PaCHash: Packed and Compressed Hash Tables

Florian Kurpicz* Hans-Peter Lehmann† Peter Sanders‡

Abstract

We introduce PaCHash, a hash table that stores its objects contiguously in an array without intervening space, even if the objects have variable size. In particular, each object can be compressed using standard compression techniques. A small search data structure allows locating the objects in constant expected time. PaCHash is most naturally described as a static external hash table where it needs a constant number of bits of internal memory per block of external memory. Here, in some sense, PaCHash beats a lower bound on the space consumption of k-perfect hashing. An implementation for fast SSDs needs about 5 bits of internal memory per block of external memory, requires only one disk access (of variable length) per search operation, and has small internal search overhead compared to the disk access cost. Our experiments show that it has lower space consumption than all previous approaches even when considering objects of identical size.

1 Introduction

Hash tables support constant time key-based retrieval of objects and are one of the most widely used data structures. *Compressed* data structures store data in a space efficient way, preferably approaching the information theoretical limit, and support various kinds of operations without the need to decompress the entire data structure first [29, 1, 25, 58]. There has been intensive previous work on both subjects but, surprisingly, the intersection leaves big gaps. There is a lot of work on hash tables which need little more space than just the stored objects themselves [35, 8, 3, 26, 34, 51]. However, all these approaches are only space efficient for objects of identical size which makes it impossible to compress the objects with variable bit-length codes. Currently, most hash tables for objects of variable size store references from table entries to the data which entails a space overhead of at least $\log N$ bits per object, where N is the total size of all objects in the table. Throughout this paper, $\log x$ stands for $\log_2 x$. See Section 2 for an introduction of basic techniques and Table 1 for a summary of the notation.

PaCHash eliminates fragmentation by *packing* the objects contiguously in memory without leaving free space. This makes it impossible to use the approach of most previous hash tables to directly use the hash function value to (approximately) locate the objects. Instead, PaCHash uses a highly space efficient search data structure that translates hash function values to memory locations. More precisely, objects are first hashed to *bins*. The bins are stored contiguously in m *blocks* of size $\mathtt{B}$. PaCHash essentially stores one bin index per block using a searchable compressed representation which enables finding the block(s) where a bin is stored. In Section 4, we describe the data structure in more detail and in Section 5 we analyze it. Basically, for a tuning parameter a, the expected number of block reads to retrieve an object x of size $|x|$ is about $1 + 1/a + |x|/\mathtt{B}$ while the internal memory data structure needs $2 + \log(a)$ bits per *block*. We also discuss even smaller representations.

Even though hash tables like PaCHash have applications in object stores, there is little previous work on space efficient hash tables for objects of variable size (see Section 3). For objects of identical size s, the most space efficient previous solutions are based on minimal perfect hashing (MPH) [20, 7] and require a constant number of bits per object. PaCHash approximates this when choosing $\mathtt{B} = s$, also needing a (slightly larger) constant number of bits per object but lower construction time. The picture changes when we look at larger block sizes $\mathtt{B} = ks$ and the corresponding approach of *minimal k-perfect hashing (MkPH)* [7]. Now, PaCHash still needs only a constant number of bits per block, while there is a *lower bound* of $\Omega(\log k)$ bits per block using MkPH (see Section 5).

Another fundamental data structure related to variable size objects and PaCHash is the variable-bit-length array (VLA). A VLA is an array that allows direct access to objects of variable size. Oftentimes, VLAs are used to efficiently access variable-length codes, e.g., Elias-γ and -δ codes [19] or Golomb codes [30], see Section 3.

Section 6 describes different implementation variants of PaCHash including fully internal and fully external versions as well as a variant that is usable as VLA. Section 7 describes experiments for an external implementation. Section 8 summarizes the results and discusses possible directions for further research.

*Karlsruhe Institute of Technology, Germany.
†Karlsruhe Institute of Technology, Germany.
‡Karlsruhe Institute of Technology, Germany.

Table 1: Symbols used in this paper

S	Set of objects
n	Number of objects
N	Total size of objects (bits)
p	Internal index data structure
a	Tuning parameter: Bins per block
$m = N/\bar{\mathtt{B}}$	Number of blocks
$\mathtt{B}$	Block size (bits)
$\bar{\mathtt{B}} = \mathtt{B} - d$	Payload data per block
$d \in 0..\log \mathtt{B}$	Encoding-dependent number of bits to store position of first bin of block

Our Contribution. In this paper, we design the new hash table PaCHash. The data structure supports objects of variable size with space overhead close to competitors that only support objects of identical size. We analyze it thoroughly in a variant of the external memory model. Finally, we compare our implementation with competitors from the literature. As close contenders, we also implement *Separator Hashing* [31, 39] and *Cuckoo Hashing* [5, 50] with adaptions that partially allow variable size objects.

2 Preliminaries

Monotonic Sequences and Bit Vectors. The index data structure of PaCHash mainly consists of a compressed representation of a monotonically increasing sequence $p = \langle p_1, \ldots, p_k \rangle$ of integers in the range $1..U$. Searching boils down to predecessor queries in p, i.e., given a query integer i, the largest sequence element $\leq i$ is returned.

A well-known practical solution is *Elias-Fano coding* [19, 23] which splits each p_i. The $\log(U/k)$ least significant bits are directly stored in an array L requiring $k \log(U/k)$ bits of space. The $\log(k)$ most significant bits form a monotonic sequence of integers $H = \langle u_1, \ldots, u_k \rangle$ in the range $0..k$. H is stored in a bit vector of size $2k+1$ where u_i is represented as a 1-bit in position $i + u_i$. The total space usage therefore is $k(2 + \log(U/k)) + 1$ bits. A predecessor query in p executes a $select_0$ query in H (finding the i-th 0-bit in H) which locates a cluster of entries in L that must contain the sought element. Using additional space $o(k)$, $select_0$ queries can be answered in constant time [12]. In contrast to the general case, we will show that searching the cluster takes expected constant time in our application.

One can also interpret p as the positions of 1-bits in a sparse bit vector which enables even more compact representations. For example, using Succincter [52], about $k(1.44 + \log(U/k)) + 1$ bits are achievable which is almost information theoretically optimal. In Section 4.2.3, we give an even more compact format exploiting additional structure in the bit vector.

Model of Computation. We describe our results in a variant of the external memory model [57] adapted to a situation where objects are compressed to variable length sequences of bits. We have a *fast memory* of size $\mathtt{M}$ *bits*. Accesses to a large *external memory* are *I/Os* to blocks of $\mathtt{B}$ consecutive bits. In contrast to the original model, we analyze both I/Os and internal work. $scan(N)$ denotes the cost (I/Os *and* internal work) of scanning N bits of data.[1] $sort(N)$ denotes the cost of sorting N bits.[2] In particular, we are interested in a high load factor, which is N divided by the total external space usage.

3 Related Work

The following section introduces related data structures from the literature. Table 2 provides an overview over the most important parameters. There are close contenders in the form of *object stores* from the database literature. BerkeleyDB [48] uses a B$^+$-Tree [15] of order d, where each node branches between d and $2d$ times. LevelDB [32] and RocksDB [21] use a Log-Structured Merge tree [49], which stores multiple levels of a static data structure with increasing size. Insertions go into the first level and when a level gets too full, it is merged into the next level. SILT's *LogStore* [41], Facebook *Haystack* [6] and *FAWN* [2] simply store a pointer of size $\Omega(\log N)$ to each object. Real world instances often store very small objects [47], so the pointers add a considerable amount of overhead.

Sorted Objects. *LevelDB*'s static part [32] stores objects in key order, enabling range searches and common-prefix-compression. *SortedStore* in SILT [41] sorts the objects by their hashed key and uses entropy coded tries as an index. Pagh [50] proposes to sort the n objects by a hash function with range $\geq n^3$. The internal memory stores the first hash function value mapped to each block. This data structure can be queried using a predecessor data structure in time $O(\log \log n)$. A novel idea in PaCHash is that it uses a hash function range based on the total space N instead of the number of objects n, which enables efficient queries and compact representation.

[1] The internal work may depend on the encoding of the data. For example, we may need $\Theta(N)$ machine instructions, or, a faster encoding may enable bit-parallel processing in $O(N/\log n)$.

[2] This entails $(N/\mathtt{B})(1 + \lceil \log_{\mathtt{M}/\mathtt{B}}(N/\mathtt{M}) \rceil)$ I/Os. In this paper algorithms with linear internal work are possible exploiting random integer keys. The cost also includes (de)coding overhead as in *scan* operations.

Table 2: Space efficient object stores from the literature. To unify the notation, we convert all values so that they refer to objects of size $s = 256$ bytes stored in blocks of $\mathtt{B} = 4096$ bytes. The load factor α is given in percent and the internal space I_b in bits per *block* of $\mathtt{B}/s = 16$ objects. Top: Stores for objects of identical size. Can be used for objects of variable size by using indirection or for some methods by accepting significantly lower load factors. Bottom: Dedicated variable size object stores. This table also contains VLAs, even though those are a slightly different field.

	Method	I_b	α	I/Os
fixed size	Extendible Hashing [22]	$\log m$	90	1
	Larson et al. [40]	96	<96	1
	SILT SortedStore [41]	51	100	1
	Linear Separator [38]	8	85	1
	Separator [31, 39]	6	98	1
	Robin Hood [10]	3	99	1.3
	Ramakrishna et al. [54]	4	80	1
	Jensen, Pagh [33]	0	80	1.25
	Cuckoo [5, 50]	0	<100	2
	PaCHash, $a = 1$	2	100	2[3]
	PaCHash, $a = 8$	5	100	1.13[3]
variable size	SILT LogStore [41]	832	100	1
	Külekci [36] (VLA)	176	<100	0–11[4]
	SkimpyStash [17]	32	≤98	8
	Blandford, Blelloch [9] (VLA)	16	≤50	1
	PaCHash, $a = 1$	2	99.95	2.06[3]
	PaCHash, $a = 8$	5	99.95	1.19[3]

External Hash Tables. In external hash tables, each table cell corresponds to a fixed size block. A common technique to support variable size objects is using indirection by internally storing a pointer to the object contents, possibly inlining parts of the objects [41, Section 4]. NVMKV [45] and KallaxDB [11] use an SSD as one large hash table and rely on SSD internals to handle empty blocks in a space efficient way. Overflowing blocks due to hash collisions can be handled with perfect hashing [40, 54] or using one of the following techniques.

With *Hashing with Chaining*, objects of overflowing blocks are stored in linked lists. *SkimpyStash* [17] chains objects using an external successor pointer for each object. This trades internal memory space for latency because of multiple dependent I/Os. Jensen and Pagh's [33] data structure reserves parts of the external memory as a buffer to reduce the need for chaining. *Extendible Hashing* [22] keeps a balanced tree of blocks. Overflowing blocks are split into two children indexing one more bit of the hashed key.

Another method for resolving collisions is *open addressing*, where each object could be located in multiple blocks. *Cuckoo Hashing* [51, 18] locates each object in one of two (or more [26]) independently hashed blocks. Queries can load both blocks in parallel to reduce latency. With *Separator Hashing* [31, 39], each object has a sequence of blocks it could be stored in and a corresponding sequence of signatures. When a block overflows, the objects with the highest signature values are pushed out to the next block in their respective sequences. The internal memory stores the highest signature value of the objects placed in each block. A query follows the object's sequence of blocks and stops when it finds a separator that is larger than the corresponding signature. *Linear hashing with separators* [38] is a dynamic variant with a linear probe sequence. *External Robin Hood Hashing* [10] is similar to linear separator hashing, but it instead pushes out objects that are closest to their respective home address. For each block, the internal memory stores the smallest distance of its objects to their respective home address.

Variable-Bit-Length Arrays. Variable-bit-length arrays (VLAs) are arrays containing objects of variable size. VLAs are closely related to PaCHash, which can be used also as VLA by using the array index instead of the hash function, see Section 6. Conversely, PaCHash can be seen as a VLA where each entry stores a PaCHash bin. However, most VLAs have some limitations that rule out storing the PaCHash bins efficiently. A major difference to all VLAs described below is PaCHash allowing objects to span over multiple blocks of fixed size.

Navarro [46, Section 3.2] describes several techniques for implementing VLAs. However, none of them achieves the same favorable space-time trade-off as the PaCHash VLA. The closest one – sampled pointers – needs $N + n \log(N)/k$ bits of space with access cost bounded by the time needed to skip k objects. Note that this time can be large when large objects need to be skipped.[5] All the other described VLAs need several bits of space overhead per object (multiplied with a factor that depends on the maximum or average object size).

The VLA introduced by Külekci [36] uses wavelet trees [24] to partition the universe. This makes the query time depend double logarithmically on the largest element stored in the VLA, a limitation not existing in PaCHash.

[3] PaCHash performs one I/O of variable size which is faster than the competitors' multiple I/Os.

[4] Using 256 byte objects, we have an alphabet size of $2^{8 \cdot 256}$, and $\log \log 2^{8 \cdot 256} = 11$.

[5] Space could be reduced to $N + \frac{n}{k}(2 + \log \frac{kN}{n})$ bit using Elias-Fano coding of the pointers – resulting in similar space as the PaCHash VLA with $B = kN/n$ but with worse access costs.

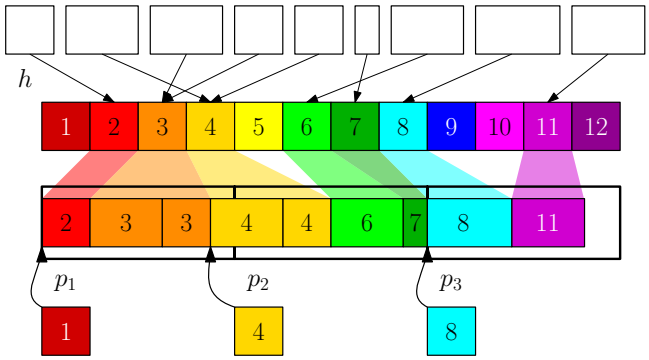


Figure 1: Example of PaCHash with $n = 9$ objects and $m = 3$ blocks. Using the hash function h, the objects are mapped to 12 bins shown as colors, i.e., $a = 4$. The bin content is then contiguously written to the external memory blocks. The internal memory index p stores the first bin intersecting with each block. Note that locating bin 8 will return the range 2..3, i.e., block 2 is loaded superfluously because there is no preceding empty bin that can encode whether it overlaps into the previous block. All other bins are located optimally.

Blandford and Blelloch [9] describe dynamic VLAs and hash tables for variable sized objects. However, their technique incurs a constant factor of space overhead and is limited to objects of bounded size. They partition the objects into blocks, but the blocks are generally only partially filled and do not allow objects crossing block boundaries as in PaCHash.

4 The PaCHash Data Structure

We now present PaCHash in detail – a hash table which considerably improves on the data structures from the literature. It natively supports variable size objects without the need for indirection or empty cells. It needs only a few bits of internal memory per *block* and still needs only one single I/O operation (of variable length) per query. PaCHash consists of an *external part* subdivided into m blocks of exactly $\mathtt{B}$ bits each that store the actual objects and an *internal part* that allows finding the blocks storing an object. Figure 1 gives an example for the external and internal memory data structures. We deliberately use the word *object* for the stored data because that highlights the flexibility of PaCHash. Naturally, an object stores a key-value-pair, but it can also store only a value to obtain an external dictionary data structure. It is even possible to use quotienting by storing the bin index inside the first object of each bin.

4.1 External Object Representation.

PaCHash stores the objects sorted by a hash function h with a rather small domain, namely $h : K \to 1..am$, where K is the set of possible keys, m is the number of blocks and a is a tuning parameter that we assume to be a power of two. The hashes can collide and therefore group the objects into am *bins*. The objects are now basically stored contiguously. "Basically" means that blocks may also contain information needed to find the first object or bin stored in them. Refer to Section 6 for a discussion of alternative encodings. Our default assumption is as follows: Each external block stores an offset of size $d = \log \mathtt{B}$ bits indicating the bit where the first bin in the block starts. The remaining space stores the objects contiguously where an object may have an arbitrary size in bits. No space is left between subsequent objects. In particular, object representations may overlap block boundaries. We assume that objects are encoded in a self-delimiting way, i.e., when we know where an object starts, we can also find its end. For example, we could have a prefix-free code for the objects. Construction first sorts the objects by their hash function value. Then it scans the sorted objects, constructing both the external and the internal data structure along the way. Refer to Section 5 for more details. If the internal data structure gets lost, for example due to a power outage, it can be re-generated using a single scan over the external memory data.

4.2 Internal Memory Data Structure.

Given a bin b, the internal memory data structure p can be used to determine a (near-)minimal range $i..j$ of block indices such that b is stored in that range. When performing a query, that block range can then be loaded from external memory and scanned for the sought key. In practice, the resulting latency is often close to that of loading a single block since it includes only one disk seek. Conceptually, p stores a sequence $\langle p_1, \ldots, p_m \rangle$ where p_i specifies the first bin whose data is at least partially contained in block i.[6] We can use a predecessor query on p to determine i. When the predecessor is b itself, we also need to load the previous block. Another predecessor query or scanning then determines j, as illustrated by the pseudocode in Algorithm 1. To get the most out of this specification, we take empty bins into account: When a bin starts exactly at a block boundary and has an empty predecessor, we store that predecessor. This implies that if (and only if) a bin b starts at a block boundary and the previous bin $b-1$ is nonempty, retrieving bin b will load one block too much.

[6] An alternative would be to store the first bin that *starts* in each block. This introduces a special case when a block is fully overlapped by a bin and needs slightly more work when performing queries.

Algorithm 1: A query for an object x calls locate(x), loads the returned block range, and scans the blocks to find the object content. Determining the range boils down to predecessor queries on p.

Function locate(x)
  $b := h(x)$
  find i such that $p_{i-1} < b \leq p_i$ // predecessor query
  if $p_i = b$ **then** $i := i - 1$ // b may start in previous block
  find first j such that $p_j > b$ // predecessor query or scan
  return $i..(j-1)$

Note that p is a monotonically increasing sequence of integers which can be represented with different methods and trade-offs.

4.2.1 Elias-Fano Coding. A standard technique for storing monotonic sequences is Elias-Fano coding (see Section 2). A way to interpret the vector H of upper bits of an Elias-Fano coded sequence is that it stores the number of items having each possible combination of most significant bits in unary coding. To locate the predecessor of item $b = au + \ell$ in the sequence, we calculate $select_0(u-1)$ on the upper bits H, which gives us the start of a cluster of entries that all have most significant bits u. The corresponding index in L can be calculated by subtracting $(u-1)$. We scan the cluster to find the largest index i with $p_i \leq b$. In our case, this takes constant expected time (see Lemma 5.4). The internal memory usage is $m(2 + \log(a) + o(1))$ bits (see Lemma 5.1).

4.2.2 Bit Vector with Succincter. It is also possible to store p as a bit vector with rank and select support. An item p_i at position i is then represented as a 1-bit in position $i + p_i$. The position of the predecessor of a bin b can be found in constant time by calculating $select_0(b) - b$. The actual value can be calculated using a $select_1$ query. Because the bit vector is sparse, we can use Succincter [52] to compress it and its rank and select structures down to about $m(1.44 + \log(a+1) + o(1))$ bits (see Lemma 5.2).

4.2.3 Entropy Coding. We observed that in practice, the bit vector is considerably more regular than a truly random one and thus allows additional compression. This can be made fast by splitting it into ranges that are compressed individually, e.g., using dictionary compression. In our experimental evaluation in Section 7.2, we see a space-time trade-off, where we can achieve internal memory space consumption less than the theoretically best results described above in Section 4.2.2.

5 Analysis

We now formalize the properties of PaCHash in Theorem 5.1 which basically says the following: External space is just the space needed to store the variable sized objects plus possibly a few bits per block to know where the first object in the block starts. Internal space is about $2 + \log a$ bits per block where a is a tuning parameter that also shows up in a term adding $1/a$ expected I/Os to the retrieval cost.

While proving the theorem, we discuss some variants and implications. Section 5.1 considers construction cost and final space consumption, while Section 5.2 looks at I/Os and internal work of queries.

THEOREM 5.1. *Consider n objects of total size N bits which are stored in m blocks of size $\mathtt{B}$. Let $d \in 0..\log \mathtt{B}$ be an encoding-dependent number of bits needed to specify where the first bin or object of a block starts and $\bar{\mathtt{B}} = \mathtt{B} - d$ be the payload size per block, i.e., $m = N/\bar{\mathtt{B}}$. For a parameter a, let a random uniform hash function map the objects to am bins.*

Then, PaCHash with Elias-Fano coding needs $m(2 + \log a + o(1))$ bits of internal memory and $N(1 + d/\bar{\mathtt{B}})$ bits of external memory. The construction cost is the same as that of sorting the objects using am random integer keys. The expected time for retrieving an object of size $|x|$ bits is constant plus the time for scanning $1 + |x|/\bar{\mathtt{B}} + 1/a$ blocks. The unsuccessful search time is the same except that $|x|$ is replaced by 0.

5.1 Construction. Assuming that the set of input objects is stored in compressed form on external memory, we mainly need to sort the objects by their hash function value. In our model, this has complexity $sort(N)$. In most practically relevant situations, this can even be done in $O(scan(N))$ using integer sorting, see Section 5.3 for details.

The sorted representation is then scanned and basically copied to the output, only adding d bits of information within each block, which allow a query to initialize the scanning operation. What d is depends on the concrete encoding of the data, ranging from $d = 0$ for objects of identical size or for 0-terminated strings to $d = \log(\mathtt{B})$ bits when we explicitly encode the starting position of an object or bin. Refer to Section 6 for examples.

LEMMA 5.1. *When using Elias-Fano coding to store p, the index needs $2 + \log a + o(1)$ bits of internal memory per block and can be constructed in time $O(m)$.*

Proof. p consists of $k = m$ integers $\leq am = U$. Inserting this into the space usage of Elias-Fano coded sequences (see Section 2) gives us space$(p) = k(2 + \log(U/k)) + 1 =$

$m(2+\log(am/m))+1 = m(2+\log a)+1$. The *select*$_0$ data structure on the upper bits H can be stored in $o(m)$ bits [12]. Each of the m insertions into the sequence can be done in constant time while generating the external object representation. The construction of the *select*$_0$ data structure takes time $\mathrm{O}(m)$. ∎

LEMMA 5.2. *When using Succincter [52] to store p, the index needs $1.4427 + \log(a+1) + o(1)$ bits of internal memory per block.*

Proof. (Sketch, for full proof see full paper.) Using Succincter, i.e., [52, Theorem 2] with a length-$(a+1)m$ bit vector containing m ones, we can represent the internal memory index using only $\log\binom{(a+1)m}{m} + o(m) \leq m\,(1.4427 + \log(a+1)) + o(m)$ bits, which results in the space mentioned above per external memory block. ∎

The lower bound for the space usage of a minimum k-perfect hash function for objects of identical size approaches $n\cdot(\log(e)+\log(k!/k^k)/k)$ [7]. Using Stirling's approximation, we derive a new lower space bound that is easier to interpret.

$$
\begin{aligned}
&n\cdot(\log(e)+\log(k!/k^k)/k)\\
&\quad\approx n\cdot\left(\log(e)+\log\left(\frac{\sqrt{2\pi k}(k/e)^k}{k^k}\right)/k\right)\\
&\quad= n\cdot\left(\log(e)+\log(\sqrt{2\pi k}(1/e^k))/k\right)\\
&\quad= n\cdot\left(\log(e)+\frac{\log(\sqrt{2\pi k})}{k}-\frac{\log(e^k)}{k}\right)\\
&\quad= n\cdot\left(\log(e)+\frac{\log((2\pi k)^{1/2})}{k}-\log(e)\right)\\
&\quad= \frac{n}{k}\cdot\frac{1}{2}\log(2\pi k)
\end{aligned}
$$

The value n/k is the number of blocks, so MkPHFs need $\Omega(\log k)$ bits of space per block, while we show above that PaCHash needs a constant number. In a way, PaCHash therefore breaks the theoretical lower space bounds of MkPHFs while keeping the same $\mathrm{O}(1)$ query time. Choosing parameter a large can bring the number of I/O operations arbitrarily close to optimal, independently of k.

5.2 Query. We first show that a query loads a small expected number of blocks, depending only on the size of that specific object – not the other objects in the data structure. We then show that the exact blocks to be loaded can be determined upfront without any I/O operations, using constant time.

LEMMA 5.3. *Retrieving an object x of size $|x|$ from a PaCHash data structure loads $\leq 1 + |x|/\bar{\mathtt{B}} + 1/a$ consecutive blocks from the external memory in expectation (setting $|x| = 0$ if x is not in the table).*[7]

Proof. We first derive the expected number of blocks overlapped by the bin $b_x = h(x)$ that x is stored in. We then analyze the edge case that PaCHash sometimes loads one additional block unnecessarily even though it is not overlapped.

The expected size $\mathbb{E}(|b_x|)$ of b_x is the sum of $|x|$ and all other objects from the input set S that are mapped to it:

$$
\begin{aligned}
\mathbb{E}(|b_x|) &= |x| + \sum_{y\in S, y\neq x} |y|\mathbb{P}(y\in b_x)\\
&\leq |x| + \sum_{y\in S} |y|\mathbb{P}(y\in b_x) = |x| + \sum_{y\in S}|y|\cdot\frac{1}{am}\\
&= |x| + \bar{\mathtt{B}}m\cdot\frac{1}{am} = |x| + \frac{\bar{\mathtt{B}}}{a}
\end{aligned}
$$

Let X denote the number of blocks overlapped by bin b_x. Assuming that the block boundaries and bin boundaries are statistically independent,[8] and using the linearity of the expected value, we get $\mathbb{E}(X) = 1 + (\mathbb{E}(|b_x|) - 1)/\bar{\mathtt{B}} = 1 + |x|/\bar{\mathtt{B}} + 1/a - 1/\bar{\mathtt{B}}$.

At a position i, the sequence p stores the first bin b_i that intersects with block i. Most of the time, this also means that b_i extends into block $i-1$, which is why queries load that block as well. When a bin starts *exactly* at a block boundary, though, the previous block is not actually needed. Because bin boundaries are statistically independent of block boundaries, the probability of that happening is $1/\bar{\mathtt{B}}$.[9]

We get the result by putting together the expected blocks overlapped by a bin and the probability for loading one single block too much. For negative queries, we are interested in the size of the bin that x would be hashed to, so we can simply set $|x| = 0$. ∎

[7] Using fewer estimates in the proof one can derive a bound of $1 + \frac{|x|-c+1-e^{-\beta}}{\bar{\mathtt{B}}} + \frac{1}{a}$ where $\beta = \frac{n\bar{\mathtt{B}}}{Na}$ is the average number of objects per bin and c is the greatest common divisor of $\bar{\mathtt{B}}$ and all object sizes. In particular, for objects of identical size dividing $\mathtt{B}$, the bound is close to $1 + 1/a$.

[8] We can guarantee the independence by cyclically shifting the data structure, i.e., we set the offset of the first block to a random number in $0..(\bar{\mathtt{B}}-1)$ and let the last bins wrap around into the first block.

[9] When the preceding bin b_{-1} is empty, PaCHash stores that empty bin in p, as described in Section 4. This means that the probability of unnecessary block loads actually is smaller, namely $\frac{1}{\bar{\mathtt{B}}}(1 - \mathbb{P}(|b_{-1}| > 0))$, where $\mathbb{P}(|b_{-1}| > 0) = \left(1 - \frac{1}{am}\right)^n \approx e^{-\frac{n}{am}}$ is the probability of b_{-1} being empty.

LEMMA 5.4. *When using Elias-Fano coding for the index data structure of PaCHash, the range of blocks containing the bin of an object x can be found in expected constant time.*

Proof. A query for an object x consists of four steps. First, we hash x to get the corresponding bin $b_x = au + \ell$, where a is the tuning parameter of PaCHash. We then execute a constant time [12] *select*$_0$ query on the upper bits H. That gives us the start of a cluster of entries in the sequence that all have the same $\log(m)$ most significant bits u. We need to iterate over the cluster entries which are $< b_x$ until we find the predecessor. Each cluster entry corresponds to a stored bin index. Let us bound the expected size $\mathbb{E}(Y_u)$ of all bins that have most significant bits u and are $< b_x$.

$$\begin{aligned}\mathbb{E}(Y_u) &= \sum_{y \in S} |y| \cdot \mathbb{P}(h(y) \text{ has MSB} = u;\, h(y) < h(x)) \\ &\leq \sum_{y \in S} |y| \cdot \mathbb{P}(h(y) \text{ has MSB} = u) \\ &= \frac{1}{m} \sum_{y \in S} |y| = \frac{m\bar{\mathtt{B}}}{m} = \bar{\mathtt{B}}\end{aligned}$$

The expected number of cluster entries we need to scan is therefore $\mathbb{E}(Y_u)/\bar{\mathtt{B}} = 1$. The practical implementation then further scans the cluster to find the last block overlapping b_x. This takes non-constant time $\mathrm{O}(1 + |x|/\bar{\mathtt{B}})$, which is not a problem since a proportional number of blocks are loaded anyway. However, we strengthen the lemma by observing that we can also use another *select*$_0$ query followed by a backward scan of the cluster. □

5.3 Details on External Sorting. We now show that the external sorting needed during construction of a PaCHash data structure can be done in scanning complexity using very modest additional assumptions. First note that the problem of sorting objects during construction is easy when the average object size exceeds the block size, i.e., $N/n > \mathtt{B}$ and thus $n < N/\mathtt{B}$. In that case, a variant of bucket sort that maps the keys to $\mathrm{O}(n)$ buckets runs with linear internal expected work and $\mathrm{O}(n + N/\mathtt{B}) = \mathrm{O}(N/\mathtt{B})$ I/Os [55, Theorem 5.9].

On the other hand, the average object size N/n must be at least $\log n$ since we are looking at objects with unique keys. For the remaining case $\log n \leq N/n \leq \mathtt{B}$, we additionally make a *tall cache assumption* quite usual for external memory [27] where $\mathtt{M} > \mathtt{B}^2$. Since the index data structure has at least $N/\mathtt{B}$ bits, we also know that $\mathtt{M} \geq N/\mathtt{B}$. A single scan of the input can partition it into pieces of size about $\frac{N}{\mathtt{M}/\mathtt{B}} \leq \frac{N}{(N/\mathtt{B})/\mathtt{B}} = \mathtt{B}^2 \leq \mathtt{M}$ which fit into internal memory. Moreover, since the average object size is $\geq \log n$, we can afford to replace the objects in an internally sorted fragment of the input by key-pointer pairs which once more allows us to use bucket sort – this time running in internal memory.

Table 3: External space overhead of d bits per block in order to facilitate scanning that block. The term $+1$ when $d \neq 0$ is needed for the case that no object starts in a block.

d	Case Description
0	Identical object sizes, zero terminated strings and analogous cases
$\lceil \log(w+1) \rceil$	Objects that use variable bit-length encoding with $\leq w \leq \mathtt{B}$ bits
$\lceil \log(W/w+1) \rceil$	Objects of size divisible by w with $W = \min(\mathtt{B}, \max \text{ object size})$
$\lceil \log(\mathtt{B}) \rceil$	Explicit storage of a starting position of a bin

6 Variants and Refinements

Up until now, PaCHash was described as a static, external hash table for objects of variable size. The following section describes variants of the scheme.

Object Encoding. Instead of storing objects contiguously with a self-delimiting encoding, PaCHash allows for a wide range of other options, as shown in Table 3. In general, we have a trade-off between the space needed to decode the objects in a block and the strength of assumptions made on object representation. For example, explicitly storing the offsets of objects in blocks removes the restriction to a self-delimiting encoding, without increasing the size of the internal data structure. Another important case are objects of identical size where we can calculate the block offset at query time and therefore need no external space overhead. When the object size divides the block size, it can be shown that the expected number of I/O operations is close to $1 + 1/a$.

Memory Locations. PaCHash can be stored fully externally. By doing so, the number of I/Os for a query is increased by three (two I/Os to query the rank and select data structure on the bit vector of the Elias-Fano coding and one I/O to get the remaining bits). The number of I/Os can be reduced by interleaving the arrays of the Elias-Fano coding. PaCHash is also interesting as a purely internal data structure since it allows for configurations that need less space than any previous approach, even for objects of identical size. A variant

that simplifies the external memory representation is to store the d bits of offsets per block in an internal memory data structure, possibly interleaved with the Elias-Fano representation. A variant enabling faster scanning of blocks separates keys and values [43], for example by storing $\log \mathtt{B}$ bits of offset for each object.

Functional Enhancements. Because PaCHash sorts objects by their hashed key, *range queries* with respect to the original keys are not immediately possible. Litwin and Lomet [42] implement range queries for hash tables by partitioning the key space into smaller pieces. An index tree then leads to a number of small (PaCHash) tables that are fully scanned. Order-preserving hash functions [28] are another alternative. PaCHash can be made *dynamic* using standard techniques like a Log-Structured Merge Tree [49, 44]. Merging multiple PaCHash data structures is possible efficiently. The idea is to construct the hash function h by first hashing to a larger range and then mapping it linearly to the range am. When updating h to the new total number of blocks, the objects of both input data structures are already sorted and can be merged with a linear sweep.

PaCHash as Variable-Bit-Length Array. Since one of PaCHash's key features is to store objects of variable size efficiently, it can also be used as variable-bit-length array. To this end, we simply use the array index as hash function if we also store the number of previously stored objects. However, we then have to assume that objects stored in the PaCHash VLA are self-delimiting, as this allows us to identify the objects within a block. Note that this assumption is satisfied in a lot of applications VLAs are used in, e.g., when storing variable length codes like Elias-γ and -δ codes [19] or Golomb codes [30]. Alternatively, in external memory, we can lift the restriction to self-delimiting objects by storing offsets as described above. The number of previously stored objects is necessary to identify the element within the block, and requires at most $\lceil \log n \rceil$ bits per external memory block.

7 Experiments

The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License: https://github.com/ByteHamster/PaCHash. The code for the comparison with competitors (including our competitors' code with some patches) is available on GitHub as well: https://github.com/ByteHamster/PaCHash-Experiments. The latter repository also contains a Docker image that can build and run a simplified version of the experiments from Figure 2 and Figure 4 in about 30 minutes.

Experimental Setup. We run our experiments on an Intel i7 11700 processor with 8 cores and a base clock speed of 2.5 GHz. We use a Samsung 980 Pro NVMe SSD with a capacity of 1 TB. The machine runs Ubuntu 21.10 with Linux 5.13.0. We use the GNU C++ compiler version 11.2.0 with optimization flags `-O3 -march=native`. Externally, each block of size $\mathtt{B} = 2^{15}$ bits (4096 bytes) stores a table of 8 byte keys and 2 byte object offsets. During construction, we sort pointers to the objects using IPS^{2}Ra [4]. Unless otherwise specified, the index is an Elias-Fano coded sequence based on sdsl's [29] arrays of flexible bit width and the select data structures by Kurpicz [37]. For the I/O operations, we use `io_uring`. Query operations keep a queue of 128 asynchronous requests in flight.

Competitors. To our knowledge, there is no existing implementation of a hash table for variable size objects that is simultaneously aimed at low internal memory usage and few I/O operations. As the main competitors, we choose LevelDB [32], RocksDB [21], and SILT [41]. To abstract from the different implementations of I/O operations, we also extract the internal memory index (address calculation) from some competitors. Additionally, we compare PaCHash to std::unordered_map, as well as the perfect hash functions RecSplit [20], CHD [7, 16], and PTHash [53]. Despite std::unordered_map not being tuned for efficiency, it is a widely available, general purpose hash table that can be seen as baseline for the simple idea of explicitly storing pointers instead of building a compressed index data structure.[10]

We also implement Separator Hashing [31, 39] and Cuckoo Hashing [5, 50]. In contrast to the original papers, our implementations can be used with objects of variable size $\leq \mathtt{B}$ when setting the load factor low enough. Note that decreasing the load factor increases the number of blocks and therefore the space needed for indexing. The construction of PaCHash always succeeds, while it can fail for Separator and Cuckoo Hashing depending on the preselected load factor or tuning parameter. Refer to Figure 5 for details.

7.1 PaCHash Configurations.

The parameter a provides a trade-off between internal space usage and query performance, see Table 4. Figure 2 plots the bytes read per query, depending on the average object size and parameter a. It confirms the results of our theoretical analysis in practice. The throughput of the Elias-Fano representation increases when parameter a gets larger

[10] In this setting, general purpose internal memory hash tables do not work well, as they introduce an overhead of at least $\log m$ bits per element to store the positions, and they also have to store the length of the element.

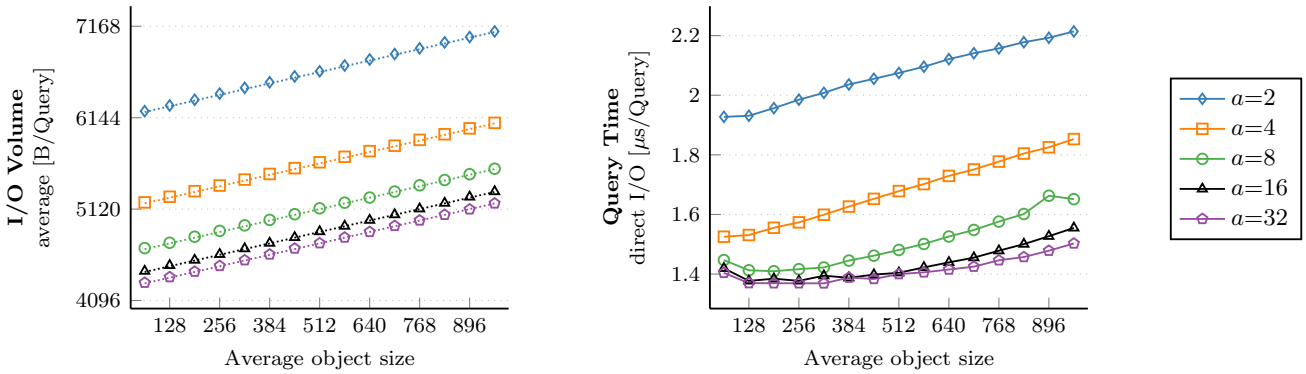


Figure 2: Dependence of I/O volume and query time on the average object size s. Sizes are normal distributed with variance $s/5$, rounded to the next positive integer. Dotted lines show theoretic I/O volumes, while marks show measurements. Note that the measurements closely match the analysis. Using other distributions and plotting over the returned objects' sizes gives equivalent results.

Table 4: Average internal space usage and average query time for different values of the parameter a and normal distributed object sizes. For more information on the query time, which is influenced by the object size, see Figure 2. Note that the internal space usage does not depend on the object size.

a	avg. internal space [B/block]	avg. query time [μs/query]
2	3.01	2.07
4	4.01	1.68
8	5.01	1.50
16	6.01	1.43
32	7.01	1.41

because the SSD needs to load fewer blocks. We also see that (at least for larger a) query times grow more slowly with object size than the I/O volume. We choose $a = 8$ for the comparison with competitors because it achieves a good balance between space usage (≈ 5 bits/block) and throughput ($\approx$ 700k Queries/second).

7.2 PaCHash with Real World Data Sets. Figure 3c compares throughput and space usage of PaCHash using real world size distributions and different index data structures. The Twitter data set contains tweets from 01.08.–05.08.2021 and has only small objects. The UniRef 50 protein database [56] contains some objects larger than the block size and the LZ4 compressed [14] English Wikipedia from November 2021 contains significantly larger objects. See Figures 3a and 3b for details.

The entropy coded bit vector saves up to one bit of internal memory per block for small a. While it comes with a performance penalty caused by decompression (up to eight times slower than Elias-Fano), it is fast enough that it can be useful for some applications. Succincter provides space usage lower than Elias-Fano but has no implementation. Note that for $a \leq 16$, the entropy coded bit vector requires even less space than succincter. Only for $a \geq 64$ it requires more space than Elias-Fano.

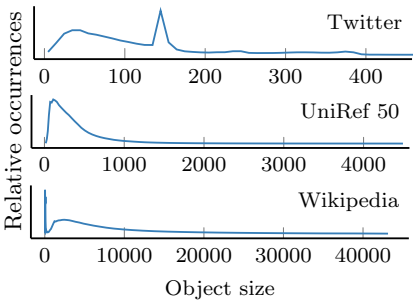
7.3 Comparison with Competitors. Figure 4 compares PaCHash to other hash table data structures – see Table 5 for the exact configurations used. These plots include measurements for identical size objects in order to allow for a large set of competitors and measurements for variable size objects containing fewer data points due to the lack of support for variable size objects by most competitors.

Perhaps the closest contender to PaCHash is the Separator method where our implementation partially allows variable object size. It needs comparable internal space and has faster queries (always a single block access). However, Separator not only has slower construction, but it also cannot achieve a load factor close to 100% except for objects with identical size when the block size is divisible by the object size. Figure 5 gives details showing load factors between 85% and 95% in typical cases.

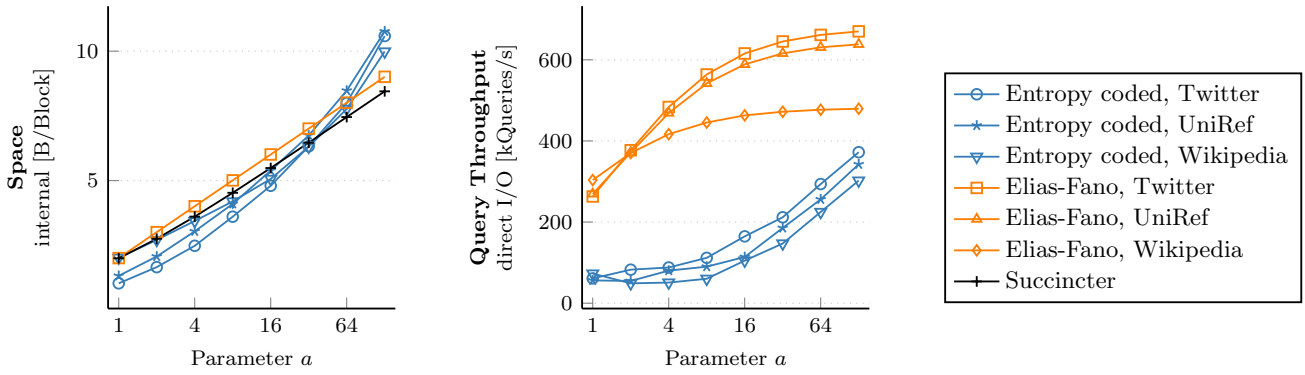
The perfect hashing methods CHD and RecSplit have similar problems with respect to variable size objects and are more expensive with respect to internal space and construction costs. While PTHash offers fast construction and queries, it does not support variable size objects and needs more internal space. Cuckoo hashing needs no internal space but has more expensive queries and the same problem with high load factors as Separator or perfect hashing.

	Twitter	UniRef 50	Wikipedia
Objects n	20 238 968	48 531 431	16 181 427
Average size	115 B	281 B	1731 B
Median size	94 B	194 B	77 B
Maximum size	560 B	45 KB	272 KB
Total size N	2.4 GB	13.2 GB	26.3 GB
Objects $>$ `B`	0%	0.08%	12%

(a) Twitter, UniRef, and Wikipedia real world data sets we use for benchmarks. The median of 77 bytes of the Wikipedia data set is caused by pages that are redirects.



(b) Relative occurrences of object sizes in the real world data sets described in Figure 3a.



(c) PaCHash with real world data sets using different index data structures. There is no practical implementation of Succincter [52], so we only give calculated values and no throughput. The space usage of Elias-Fano and Succincter is independent of the object size distribution, so we plot it only for one data set.

Figure 3: Space and query throughput of PaCHash with real world data sets.

The object stores LevelDB, RocksDB, and SILT have much larger internal space requirements *and* some external overhead. In part this comparison is unfair since they have additional functionality like dynamic operation. For SILT and LevelDB we have been able to extract the static part but still get considerably more space and lower performance than PaCHash. Figure 4 contains measurements for both the full competitors and their static parts, so the overhead originating from dynamic operation can be read off it. Comparing query throughput is complicated because of different file access modes, internal caching, and history dependent performance for the actual SSD accesses (the controller uses caching and rearranges data outside the control of the user). We have therefore looked at two different access methods and also at only the index data structure. However, overall, we get a consistent picture with Separator being the fastest method followed by PaCHash. A comparison with the vanilla internal hash table std::unordered_map is also instructive. We naturally get faster construction and high internal space consumption. Surprisingly, access to the internal data structure is only faster than PaCHash for very small inputs that fit into cache.

While not as surprisingly, it should be noted that all object stores supporting variable size objects do not show any difference with respect to (internal and/or external) space requirements, construction and query throughput when storing variable size objects compared to identical size objects. Thus, all benefits of PaCHash described above hold true for variable size objects as well.

8 Conclusion and Future Work

With PaCHash, we present a static hash table that can space-efficiently store variable size (possibly compressed) objects. The objects are stored contiguously without the usual need for empty space to equalize the nonuniformity in assignment by a hash function. This is facilitated by an index data structure that needs only a constant number of internal memory bits per external memory block. In constant expected time, it yields a near-optimal range of blocks that contain the sought object. Our implementation of PaCHash considerably outperforms previous object stores for variable size objects and even matches or outperforms systems that are purely internal memory or only handle objects of identical size.

Future work might include integrating PaCHash

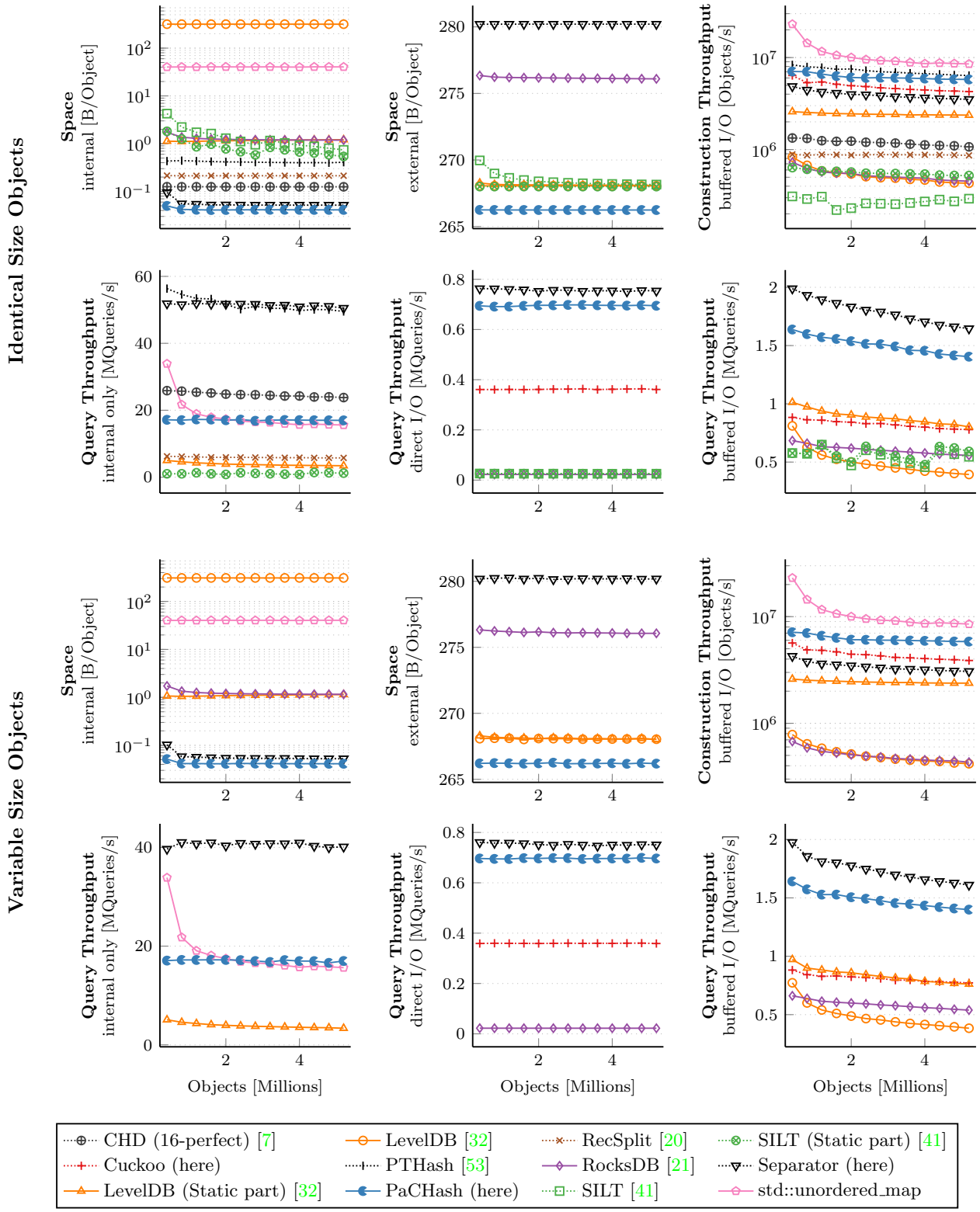


Figure 4: Comparison of object stores using objects of identical size 256 bytes (top) and uniform random size $\in [128, 384]$ bytes (bottom). Keys are 8 byte random strings. Dotted lines indicate methods supporting only objects of identical size natively. We enhanced two of them to partially support variable size objects (see Section 7).

Table 5: Configurations of competitors

Competitor	Configuration parameters
CHD [7]	Load factor 0.98. $k = 16$ collisions. Bin size 12.
Cuckoo (here, based on [5, 50])	2 alternative positions for each object, loaded in parallel to reduce latency. Streamed queries with *await any*. Load factor 0.95. Random walk insertion.
LevelDB [32]	No compression. Construction using a single, large write batch. No Bloom filters.
PaCHash (here)	$a = 8$. External blocks store a table of keys and offsets. Streamed queries with *await any*.
PTHash [53]	"Optimizing the general trade-off" [53] with $\alpha = 0.94, c = 7$, D-D Encoding.
RecSplit [20]	Leaf size $\ell = 8$. Bucket size $b = 2000$.
RocksDB [21]	Block cache disabled. No memory mapping or WAL. Queries use batches of size 64. No Bloom filters.
Separator (here, based on [31, 39])	6 bit separators. Load factor 0.96. Streamed queries with *await any*.
SILT [41]	`testCombi.xml` configuration from original repository.
std::unordered_map	8 byte keys. 64 bit pointers to object contents.

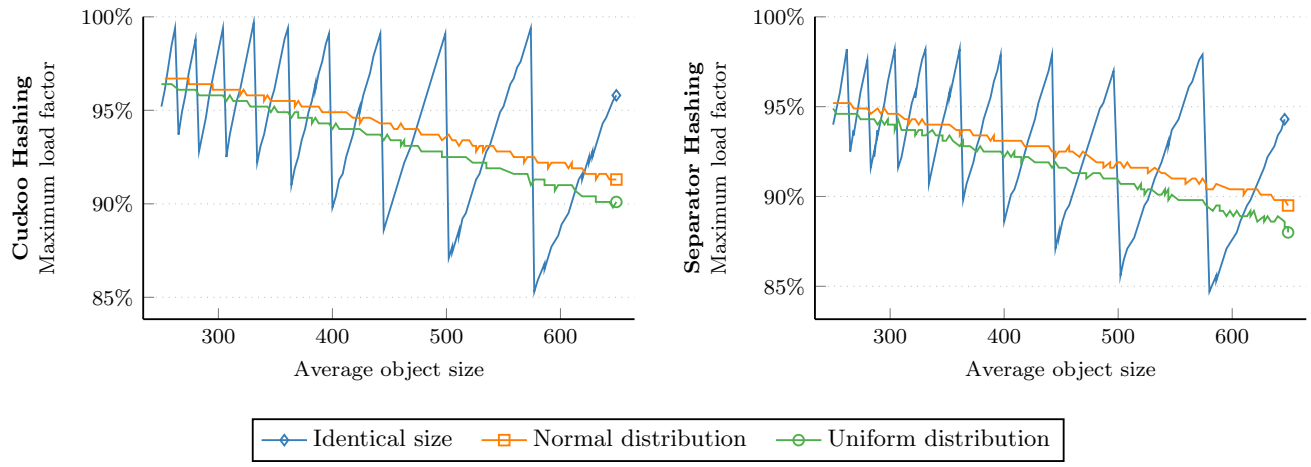


Figure 5: Maximum achievable load factor with different distributions of object sizes of our implementations of Separator Hashing and Cuckoo Hashing that support variable size objects. For an average object size s, the normal distribution has a variance of $s/5$ and the uniform random sizes are drawn from $[0.25s, 1.75s]$

into dynamic external memory object stores, as well as engineering fast and space efficient internal memory variants. On the theoretical side, we would like to better understand the space requirements and lower bounds of bit vectors with entropy coding. This includes relations to different variants of perfect hashing. Although our current analysis assumes random hash functions, PaCHash may also be provably efficient for more realistic simple hash functions. Further possible space-saving can use the quotienting idea [35, 8, 3, 13] where some bits of the stored keys are derived from the (now invertible) hash function value. It is interesting how this works best in the presence of nonuniformly distributed keys.

Acknowledgements. The authors would like to thank Peter Dillinger and Stefan Walzer for early discussions leading to this paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

References

[1] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *NSDI*, pages 337–350. USENIX Association, 2015.

[2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: a fast array of wimpy nodes. In *SOSP*, pages 1–14. ACM, 2009. doi:10.1145/1629575.1629577.

[3] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *FOCS*, pages 787–796. IEEE Computer Society, 2010. doi:10.1109/FOCS.2010.80.

[4] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Trans. Parallel Comput.*, 9(1):2:1–2:62, 2022. doi:10.1145/3505286.

[5] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations (extended abstract). In *STOC*, pages 593–602. ACM, 1994. doi:10.1145/195058.195412.

[6] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, pages 47–60. USENIX Association, 2010.

[7] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *ESA*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer, 2009. doi:10.1007/978-3-642-04128-0_61.

[8] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. All-purpose hashing. *CoRR*, abs/2109.04548, 2021.

[9] Daniel K. Blandford and Guy E. Blelloch. Compact dictionaries for variable-length keys and data with applications. *ACM Trans. Algorithms*, 4(2):17:1–17:25, 2008. doi:10.1145/1361192.1361194.

[10] Pedro Celia. External robin hood hashing. Technical report, Computer Science Department, Indiana University. TR246, 1988.

[11] Xubin Chen, Ning Zheng, Shukun Xu, Yifan Qiao, Yang Liu, Jiangpeng Li, and Tong Zhang. Kallaxdb: A tableless hash-based key-value store on storage hardware with built-in transparent compression. In *DaMoN*, pages 3:1–3:10. ACM, 2021. doi:10.1145/3465998.3466004.

[12] David Clark. *Compact PAT trees.* PhD thesis, University of Waterloo, 1997. URL: http://hdl.handle.net/10012/64.

[13] John G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers*, 33(9):828–834, 1984. doi:10.1109/TC.1984.1676499.

[14] Yann Collet. LZ4: Extremely fast compression algorithm. https://github.com/lz4/lz4.

[15] Douglas Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979. doi:10.1145/356770.356776.

[16] Davi de Castro Reis, Djamel Belazzougui, Fabiano Cupertino Botelho, and Nivio Ziviani. CMPH - C minimal perfect hashing library. http://cmph.sourceforge.net/, 2012.

[17] Biplob K. Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: RAM space skimpy key-value store on flash-based storage. In *SIGMOD Conference*, pages 25–36. ACM, 2011. doi:10.1145/1989323.1989327.

[18] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theor. Comput. Sci.*, 380(1-2):47–68, 2007. doi:10.1016/j.tcs.2007.02.054.

[19] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974. doi:10.1145/321812.321820.

[20] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In *ALENEX*, pages 175–185. SIAM, 2020. doi:10.1137/1.9781611976007.14.

[21] Facebook. RocksDB. a persistent key-value store for fast storage environments. https://rocksdb.org, 2021.

[22] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - A fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979. doi:10.1145/320083.320092.

[23] Robert Mario Fano. On the number of bits required to implement an associative memory. Technical report, MIT, Computer Structures Group, 1971. Project MAC, Memorandum 61".

[24] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.

[25] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

[26] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005. doi:10.1007/s00224-004-1195-x.

[27] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814600.

[28] Anil K. Garg and C. C. Gotlieb. Order-preserving key transformations. *ACM Trans. Database Syst.*, 11(2):213–234, 1986. doi:10.1145/5922.5923.

[29] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.

[30] Solomon W. Golomb. Run-length encodings. *IEEE Trans. Inf. Theory*, 12(3):399–401, 1966. doi:10.1109/TIT.1966.1053907.

[31] Gaston H. Gonnet and Per-Åke Larson. External hashing with limited internal storage. *J. ACM*,

35(1):161–184, 1988. doi:10.1145/42267.42274.

[32] Google. LevelDB is a fast key-value storage library written at google. https://github.com/google/leveldb, 2021.

[33] Morten Skaarup Jensen and Rasmus Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008. doi:10.1007/s00453-007-9155-x.

[34] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[35] Dominik Köppl, Simon J. Puglisi, and Rajeev Raman. Fast and simple compact hashing via bucketing. *Algorithmica*, pages 1–32, 2022. doi:https://doi.org/10.1007/s00453-022-00996-y.

[36] M. Oguzhan Külekci. Enhanced variable-length codes: Improved compression with efficient random access. In *DCC*, pages 362–371. IEEE, 2014. doi:10.1109/DCC.2014.74.

[37] Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE*, volume 13617 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2022. doi:10.1007/978-3-031-20643-6_19.

[38] Per-Åke Larson. Linear hashing with separators - A dynamic hashing scheme achieving one-access retrieval. *ACM Trans. Database Syst.*, 13(3):366–388, 1988. doi:10.1145/44498.44500.

[39] Per-Åke Larson and Ajay Kajla. File organization: Implementation of a method guaranteeing retrieval in one access. *Commun. ACM*, 27(7):670–677, 1984. doi:10.1145/358105.358193.

[40] Per-Åke Larson and M. V. Ramakrishna. External perfect hashing. In *SIGMOD Conference*, pages 190–200. ACM Press, 1985. doi:10.1145/318898.318916.

[41] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *SOSP*, pages 1–13. ACM, 2011. doi:10.1145/2043556.2043558.

[42] Witold Litwin and David B. Lomet. The bounded disorder access method. In *ICDE*, pages 38–48. IEEE Computer Society, 1986. doi:10.1109/ICDE.1986.7266204.

[43] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage*, 13(1):5:1–5:28, 2017. doi:10.1145/3033273.

[44] Chen Luo and Michael J. Carey. LSM-based storage techniques: a survey. *VLDB J.*, 29(1):393–418, 2020. doi:10.1007/s00778-019-00555-y.

[45] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A scalable, lightweight, ftl-aware key-value store. In *USENIX Annual Technical Conference*, pages 207–219. USENIX Association, 2015.

[46] Gonzalo Navarro. *Compact Data Structures – A Practical Approach*. Cambridge University Press, 2016.

[47] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *NSDI*, pages 385–398. USENIX Association, 2013.

[48] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191. USENIX, 1999.

[49] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996. doi:10.1007/s002360050048.

[50] Rasmus Pagh. Basic external memory data structures. In *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 14–35. Springer, 2003. doi:10.1007/3-540-36574-5_2.

[51] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. doi:10.1016/j.jalgor.2003.12.002.

[52] Mihai Patrascu. Succincter. In *FOCS*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

[53] Giulio Ermanno Pibiri and Roberto Trani. Pthash: Revisiting FCH minimal perfect hashing. In *SIGIR*, pages 1339–1348. ACM, 2021. doi:10.1145/3404835.3462849.

[54] M. V. Ramakrishna and Walid R. Tout. Dynamic external hashing with guaranteed single access retrieval. In *FODO*, volume 367 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 1989. doi:10.1007/3-540-51295-0_127.

[55] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. doi:10.1007/978-3-030-25209-0.

[56] Baris E. Suzek, Hongzhan Huang, Peter B. McGarvey, Raja Mazumder, and Cathy H. Wu. Uniref: comprehensive and non-redundant uniprot reference clusters. *Bioinform.*, 23(10):1282–1288, 2007. doi:10.1093/bioinformatics/btm098.

[57] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: two-level memories. *Algorithmica*, 12(2/3):110–147, 1994. doi:10.1007/BF01185207.

[58] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *Proc. VLDB Endow.*, 11(11):1522–1535, 2018. doi:10.14778/3236187.3236203.