Separate Chaining Meets Compact Hashing

Dominik Köppl[1,a)]

Abstract: While separate chaining is a common strategy for resolving collisions in a hash table taught in most textbooks, compact hashing is a less common technique for saving space when hashing integers whose domain is relatively small with respect to the problem size. It is widely believed that hash tables waste a considerable amount of memory, as they either leave allocated space untouched (open addressing) or store additional pointers (separate chaining). For the former, Cleary introduced the compact hashing technique that stores only a part of a key to save space. However, as can be seen by the line of research focusing on compact hash tables with open addressing, there is additional information, called displacement, required for restoring a key. There are several representations of this displacement information with different space and time trade-offs. In this article, we introduce a separate chaining hash table that applies the compact hashing technique without the need for the displacement information. Practical evaluations reveal that insertions in this hash table are faster or use less space than all previously known compact hash tables on modern computer architectures when storing sufficiently large satellite data.

1. Introduction

A major layout decision for hash tables is how collisions are resolved. A well-studied and easy-implementable layout is separate chaining, which is also applied by the hash table `unordered_map` of the C++ standard library `libstdc++` [4, Sect. 22.1.2.1.2]. On the downside, it is often criticized for being bloated and slow[*1]. In fact, almost all modern replacements feature open addressing layouts. Their implementations are highlighted with detailed benchmarks putting separate chaining with `unordered_map` as its major representation in the backlight of interest. However, when considering compact hashing with satellite data, separate chaining becomes again a competitive approach, on which we shed a light in this article.

1.1 Related Work

The hash table of Askitis [2] also resorts to separate chaining. Its buckets are represented as dynamic arrays. On inserting an element into one of these array buckets, the size of the respective array increments by one (instead of, e.g., doubling its space). The approach differs from ours in that these arrays store a list of (key,value)-pairs while our buckets separate keys from values.

The scan of the buckets in a separate chaining hash table can be accelerated with SIMD (single instruction multiple data) instructions as shown by Ross [20] who studied the application of SIMD instructions for comparing multiple keys in parallel in a *bucketized Cuckoo hash table*.

For reducing the memory requirement a of hash table, a *sparse* hash table layout was introduced by members of Google[*2]. Sparse hash tables are a lightweight alternative to standard open addressing hash tables, which are represented as plain arrays. Most of the sparse variants replace the plain array with a bit vector of the same length marking positions at which an element would be stored in the array. The array is emulated by this bit vector and its partitioning into buckets, which are dynamically resizeable and store the actual data.

The notion of *compact hashing* was coined by Cleary [5] who studied a hash table with bidirectional linear probing. The idea of compact hashing is to use an injective function mapping keys to pairs of integers. Using one integer, called *remainder*, as a hash value, and the other, called *quotient*, as the data stored in the hash table, the hash table can restore a key by maintaining its quotient and an information to retain its corresponding remainder. This information, called *displacement*, is crucial as the bidirectional linear probing displaces elements on a hash collision from the position corresponding to its hash value, i.e., its remainder. Poyias and Raman [19] gave different representations for the displacement in the case that the hash table applies linear probing.

In this paper, we show that it is not necessary to store additional data in case that we resort to separate chaining

[1] Department of Informatics, Kyushu University, Japan Society for Promotion of Science
[a)] dominik.koeppl@inf.kyushu-u.ac.jp

[*1] Cf. `http://www.idryman.org/blog/2017/05/03/writing-a-damn-fast-hash-table-with-tiny-memory-footprints/`, `https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/`, `https://attractivechaos.wordpress.com/2018/10/01/advanced-techniques-to-implement-fast-hash-tables`, `https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html`, to name a few.

[*2] `https://github.com/sparsehash/sparsehash`

© 1992 Information Processing Society of Japan

as collision resolution. The main strength of our hash table is its memory-efficiency during the construction while being at least as fast as other compact hash tables. Its main weakness is the slow lookup time for keys, as we do not strive for small bucket sizes.

2. Separate Chaining with Compact Hashing

Our hash table H has $|H|$ buckets, where $|H|$ is a power of two. Let h be the hash function of H. An element with key K is stored in the $(h(K) \bmod |H|)$-th bucket. To look up an element with key K, the $(h(K) \bmod |H|)$-th bucket is linearly scanned.

A common implementation represents a bucket with a linked list, and tries to avoid collisions as they are a major cause for decelerating searches. Here, the buckets are realized as dynamic arrays, similar to the array hash table of Askitis [2]. We further drop the idea of avoiding collisions. Instead, we want to maintain buckets of sufficiently large sizes to compensate the extra memory for maintaining (a) the pointers to the buckets and (b) their sizes. To prevent a bucket from growing too large, we introduce a threshold $b_{\max}$ for the maximum size. Choosing an adequate value for $b_{\max}$ is important, as it affects the resizing and the search time of our hash table.

Resize. When we try to insert an element into a bucket of maximum size $b_{\max}$, we create a new hash table with twice the number of buckets $2|H|$ and move the elements from the old table to the new one, bucket by bucket. After a bucket of the old table becomes empty, we can free up its memory. This reduces the memory peak commonly seen in hash tables or dynamic vectors reserving one large array, as these data structures need to reserve space for $3m$ elements when resizing from m to $2m$. This technique is also common for sparse hash tables.

Search in Cache Lines. We can exploit modern computer architectures featuring large cache sizes by selecting a sufficiently small $b_{\max}$ such that buckets fit into a cache line. Since we are only interested in the keys of a bucket during a lookup, an optimization is to store keys and values separately: In our hash table, a bucket is a composition of a key bucket and a value bucket, each of the same size. This gives a good locality of reference [7] for searching a key. This layout is favorable for large values of $b_{\max}$ and (keys,value)-pairs where the key size is relatively small to the value size, since (a) the cost for an extra pointer to maintain two buckets instead of one becomes negligible while (b) more keys fit into a cache line when searching a key in a bucket. An overview of the proposed hash table layout is given in Fig. 1.

2.1 Compact Hashing

Compact hashing restricts the choice of the hash function h. It requires an injective transform f that maps a key K to two integers (q, r) with $1 \le r \le |H|$, where r acts as the hash value $h(K)$. The values q and r are called *quotient* and *remainder*, respectively. The quotient q can be used to restore the original key K if we know its corresponding remainder r. We translate this technique to our separate chaining layout by storing q as key in the r-th bucket on inserting a key K with $f(K) = (q, r)$.

A discussion of different injective transforms is given by Fischer and Köppl [11, Sect. 3.2]. Suppose that all keys can be represented by k bits. We want to construct a bijective function $f : [1..2^k] \rightarrow f([1..2^k])$, where we use the last $\lg m$ bits for the remainder and the other bits for the quotient. Our used transform[*3] is inspired by the split-mix algorithm [21]. It intermingles three xorshift [16] functions $f_j^{\otimes} : x \mapsto x \otimes (2^j x) \mod 2^k$ with three multiplicative functions $f_c^{\times} : x \mapsto cx \mod 2^k$, where $\otimes$ denotes the bit-wise exclusive OR operation. The composition of these functions is invertible, since each of them itself is invertible:

Xorshift. The function $f_j^{\otimes}$ is self-inverse, i.e., $f_j^{\otimes} \circ f_j^{\otimes}(K) = K$, for an integer j with $k \ge j > \lfloor k/2 \rfloor$ or $-k \le j < -\lfloor k/2 \rfloor$. For datasets whose keys only slightly differ (i.e., incremental values), selecting $j < -\lfloor k/2 \rfloor$ instead of $j > \lfloor k/2 \rfloor$ is more advantageous since the former distributes the last $k + j$ bits affecting the remainder. This can lead to a more uniform distribution of the occupation of the buckets.

Multiplicative. Each of our functions $f_c^{\times}$ is initialized with an odd number c less than 2^k. It is known that the family $\{f_c^{\times}\}_{c \text{ odd}}$ is universal [8, Sect. 2.2], but not strongly universal [23]. Since c and 2^k are relatively prime, there is an modular multiplicative inverse of c with respect to the divisor 2^k, which we can find with the extended Euclidean algorithm in $\mathcal{O}(k)$ time in a precomputation step [13, Sect. 4.5.2].

2.2 Resize Policies

We resize a bucket with the C function `realloc`. Whether we need to resize a bucket on inserting an element depends on the policy we follow:

Incremental Policy : Increment the size of the bucket such that the new element just fits in. This policy saves memory as only the minimum required amount of memory is allocated. As buckets store at most $b_{\max} = \mathcal{O}(1)$ elements, the resize can be performed in constant time. In practice, however, much of the spent time depends on the used memory allocator for increasing the allocated space. We append '$^{++}$' to a hash table in subscript if it applies this policy.

Half Increase : Increase the size of a bucket by 50%[*4]. This policy eases the burden of the allocator at the expense of possibly wasting memory for unoccupied space in the buckets. We append '50' in subscript to a hash table if it applies this policy.

2.3 Bucket Variations

Our hash table layout in Fig. 1 supports different quotient and value bucket types. In the experiments, we call a

[*3] https://github.com/kampersanda/poplar-trie

[*4] Inspired by the discussion in https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md.

© 1992 Information Processing Society of Japan

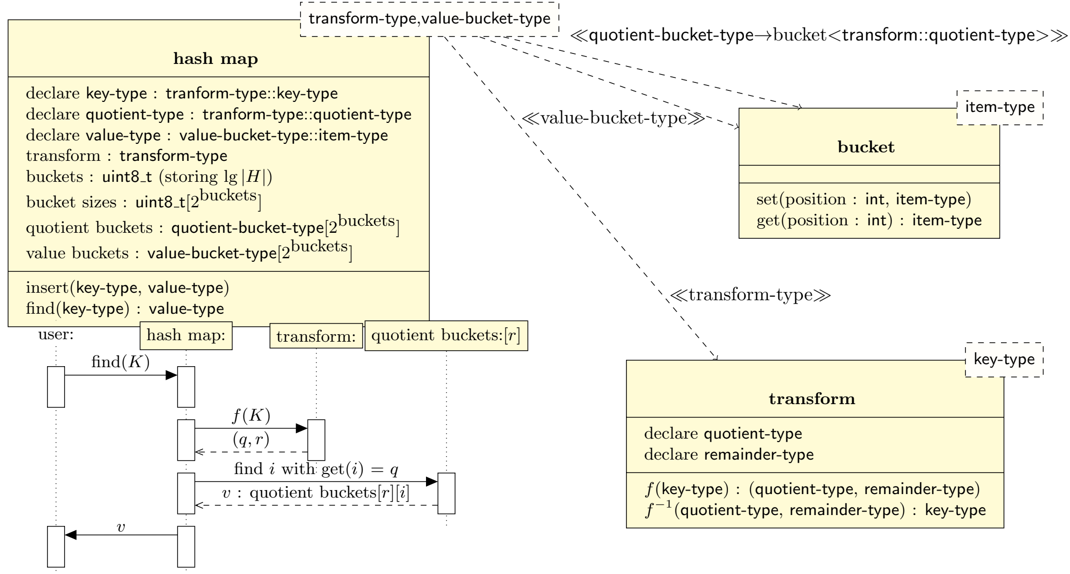

Fig. 1 Diagram of our proposed hash table. The call of find(K) returns the i-th element of the r-th bucket at which K is located. The injective transform determines the types of the key and the quotient.

hash table by the name of its quotient bucket representation. There, we evaluated the following representations:

cht. Our default quotient bucket stores quotients bit-compactly, i.e., it stores a quotient in $k - \lg |H|$ bits[*5], where k is the bit size needed to represent all keys and $|H|$ is the number of buckets of H. For that, it uses bit operations to store the quotients in a byte array. The number of bits used by a key bucket is quantized at eight bits (the last byte of the array might not be full). Since $\lg |H|$ is constant until a rehashing occurs, we do not have to maintain this value in each bucket.

single. A variant storing keys and values in a single bucket instead of two separate ones can save additional space. However, this space improvement is marginal compared to the more severe slowdown for either (a) locating a quotient if we maintain a list of (key,value)-pairs or (b) changing the size of the bucket if we first store all keys and then subsequently all values. For the experiments, we used the representation (b).

avx. Another representation of the quotient bucket applies SIMD instructions to speed up the search of a key in a large bucket. For that, it restricts the quotients to be quantized at 8 bits. We use the AVX2 instructions `_mm256_set1_epi`k and `_mm256_cmpeq_epi`k for loading a quotient with k bits and comparing this loaded value with the entries of the bucket, respectively. The `realloc` function for resizing a bucket cannot be used in conjunction with avx, since the allocated memory for avx must be 32-byte aligned.

plain. For comparison, we also implemented a variant that does not apply compact hashing. For that, we created the trivial injective transform $f_h : K \mapsto (K, h(K) \bmod 2^m)$ that uses an arbitrary hash function h for computing the remainders while producing quotients equal to the original keys. Its inverse is trivially given by $f^{-1}(q, r) = q$.

*5 More precisely, the quotient needs $\lceil \lg |f| \rceil - \lg |H|$ bits, where $\lceil \lg |f| \rceil$ is the number of bits needed to represent all values of the transform f, which is k in our case.

2.4 Details on Sizes

We set the maximum bucket size $b_{\max}$ to 255 elements such that we can represent the size of a bucket in a single byte. A full bucket with 64-bit integers takes roughly 2KiB of memory, fitting in the L1 cache of a modern CPU.

For each bucket we store its size and pointers to its quotient and value bucket, using altogether 17 bytes. Since we additionally waste less than one byte in cht for storing the quotients in a byte array, this gives an overhead of at most 18 bytes per bucket. Let m denote the (fractional) number of bytes needed to store an element. Then our hash table uses $18|H| + nm$ bytes for storing n elements, where $|H|$ is at most $\lceil 2n/b_{\max} \rceil$ if we assume a uniform distribution of the elements among all buckets.

A non-sparse open addressing hash table with maximum load factor $\alpha \leq 1$ uses at least nm/α bytes. If $m \geq 3.03$ bytes, we need to set α to more than 0.956 to make the open addressing hash table slimmer than our separate hashing table. When resorting to linear probing we encounter

$$c_\alpha := (1/2) \cdot (1 + (1/(1-\alpha))^2) \quad (1)$$

collisions on average for an insertion operation [14, Sect. 6.4]. But $c_\alpha < b_{\max} \Leftrightarrow \alpha < 1 - 1/\sqrt{2b_{\max} - 1} \approx 0.956$, and hence such a table faces more collisions on average or uses more space than our proposed hash tables. If $m < 3.03$ represents the number of bytes for storing a key and a value, one would usually resort to storing the data in a plain array, as there can be at most $3^{8m} \approx 20 \cdot 10^6$ keys. The only interesting domain is when we consider compact hashing for $m < 3.03$, where m now represents the number of bytes we need for the *quotient* and the value. However, compact representations of open addressing hash tables need to store displacement information, which should take at most

© 1992 Information Processing Society of Japan

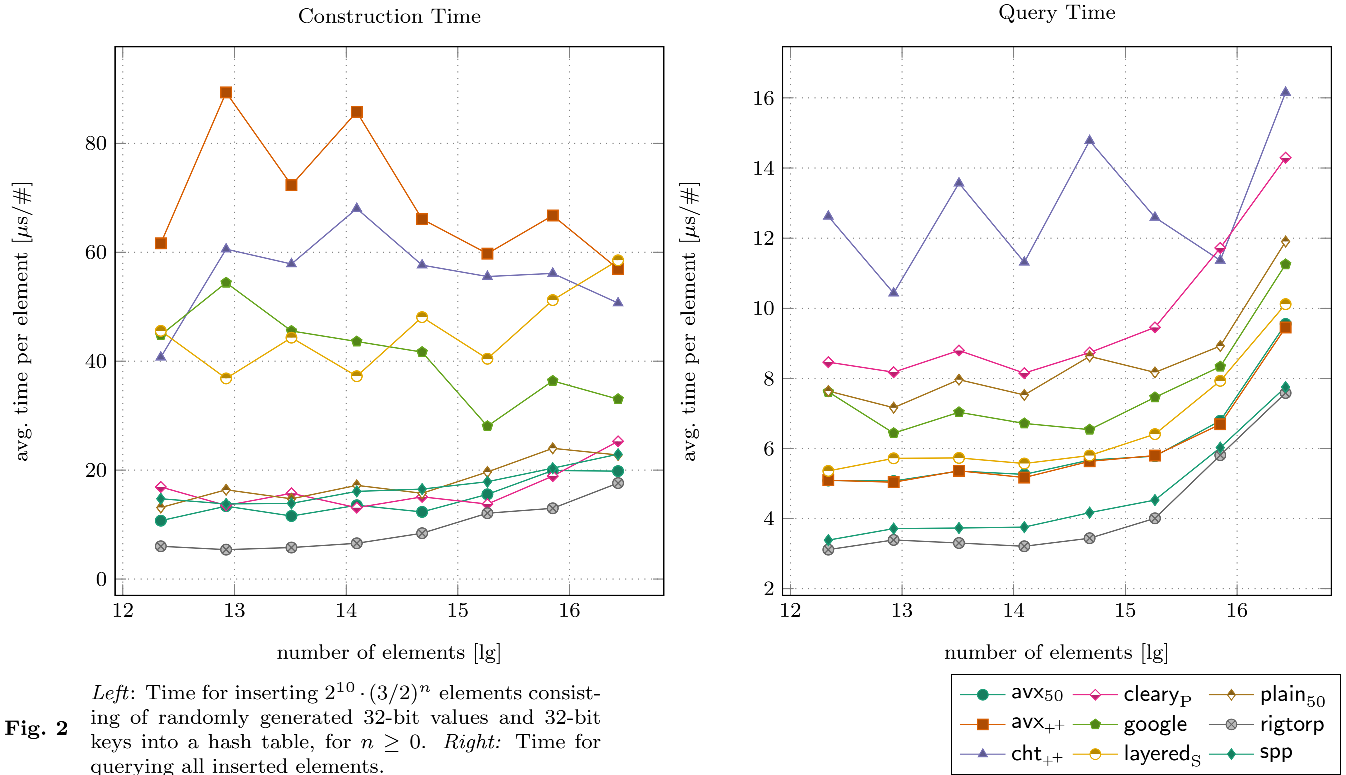

Fig. 2 *Left*: Time for inserting $2^{10} \cdot (3/2)^n$ elements consisting of randomly generated 32-bit values and 32-bit keys into a hash table, for $n \geq 0$. *Right:* Time for querying all inserted elements.

$18\,|H|$ bytes ≤ 1.13 bits per element to be on par with the memory overhead of the separate chaining layout.

3. Experiments

We implemented our proposed hash table in C++17. The implementation is freely available at `https://github.com/koeppl/separate_chaining`.

Evaluation Setting. Our experiments ran on an Ubuntu Linux 18.04 machine equipped with 32 GiB of RAM and an Intel Xeon CPU E3-1271 v3 clocked at 3.60GHz. We measure the memory consumption by overloading the calls to `malloc`, `realloc`, `free` and its C++ counterparts with the tudostats framework[*6]. The benchmark was compiled with the flags `-O3 -DNDEBUG --march=native`, the last option for supporting AVX2 instructions.

Contestants. We selected the following hash tables that are representative C++ hash tables, are sparse, or work with compact hashing.

- std: The `unordered_map` implementation of `libstdc++`. We used the default maximum load factor 1.0, i.e., we resize the hash table after the number of stored elements exceeds the number of buckets.
- rigtorp: The fast but memory-hungry linear-probing hash table of Erik Rigtorp[*7]. The load factor is hard-coded to 0.5.
- google: Google's sparse hash table[*2] with quadratic probing. Its maximum load factor is set to the default value 0.8.
- spp: Gregory Popovitch's Sparsepp[*8], a derivate of Google's sparse hash table. Its maximum load factor is 0.5.
- tsl: Tessil's sparse map[*9] with quadratic probing. Its default maximum load factor is 0.5.
- cleary, elias, layered: The compact hash tables of Cleary [5] and Poyias and Raman [19].
  - elias partitions the displacement into integer arrays of length 1024, which are encoded with Elias-γ [10].
  - layered stores this information in two multiple associative array data structures. The first is an array storing 4-bit integers, and the second is an `unordered_map` for displacements larger than 4 bits.

  The implementations are provided by the tudocomp project[*10]. All hash tables apply linear probing, and support a sparse table layout. We call these hash tables *Bonsai tables* for the following evaluation, and append in subscript 'P' or 'S' if the respective variant is in its plain form or in its sparse form, respectively. We used the default maximum load factor of 0.5.

3.1 Micro-Benchmarks

Our micro benchmarks are publicly available at `https://github.com/koeppl/hashbench` for third-party evaluations. We provide benchmarks for insertions, deletions, and lookups of (a) inserted keys (for successful searches) and (b) keys that are not present in the hash tables (for unsuccessful searches).

*6 `https://github.com/tudocomp/tudostats`
*7 `https://github.com/rigtorp/HashMap`
*8 `https://github.com/greg7mdp/sparsepp`
*9 `https://github.com/Tessil/sparse-map`
*10 `https://github.com/tudocomp/compact_sparse_hash`

© 1992 Information Processing Society of Japan

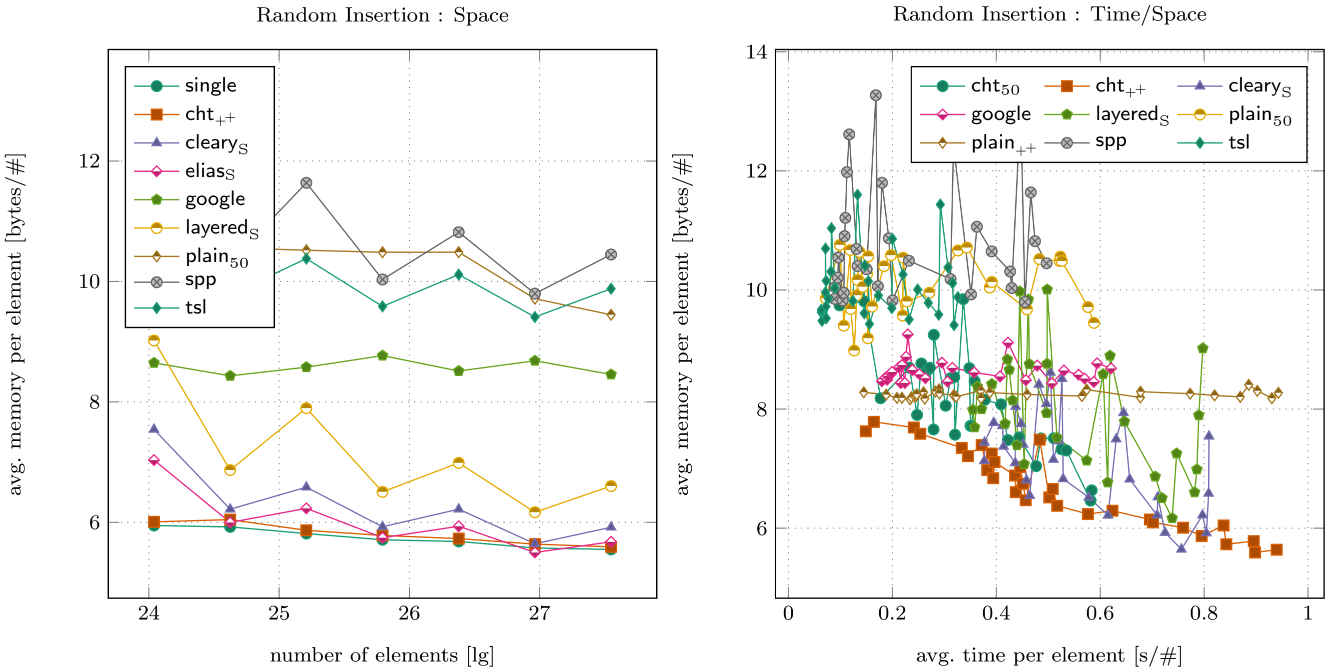

Fig. 3 *Left*: Space needed for constructing the hash tables in the setting of Fig. 2. *Right*: Memory and time divided by the number of stored elements. Each element is composed of a 32-bit key and a 32-bit value, using combined 8 bytes.

Inserting Random Elements. We used `std::rand` as a random generator to produce 32-bit keys and values. The measured times and memory consumptions are depicted in Fig. 2 and Fig. 3. Our variants plain and avx do not use the compact hashing technique. Instead, like other non-compact hash tables, they use the identity function in this setting. Here, avx_{50} is faster than $plain_{50}$ during the construction, and far superior when it comes to searching keys.

While the discrepancy in time between the incremental and the half increase policy is small for most bucket representations, the construction of avx_{++} takes considerably longer than avx_{50}, as we cannot resort to the fast `realloc` for allocating *aligned* memory required for the SIMD operations.

The construction of single is tedious, as it needs to move all values of a bucket on each insertion. On the other hand, its search time is on par with cht_{++}. Our compact and non-compact hash tables match the speed of the sparse and non-sparse Bonsai tables, respectively.

Reversed Space. Like in the previous experiment, we fill the hash tables with n random elements for $n \geq 2^{16}$. However, this time we let the hash tables reserve 2^{16} buckets in advance. We added a percent sign in superscript to the plain hash tables that (a) use our injective transform and (b) take (additionally) advantage of the fact that they only need to store quotients of at most 16 bits. The results are visualized in Fig. 4. We see that $plain^{\%}_{50}$ is superior to google, while $plain^{\%}_{++}$ uses far less space that other non-compact hash tables. Like in Fig. 2, a major boost for lookups can be perceived if we exchange plain with avx[*11], which takes the same amount of space as plain.

Unsuccessful Searches. The search of not stored elements is far more time consuming with our separate chaining hash tables, as can be seen in the left of Fig. 5. When restricted to separate chaining, best bets can be made with avx, as it is the fastest for scanning large buckets. It is on a par with the Bonsai tables, but no match for the sparse hash tables.

Removing Elements. We evaluated the speed for removing arbitrary elements from the hash tables, and present the results in the right of Fig. 5. We used the hash tables created during the construction benchmark (Fig. 2). Interestingly, avx becomes faster than rigtorp in the last instance. The other implementations are on a par with the non-compact sparse contestants. We could not evaluate the Bonsai tables, as there is currently no implementation available for removing elements.

Distinct Keys. We inserted our hash tables into the `udb2` benchmark[*12], where the task is to compute the frequencies of all 32-bit keys of a multiset, in which roughly 25% of all keys are distinct. For that, the hash tables store each of these keys along with a 32-bit value counting its processed occurrences. Our results are shown in Sect. 3.3. We expect from a succinct representation to use space about 2 bytes per key, as about 25% of all keys are distinct, and each (key,value)-pair takes 8 bytes. The evaluation shows that, if time is not of importance, the memory footprint can be considerably improved with our proposed hash table layout.

[*11] avx is not shown in Fig. 4 since our memory allocation counting library does not count aligned allocations needed for avx.

[*12] `https://github.com/attractivechaos/udb2`

© 1992 Information Processing Society of Japan

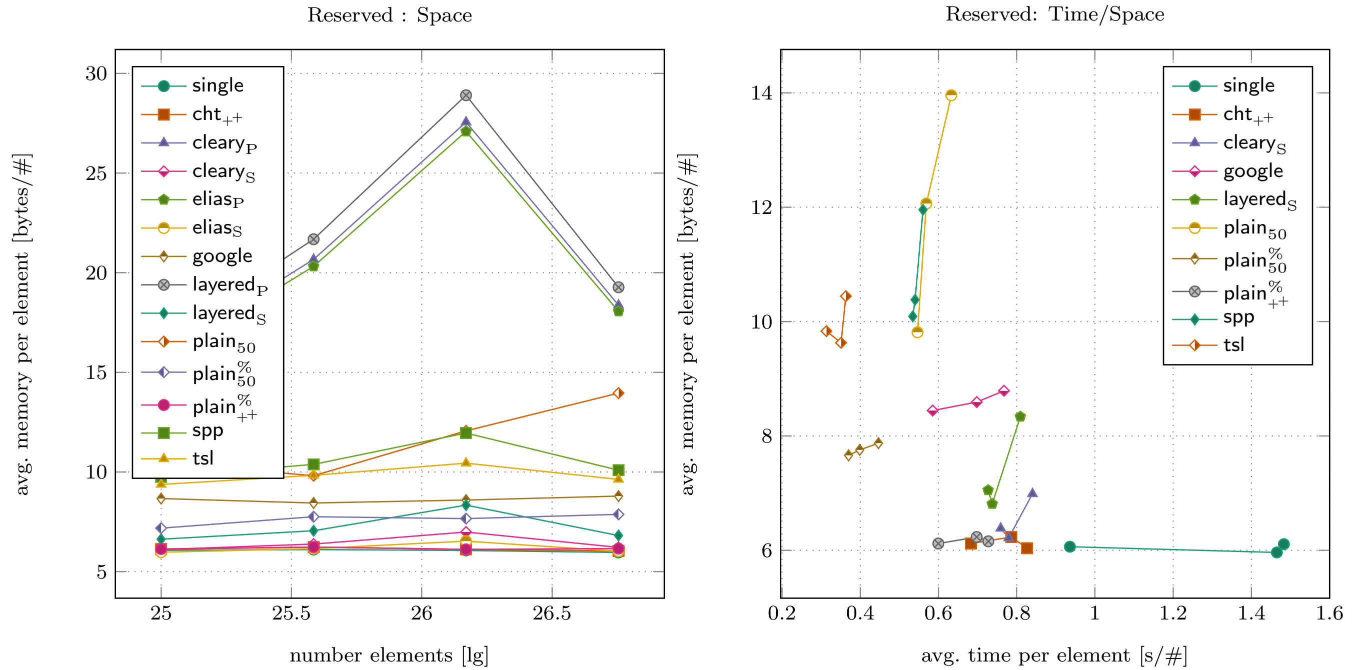

Fig. 4 Time for inserting $2^{25} \cdot (3/2)^n$ random elements for integers $n \geq 0$. The hash tables are prepared to reserve $2^9 \cdot (3/2)^n$ buckets before the insertions start.

3.2 Conclusion

On the upside, the evaluation reveals that our proposed hash tables can be constructed at least as fast as all other compact sparse hash tables (cf. Fig. 2). Our hash tables use less space than any non-sparse compact hash table (cf. Fig. 3). Especially fast are deletions (cf. Fig. 5), outpacing even some speed-optimized hash tables on large instances. Combining avx$_{50}$ with compact hashing can lead to a fast and memory-efficient hash table if there are good lower bounds on the number of elements that need to be stored (cf. Fig. 2 for the time and plain$_{50}$ in Fig. 4 for the space).

On the downside, lookups, especially when searching for a non-present key, are even slower than most of the sparse Bonsai tables, as $b_{\max}$ is much larger than the number of maximal collisions encountered during an insertion of an element in one of the Bonsai tables. That is because their default maximum load factor of $\alpha := 0.5$ gives $c_\alpha \leq 3$ collisions on average for an insertion operation (cf. Eq. (1)).

In total, the major advantage of our proposed hash table layout is its low memory footprint. Its construction speed matches with other memory-efficient hash table representations. However, if the focus of an application is on querying rather than on dynamic aspects as insertion or deletion, Cuckoo hash tables or perfect hashing provide a much better solution.

3.3 Future Work

We think that a bucketized compact Cuckoo hash table [20] based on our proposed hash table layout can be an even more memory-friendly hash table. For that, we store a master and a slave separate chaining hash table whose numbers of buckets are independent from each other. On inserting an element, we first try to insert the element in the master table. If its respective bucket B_{M} is already full, we try to insert the element in the slave table. If its respective bucket B_{S} is also full, we take a random element of both buckets B_{M} and B_{S}, exchange it with the element we want to insert, and start a random walk. By doing so, the distribution of the load factors of all buckets should become more uniform such that a resizing of the hash tables can be delayed at the expense of more comparisons. Both hash tables can be made compact, as each bucket is dedicated to exactly one injective transform (corresponding to its respective hash table).

In the experiments, the measured memory is the number of allocated bytes. The resident set sizes of our hash tables differ largely to this measured memory, as we allocate many tiny fragments of space. A dedicated memory manager can reduce this space overhead, but also reduce the memory requirement of the bucket pointers by allocating a large subsequent array, in which memory can be addressed with pointers of 32-bit width or less. For future evaluations, we also want to vary the maximum load factors of all hash tables instead of sticking to the default ones.

For searching data in an array, the more recent SIMD instruction set AVX2 provides a major performance boost unlike older instruction sets like SSE, as benchmarks for comparing strings[*13] demonstrate a speed boost of more than 50% for long string instances. We wonder whether we can experience an even steeper acceleration when working with the AVX256 instruction set.

In our implementation of cht, we extract the quotients

[*13] https://github.com/koeppl/packed_string

© 1992 Information Processing Society of Japan

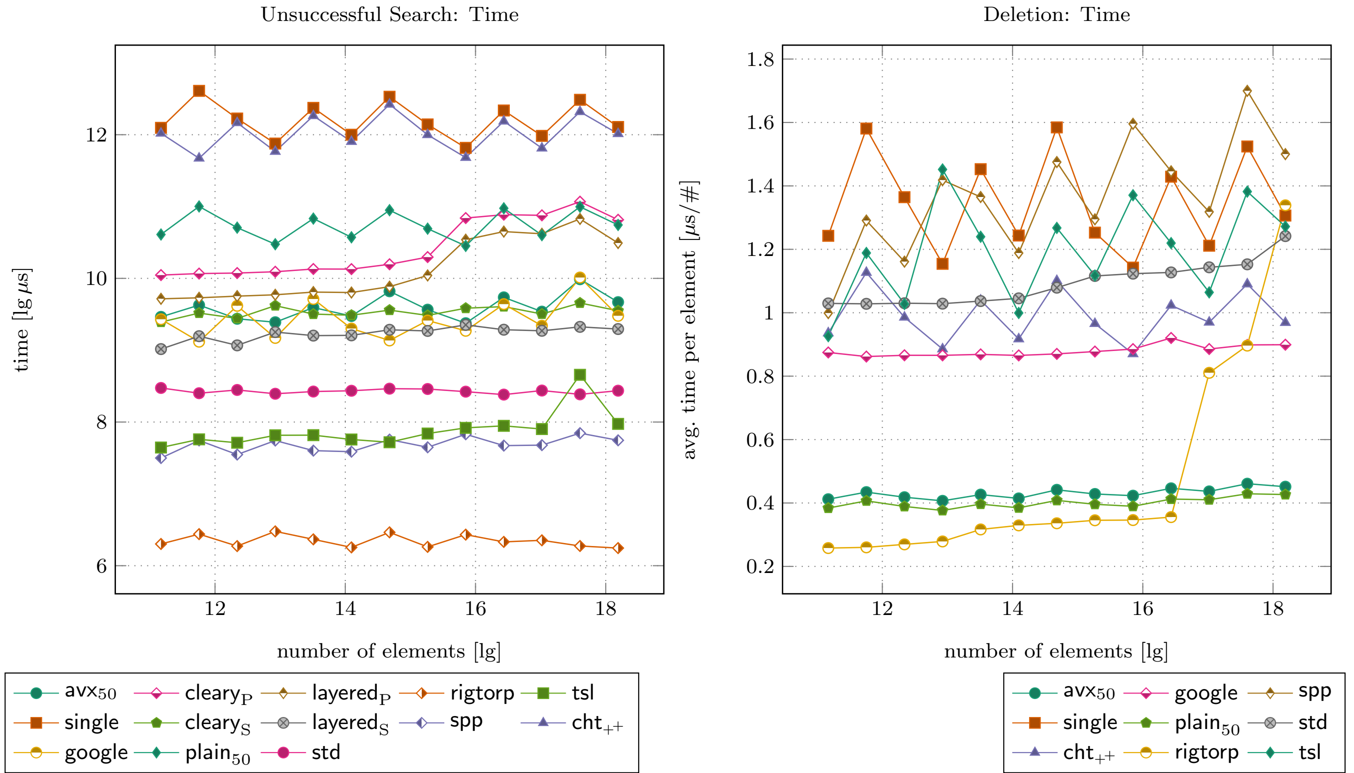

Fig. 5 *Left*: Time for looking up 2^8 random keys that are not present in the hash tables. *Right*: Time for erasing 2^8 random keys that are present in the hash tables. In both figures, the number of elements (x-axis) is the number of elements a hash table contains.

from its bit-compact byte array B *sequentially* during the search of a quotient q. We could accelerate this search by packing q $\lfloor 64/k \rfloor$ times in one 64-bit integer p, where k is the quotient bit width, and compare the same number of quotients in B with $B[i..i+63] \otimes p$ for $i = ck\lfloor 64/k \rfloor$ with an integer c, where we interpret B as a bit vector. Using shift and bitwise AND operations, we can compute a bit vector C such that $C[j] = 1 \Leftrightarrow q = B[i+(j-1)k..i+jk-1]$ for $1 \leq j \leq \lfloor 64/k \rfloor$, in $\mathcal{O}(\lg k)$ time by using bit parallelism.

Finally, we would like to see our hash table in applications where saving space is critical. For instance, we could devise the Bonsai trie [6] or the displacement array of layered [19], which are used, for instance, in the LZ78 computation [1].

Acknowledgments We are thankful to Rajeev Raman for a discussion about future work on compact hash tables at the Dagstuhl seminar 18281, to Marvin Löbel for the implementations of the Bonsai hash tables, to Shunsuke Kanda for the implementation of our used injective transform, and to Manuel Penschuck for running the entropy experiments on a computing cluster.

This work is founded by the JSPS KAKENHI Grant Number JP18F18120.

References

[1] D. Arroyuelo, R. Cánovas, G. Navarro, and R. Raman. LZ78 compression in low main memory space. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 38–50, 2017.

[2] N. Askitis. Fast and compact hash tables for integer keys. In *Proc. ACSC*, volume 91 of *CRPIT*, pages 101–110, 2009.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[4] P. Carlini, P. Edwards, D. Gregor, B. Kosnik, D. Matani, J. Merrill, M. Mitchell, N. Myers, F. Natter, S. Olsson, S. Rus, J. Singler, A. Tavory, and J. Wakely. *The GNU C++ Library Manual*. FSF, 2018.

[5] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers*, 33(9):828–834, 1984.

[6] J. J. Darragh, J. G. Cleary, and I. H. Witten. Bonsai: a compact representation of trees. *Softw., Pract. Exper.*, 23(3):277–291, 1993.

[7] P. J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, 1968.

[8] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

[9] P. Dinklage, J. Fischer, D. Köppl, M. Löbel, and K. Sadakane. Compression with the tudocomp framework. In *Proc. SEA*, volume 75 of *LIPIcs*, pages 13:1–13:22, 2017.

[10] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

© 1992 Information Processing Society of Japan

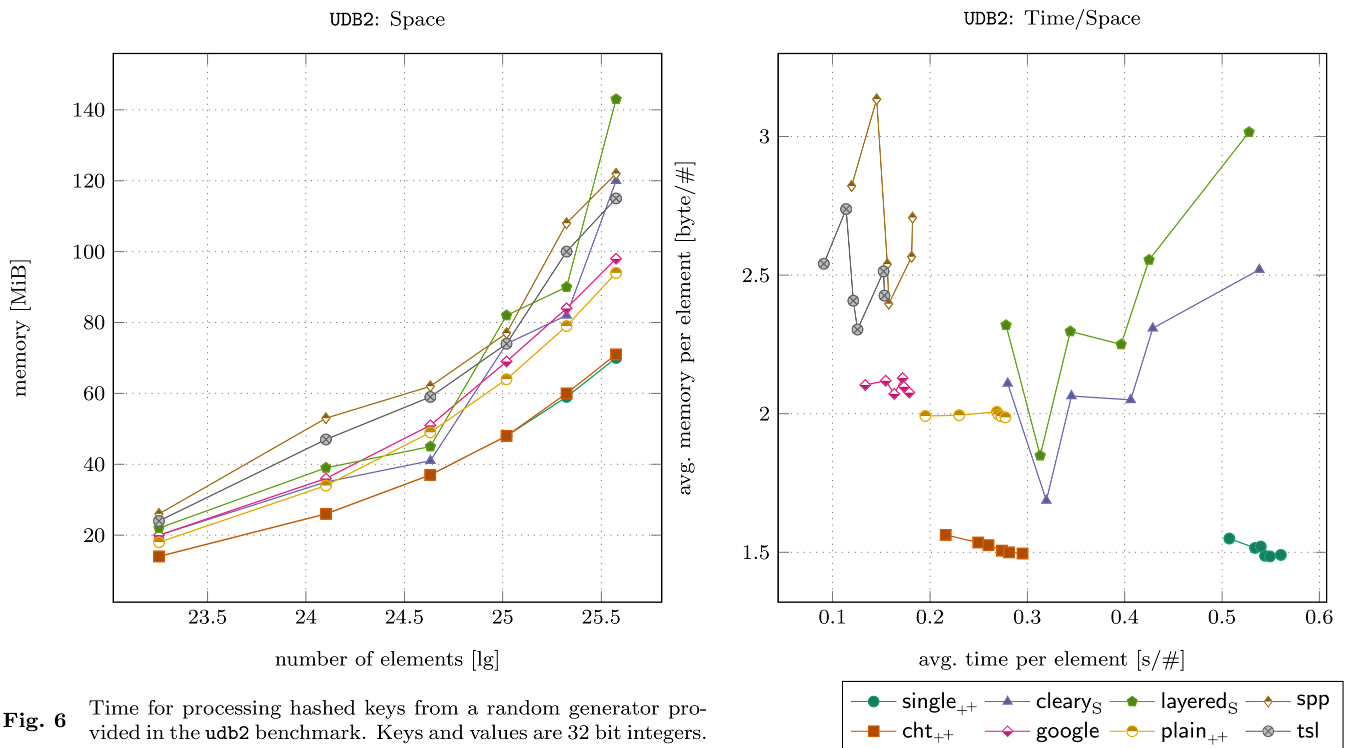

Fig. 6 Time for processing hashed keys from a random generator provided in the udb2 benchmark. Keys and values are 32 bit integers.

[11] J. Fischer and D. Köppl. Practical evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch tries. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 191–207, 2017.

[12] C. Fu, O. Bian, H. Jiang, L. Ge, and H. Ma. A new chaos-based image cipher using a hash function. *IJNDC*, 5(1):37–44, 2017.

[13] D. E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison Wesley, Redwood City, CA, USA, 1981.

[14] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley, Redwood City, CA, USA, 1998.

[15] S. C. Manekar and S. R. Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), 2018.

[16] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(i14), 2003.

[17] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[18] K. Nair and E. RoseLalson. The unique id's you can't delete: Browser fingerprints. In *Proc. ICETIETR*, pages 1–5, 2018.

[19] A. Poyias and R. Raman. Improved practical compact dynamic tries. In *Proc. SPIRE*, volume 9309 of *LNCS*, pages 324–336, 2015.

[20] K. A. Ross. Efficient hash probes on modern processors. In *Proc. ICDE*, pages 1297–1301, 2007.

[21] G. L. Steele Jr., D. Lea, and C. H. Flood. Fast splittable pseudorandom number generators. In *Proc. OOPSLA*, pages 453–472, 2014.

[22] T. Takagi, S. Inenaga, K. Sadakane, and H. Arimura. Packed compact tries: A fast and efficient data structure for online string processing. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100-A(9):1785–1793, 2017.

[23] M. Thorup. Even strongly universal hashing is pretty fast. In *Proc. SODA*, pages 496–497, 2000.

© 1992 Information Processing Society of Japan

Appendix

A.1 Applications

We provide two applications of our hash tables on real-world data sets.

A.1.1 Keyword Fingerprinting

An application of hash tables is to store fingerprints of a set of keywords. We hash each keyword with Austin Appleby's Murmurhash[*14], which is a well received hash function for generating fingerprints [12, 18]. The obtained fingerprint is put into the hash tables. Non-compact hash tables use the identity as a dummy hash function, while the compact hash tables use our injective transform. Such a hash table can be used as a Bloom filter [3] for discarding strings that are not part of the set of keywords. Table A·1 gives the time and space needed for constructing such a Bloom filter. In Table A·2 we measure the time it takes to query for all inserted keywords. We used data sets from [22] and [9], split each dataset into strings by using either the newline or whitespace as a delimiter, and removed all duplicates. We can see that our separate chaining hash table variants use far less space then the compact non-sparse hash tables, while they are smaller or on par with their sparse variants.

A.1.2 Computing the Empirical Entropy

Given a text T of length n whose characters are drawn from a finite alphabet $\Sigma := \{c_1, \ldots, c_\sigma\}$, the empirical entropy H_k of order k for an integer $k \geq 0$ is defined as $H_0(T) := (1/n) \sum_{j=1}^{\sigma} n_j \lg(n/n_j)$ for $n_j := |\{i : T[i] = c_j\}|$, and $H_k(T) := (1/n) \sum_{S \in \Sigma^k} |T_S| \, H_0(T_S)$, where T_S is the concatenation of each character in T that directly follows an occurrence of the substring $S \in \Sigma^k$ in T. We can compute $H_0(T)$ with an array using $\sigma \lceil \lg n \rceil$ bits of space storing the frequency of each character. For larger orders, we count all k-mers, i.e., substrings of length k, in T and iterate over the frequencies of all k-mers of T to compute $H_k(T)$. Using an array with $\sigma^k \lceil \lg n \rceil$ bits, this approach can become obstructive for large alphabet sizes. For small alphabets like in DNA sequences, highly optimized k-mer counters can compute the entropy up to order 55 [15].

Here, we present an approach that stores the frequencies of the k-mers with our separate chaining hash table. Our approach is similar to Jellyfish [17], but more naive as we do not apply Bloom filters or concurrency for speed-up. Instead, our target is to compute the entropy for byte alphabets, orders $k \leq 7$, but massive data sets. For that task, we use $\texttt{cht}_{++}$ and start with byte values representing the frequencies. Whenever the current representation of the frequencies becomes too small, we increment the number of bytes of the frequency representation by one. By doing so, we reach up to 3 bytes per stored frequency in our experiments. As the experimental setup is also of independent interest for computing the empirical entropy of massive data sets, we made it freely available at `https://github.com/koeppl/compression_indicators`.

For the experiments, we took two datasets, `cc` and `dna`, each of 128 GiB. The former data set has an alphabet size of 242, and consists of a web page crawl provided by the commoncrawl organization[*15]. The latter data set is a collection of DNA sequences extracted from FASTA files with an alphabet size of 4. We computed the entropies of each prefix of length $2^n(1024)^3$, for $1 \leq n \leq 7$, for the data sets `cc` and `dna` in Table A·3 and Table A·4, respectively. We summarize the needed time and space for these computations in Fig. A·1. These experiments ran on a computing cluster equipped with Intel Xeon Gold 6148 CPUs clocked at 2.40GHz with 192 GiB of RAM running Red Hat Linux 4.8.5-36. The measured memory is the maximum used resident set size.

We can see that the amount of needed memory becomes saturated after processing the first 2 GiB of `dna`, where we use less than 3 MiB of RAM in total for all orders of k. That is not surprising, as there can be only 4^7 different k-mers of length 7. For `cc`, it is more relevant to have a memory-efficient implementation, as there can be $242^7 \approx 5 \cdot 10^{10}$ 7-mers. We conclude by the strictly monotonic increase of the occupied memory that new k-mers for $k \geq 4$ are found in `cc` even after surpassing the 64 GiB prefix.

[*14] `https://github.com/aappleby/smhasher`

[*15] `http://commoncrawl.org/`

© 1992 Information Processing Society of Japan

hash table	data sets cc	dblp	proteins	urls	wiki
single	21.1	29.0	29.3	176.0	5.0
cht_{50}	25.0	34.9	35.0	218.5	6.0
cht_{++}	21.2	29.1	29.5	177.0	5.0
cleary_{S}	21.3	30.2	30.5	190.8	5.1
elias_{P}	57.5	112.0	112.0	859.7	14.6
elias_{S}	21.0	29.5	29.8	184.8	5.0
google	33.2	46.4	46.9	293.9	7.9
layered_{P}	59.5	116.0	116.1	892.3	15.1
layered_{S}	22.3	32.2	32.5	212.4	5.3
plain_{50}	29.0	40.8	40.9	265.1	6.9
plain_{++}	24.6	34.0	34.4	214.6	5.7
rigtorp	96.7	192.0	192.0	1536.0	24.0
std	85.8	95.9	96.7	706.0	20.5
tsl	35.4	52.0	52.5	339.3	8.4

memory [MiB]

hash table	data sets cc	dblp	proteins	urls	wiki
single	3.8	5.2	5.5	36.2	1.0
cht_{50}	1.5	2.0	2.3	14.4	0.4
cht_{++}	1.9	2.7	3.0	21.1	0.5
cleary_{S}	1.2	2.0	2.4	16.0	0.3
elias_{P}	5.7	8.9	9.3	63.3	1.5
elias_{S}	6.1	9.6	10.1	70.1	1.5
google	1.0	1.4	1.8	12.3	0.3
layered_{P}	0.6	0.9	1.2	7.1	0.2
layered_{S}	1.2	1.9	2.3	15.7	0.3
plain_{50}	0.8	1.1	1.5	9.3	0.2
plain_{++}	1.2	1.8	2.1	16.8	0.3
rigtorp	0.3	0.5	0.8	3.7	0.1
std	0.9	1.2	1.5	9.5	0.3
tsl	0.4	0.7	1.0	6.2	0.2

time [s]

Table A·1 Construction of a fingerprint keyword dictionary.

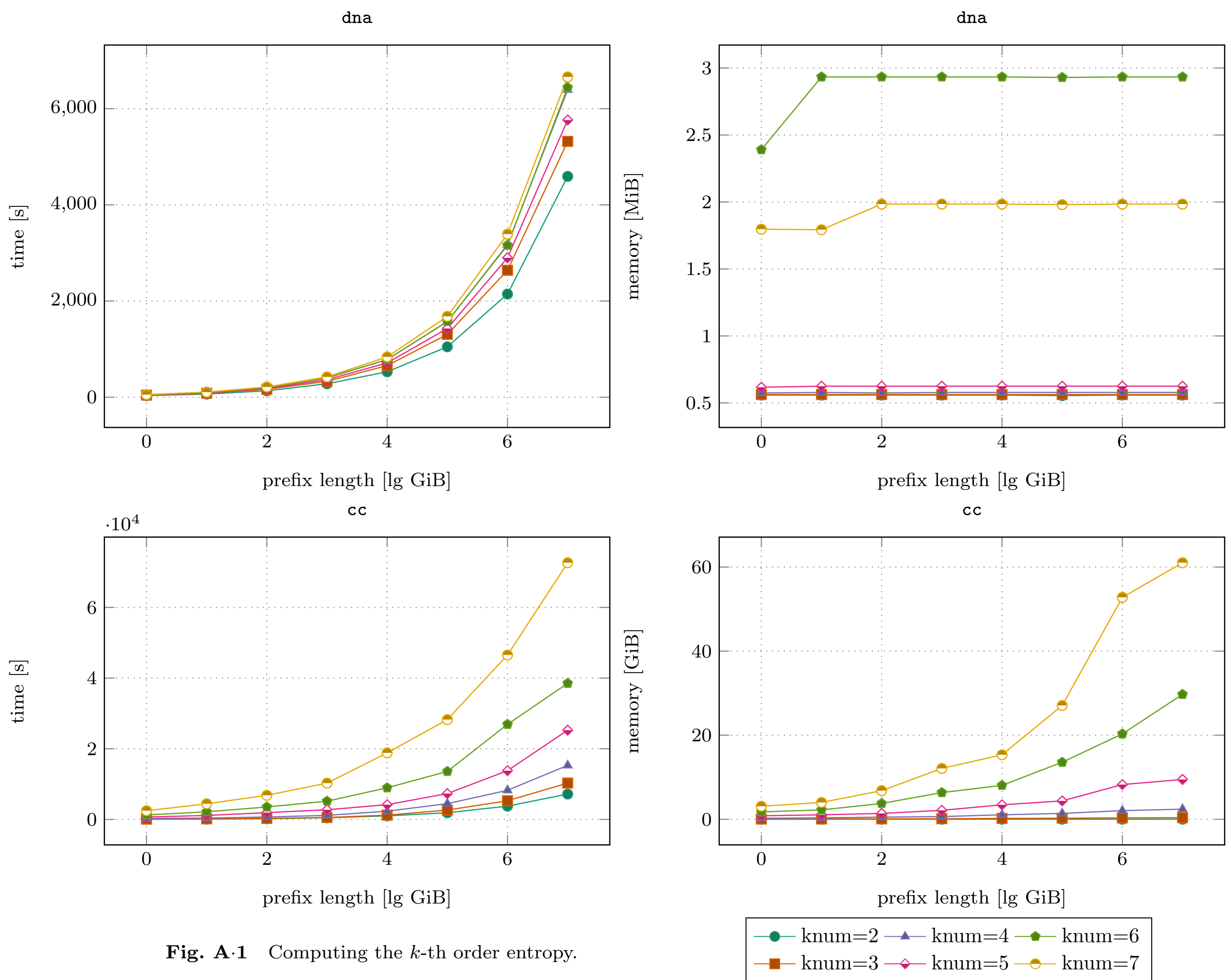

Fig. A·1 Computing the k-th order entropy.

© 1992 Information Processing Society of Japan

hash table	data sets				
	cc	dblp	proteins	urls	wiki
single	0.4	0.8	1.1	5.7	0.2
cht_{50}	0.5	1.0	1.3	7.4	0.2
cht_{++}	0.4	0.8	1.1	5.7	0.2
cleary_{S}	0.4	0.6	0.9	5.3	0.2
elias_{P}	2.6	3.2	3.5	18.8	0.7
elias_{S}	2.6	3.1	3.5	18.4	0.7
google	0.3	0.5	0.8	4.4	0.1
layered_{P}	0.3	0.5	0.7	3.3	0.1
layered_{S}	0.4	0.6	0.9	5.2	0.2
plain_{50}	0.5	0.9	1.2	6.9	0.1
plain_{++}	0.5	0.9	1.3	6.9	0.1
rigtorp	0.3	0.4	0.6	2.7	0.1
std	0.5	0.8	1.0	5.2	0.2
tsl	0.3	0.5	0.8	4.0	0.1

Table A·2 Query time in seconds for a fingerprint keyword dictionary.

prefix	order k					
length	2	3	4	5	6	7
1	3.47259	2.90481	2.35796	1.90477	1.49418	1.18580
2	3.48268	2.92203	2.38605	1.94619	1.54430	1.23745
4	3.48717	2.93171	2.40566	1.97859	1.58677	1.28333
8	3.48762	2.93742	2.41886	2.00233	1.62009	1.32094
16	3.48920	2.94113	2.42738	2.01886	1.64558	1.35130
32	3.49006	2.94411	2.43471	2.03284	1.66737	1.37798
64	3.49100	2.94669	2.44088	2.04409	1.68482	1.40001
128	3.49055	2.94684	2.44231	2.04753	1.69087	1.40839

Table A·3 Empirical entropy of the data set cc. Prefix length is in GiB.

prefix	order k					
length	2	3	4	5	6	7
1	1.94051	1.86247	1.77680	1.69265	1.61940	1.56313
2	1.91210	1.87496	1.83286	1.78250	1.73037	1.68750
4	1.92923	1.91052	1.88797	1.85679	1.82093	1.78944
8	1.93363	1.92250	1.90831	1.88764	1.86061	1.83383
16	1.93166	1.92232	1.91167	1.89491	1.87101	1.84585
32	1.93201	1.92421	1.91507	1.90190	1.88270	1.86160
64	1.93145	1.92424	1.91588	1.90445	1.88763	1.86889
128	1.93873	1.93273	1.92486	1.91341	1.89601	1.87634

Table A·4 Empirical entropy of the data set dna. Prefix length is in GiB.

© 1992 Information Processing Society of Japan