\log n + \frac{\log b}{2} + \log\frac{b}{x}\right) = 0$, which gives $x = \frac{1}{\log n}$ and hence for the superblock size $s = b\log n$. With this setting, we get $M_I = {}^{n}/_{b}(2+3\log(b)+2\log\log n)$ as required by the claim of the Lemma. ∎

Proof of Lemma 2: First, we observe that instead of calculating the space occupancy of each block b_i one by one, it is enough to deal with the size of an "average" block with $c = pb$ (the proof is trivial using Jensen's inequality, we omit the details). The UBA stores a bit for each bucket plus another bit for each bit set in the block, yielding $2^{\log c} + c = 2^{\log b - l} + c = \frac{b}{2^l} + c$ bits overall, while the LBA consists of c elements, each of l bits. Summed up for each of the $\frac{n}{b}$ blocks:

$$M_D = \frac{n}{b}\left(\frac{b}{2^l} + c + lc\right) = n\left(2^{-l} + p + pl\right) \ . \quad (4)$$

Recall that the choice for parameter l is elemental in EF; usually $l = \lfloor\log\frac{b}{c}\rfloor$. First, to demonstrate the main idea of the proof we give the treatment for the simplified case when we omit rounding to integers, then we discuss how to handle this discrepancy. Letting $l = \log b - \log c$ firstly yields

$$M_D = n\left(\frac{c}{b} + p + p\log\frac{b}{c}\right) = n\left(p + p + p\log\frac{1}{p}\right) \quad (5)$$

$$\leq n\left(p + (1-p)\log\frac{1}{1-p} + p\log\frac{1}{p}\right) = np + nH_0 \ , \quad (6)$$

by that $p \leq (1-p)\log(\frac{1}{1-p})$ if $p \in (0, \frac{1}{2}]$, as requested.

Secondly, taking care of integrality $l = \lfloor\log\frac{b}{c}\rfloor$ and using that $\frac{b}{c} = \frac{1}{p}$, we write:

$$M_D = n\left(2^{-\lfloor\log\frac{1}{p}\rfloor} + p + p\left\lfloor\log\frac{1}{p}\right\rfloor\right) \ . \quad (7)$$

To prove the statement, it is now enough to show that (5) is larger than, or equal to (7), or, equivalently, that the difference

$$2^{-x} + xp - (2^{-\lfloor x\rfloor} + \lfloor x\rfloor p)$$

is non-negative, where we used the shorthand $x = \log(\frac{1}{p})$. Clearly for $0 \leq x < 1$ the difference equals $2^{-x} + xp - 1$, which is always positive as $2^{-x} \geq 1$ in this range and x and p are positive. Next, we will show that $f(x) = 2^{-x} + xp$ is a decreasing function of x for $x \geq 1$. Substitute $p = 2^{-x}$ to get

$$f(x) = 2^{-x} + x2^{-x} = 2^{-x}(1+x)$$

Finally, we need to show that the derivate is negative:

$$\frac{\partial f(x)}{\partial x} = 2^{-x} - 2^{-x}(1+x)\ln(2) = 2^{-x}\big(1 - (1+x)\ln(2)\big).$$

Clearly, 2^{-x} is positive and $1 - (1+x)\ln(2)$ is negative for $x > \frac{1}{\ln(2)} - 1 \cong 0.44$. This completes the proof. ∎

Proof of Theorem 2: We only need to show that $M_I + M_D$ is as required. Write $M_I = \frac{2n}{b} + \frac{3n}{b}\log b + \frac{2n}{b}\log\log n$ and substitute $b = \frac{C}{p}$ to get $\frac{2np}{C} + \frac{3np}{C}\log\frac{C}{p} + \frac{2pn}{C}\log\log n$. Using that $p \leq \frac{1}{2}$ and so $p\log\frac{1}{p} \leq H_0$ and $2p \leq H_0$, we write for the first component $\frac{2pn}{C} \leq nH_0\frac{1}{C}$, for the second $\frac{3np}{C}\log\frac{C}{p} = \frac{n}{C}\left(3p\log\frac{1}{p} + 3p\log C\right) \leq \frac{n}{C}3H_0 + \frac{n}{C}3p\log C = nH_0O(\frac{\log C}{C})$, and for the third $\frac{n}{C}2p\log\log n = \frac{n}{C}2pO(\log C)$ (by that $C = O(\log n)$) and thus $nH_0O(\frac{\log C}{C})$ using the same substitutions as before. Thus, $M_I = nH_0O(\frac{\log C}{C})$ and $M_D = nH_0 + np \leq nH_0 + \frac{1}{2}nH_0$, yielding the overall size $M_I + M_D = nH_0 + nH_0\left(\frac{1}{2} + O(\frac{\log C}{C})\right)$ bits, which completes the proof. ∎