Algorithm Engineering

Peter Sanders

July 22, 2021

Chapter 8

Hashing and Hash Tables

As children we all had to learn that keeping our things in an orderly fashion helps us to find them. Somewhat surprisingly, the contrary can be also true – storing an object at a seemingly random position makes it *more* easy to find. This apparent contradiction is solved by the principle of a *hash function* – if an easy to evaluate function yields the position of the object then it is easy to find. In that situation, it helps if the functions behaves like a random function since this guarantees that only few different objects want to go to the same place.

In this chapter we have a detailed look at this approach. We begin in section 8.1 with a look on hash functions – how they are computed, their basic properties,... There we also summarize different applications. The remainder of the chapter concentrates on hashing as a technique for storing different variants of *dictionaries*, i.e., data structures that allow retrieval of *elements* based on *keys*. Section 8.2 first takes a high level view and introduces different operations possible on hash tables / dictionaries. The section also gives a small sample of the vast number of application.

In Sections 8.3–8.5, we then look at three concrete approaches to implement general purpose hash tables. Hashing with chaining (section 8.3) is perhaps the most simple one but it incurs overheads for pointers to table entries. Higher performance is often possible by storing all elements directly in the table. Section 8.4 discusses *linear probing* – the most simple approach in that direction together with a number of variants and optimizations. *Cuckoo hashing* (section 8.5) then considers a more sophisticated approach that allows very high space efficiency while guaranteeing worst case constant access time.

Section 8.6 discusses how to adaptively grow and shrink hash tables effi-

ciently. The three sections 8.7–8.9 then consider machine model aspects of hash tables, namely memory hierarchies (section 8.7), concurrent access to hash tables (section 8.8), and distributed-memory hash tables (section 8.9).

General purpose hash tables can be considered space efficient when they consume little more space than needed to represent the stored objects. The cuckoo-hash tables from section 8.5 come very close to this goal. However, if we reduce the functionality of the data structure, we can do much better – approaching a few bits per object regardless of the actual object sizes. Sections 8.10–8.12 discuss such data structures. In each case, recent work has progressed to support fast and practically useful data structures that approach information theoretical lower bounds. *Retrieval data structures* (section 8.10) allow function evaluation on a static set of elements without having to store the elements themselves. The best of these data structures basically only require the space for storing the function values. *AMQs* (section 8.11) support *approximate membership queries* (aka Bloom filters) on sets of elements using $\approx \log 1/\phi$ bits where ϕ is the probability to get a false-positive answer. *Perfect hash functions* (section 8.12) assign unique names to elements of a set without any collisions – and require just a few bits per element.

8.1 Hash Functions

A hash function h maps a *key* set K to an integer range $\mathbb{Z}_m = 0..m-1$. Often, m is a power of two, i.e., h generates a number of $\log m$ bits. Ideally, one would like h to be *random*, i.e., h is drawn uniformly at random from the set of all possible mappings from keys to $\mathbb{Z}_m$. This is possible only in exceptional cases (e.g., see [89], Section 11.5) since we usually cannot afford to store and initialze a lookup table of size $|K|$. Nevertheless, we will often assume that h behaves like a random function in order to simplify then analysis. This often works well in practice if some care is taken about the choice of h but there are notable exceptions [312].

Hence, an interesting area of research is the tradeoff between the cost of evaluating/constructing/storing a hash function and how well it approximates a random function. We are facing some gaps between theory and practice here. The hash functions most frequently used in practice [**?**] are fast but little is known about their theoretical performance. However, there are some empirical studies [**?**] that perform a long sequence of statistical tests whether the data produced by a hash function behaves like random data.

In algorithm theory, we can often show the performance of randomized algo-

rithms using a hash function if certain properties are fulfilled. For example, a hash function is *universal* if the probability that two keys collide is $1/m$. This already suffices to guarantee good expected performance for hash tables with chaining [276, Section 4.2]. More generally, a hash function h has the k-way independence property if the hash values of any k different keys behave like independent random variables. For hash tables with linear probing, we need $k = 5$ [**?**].

Open Problem 33 (Provable properties for practical hash functions) Can we prove properties like universality etc. for some popular and fast practical hash functions? Or can we directly prove performance guaranties when used within a particular application like a hash table? If not, can we modify the functions so that they have provable performance without making them significantly more expensive?

Algorithm engineering has already proposed a number of hash functions that bring theory and practice close together – at least for short keys. In particular *tabulation hashing* [334, 255, 1] and its variants is quite fast and has strong performance guarantees. In tabulation hashing, the bits of the key are split into pieces and h is the xor of different tabulated random hash functions applied to the pieces. If the required lookup tables fit in cache, this is quite fast. We can also sometimes get away with not hashing one piece of size $\log m$ [276, Exercise 4.16]. By chaining two tabulation hash function, we can even emulate a truly random hash function [255]. When using long keys, one can first map them to a short key using universal hashing and then use tabulation hashing [255].

In the remainder of this section we first have a closer look at a particular family of hash functions in section 8.1.1. These *linear congruential functions* are fast, easy to analyze, and can be used also for purposes beyond plain hashing. Then we look at additional aspects of hashing. Section 8.1.2 explains how to map bits to arbitrary ranges of values while section 8.1.3 discusses how to regurgitate a limited number of bits to several hash function values in a pseudorandom way. Section 8.1.4 introduces *rolling hash functions* that can be efficiently computed on related subranges of sequences. We summarize algorithmic applications of hash functions beyond hash tables in section 8.1.5 while section 8.1.6 considers hash functions used in cryptography. After briefly discussing the relation of hash functions and pseudorandom number generators in section 8.1.7 we conclude this section with an excursion to pseudorandom permutations in section 8.1.8.

8.4.4 Using Signatures

Finding an element in a hash table may involve many unsuccessful key comparisons. This can be accelerated if we do not compare the full key but only a signature, i.e., a short hash value of the key (e.g. 8 bits). This value should be independent of the position where the element is actually stored. Unsuccessful search then only has to compare signatures. Successful search most of the time only has to look at the single found signature match. This is particularly useful when keys are large or of variable size (e.g., strings). Then one would possibly not even store the keys within the table but only store the signature and a pointer to the element. In that case using signatures can also be considerably more cache efficient than more conventional approaches. Comparisons with signatures can also be accelerated using SIMD instructions. For example, the folly-library supports such an approach [60].

8.5 Cuckoo Hashing

The hash tables discussed so far get very slow (or do not work at all) when their space consumption approaches the space needed to just store the elements. Furthermore, search times are constant only in expectation. *Cuckoo hashing* [242, 116, 98, 199] is a simple and elegant variant of closed hashing that solves both problems at once. The idea is that elements may be stored at H different blocks of size B.[4] Typically $H \in 2..4$ and $B \in \{1,2,4,8\}$. Overall, there are m/B blocks. The H allowed blocks for each element are specified by H hash functions $h_1,\ldots,h_H$ with range $\mathbb{Z}_{m/B}$. The double hashing trick from Section 8.4.2 can be used to obtain all hash function values from two hash functions [220].

With these definitions, find, update, and delete are straightforward and work in worst case constant time. Just try all HB possible slots. This requires H hash function evaluations and (if B is below the cache line size) B cache faults.

The price we pay is that insertions become more expensive. Conceptually, insertion algorithms operate on a directed graph G whose nodes are the buckets and where an element e stored in bucket $v = h_i(e)$ induces the $H-1$ edges $\{(v,h_j(e) : j \neq i\}$. Inserting a new element x amounts to finding a path $u \to \cdots \to v$ in G such that $u \in \{h_1(x),\ldots,h_H(x)\}$ and such that v contains an empty slot.

[4] There are also numerous variants – with one subtable for each h_i, with overlapping buckets [324], etc.

Such a path allows one to put element x into bucket u. Edges on the path indicate which element has to be moved to which alternative bucket.

For basic cuckoo hashing [242] with $H = 2$ and $B = 1$, this path is unique (if it exists at all). Otherwise, we have the choice between a variety of graph search algorithms. However, two approaches are particularly attractive:

Random walk insertion. Starting at some bucket where x can be mapped, check whether the current bucket has a free slot. If so, a path has been found. Otherwise, pick a random element in the current bucket and move it to a random alternative bucket. This algorithm is not guaranteed to find a path if one exists but it will do so with high probability. Its main advantage is that it can be implemented in an online fashion using only a constant amount of temporary space for indicating the current element and the current bucket. The algorithm will run into an infinite loop if no path from the initially chosen bucket to a bucket with a free slot exists. One therefore stops the search when a bound on the maximum number of allowed steps is exceeded. In that case, the table is rehashed, i.e., rebuilt using a fresh hash function.

BFS insertion. The algorithm performs a breadt-first search (BFS) in the graph starting at the possible buckets for element x. The search is stopped when a free slot is found, when no new nodes have can be found, or when a bound on the maximum number of nodes to be explored is exceeded.[5] This algorithm has two advantages over random walk insertion. First, at least in principle, this algorithm can deterministically decide whether a feasible path exists. Second, the paths found have minimal length and hence, the necessary updates on the data structure are minimized. Its main disadvantage is the additional space for maintaining the search frontier. However, with an appropriate limit on the size of the search space, this data structure is small and fits in the cache.

Maximum load factors. For each choice of H and B, there is a constant threshold $\hat{\alpha}(H,B)$ such that for any constant load factor $\alpha < \hat{\alpha}(H,B)$ a feasible assignment of elements to slots can be found with high probability, whereas for $\alpha > \hat{\alpha}(H,B)$ no feasible assignment exists with high probability. For the basic case $H = 2$ and $B = 1$ this is the same threshold as the well known threshold on the sudden emergence of a giant component in a random graph [242, 116]. For larger values of H and B, these threshold approach 1 surprisingly fast [273, 116, 98, 94, 122, 118, 117].[6] Table 8.1 gives threshold values for important combinations of parameters.

[5] The implementation in [199] does not check for nodes that have already been visited. Since

Table 8.1: Threshold values $\hat{\alpha}(H,B)$ for cuckoo hashing [65, 324]

$H\backslash B$	1	2	3	4	5	6	7	8
2	.5	.897	.959	.980	.989	.994	.996	.998
3	.918	.988	.997	.9992				
4	.977	.998	.998	.99997				

Open Problem 41 (Insertion cost of cuckoo hashing) Cuckoo hashing is known to have constant expected insertion time. There is also intensive work on high probability bounds which are (poly)loagarithmic [218]. However, characterizing the expected insertion time as a function of the load factor remains perhaps the biggest open problem in analyzing cuckoo hashing. Experiments suggest the following conjecture:

Conjecture 3 BFS insertion or random walk insertion achieve expected insertion time $O(1/\varepsilon)$ when the load factor is below $\hat{\alpha}(H,B)-\varepsilon$.

Proving this bound would be intriguing since it would actually match the insertion time bound of random rehashing, i.e., the more complicated insertion procedure needed for cuckoo hashing is asymptotically no disadvantage.

Assuming the above conjecture implies that constructing a cuckoo hash table using incremental insertion has only logarithmic dependence on the distance to the threshold:

Corollary 4 Building a cuckoo hash table with load factor below $\hat{\alpha}(H,B)-\varepsilon$ by incremental insertion is possible in time $O(n \log 1/\varepsilon)$.

Proof: Let $\varepsilon_i = \hat{\alpha} - i/m$ denote the distance to the threshold when inserting the i-th of $n = (\hat{\alpha}-\varepsilon)m$ elements. If conjecture 3 is true, we can then bound the asymptotic time for inserting n elements by

$$\sum_{i=0}^{n} \frac{1}{\hat{\alpha} - \frac{i}{m}} \approx \int_0^n \frac{1}{\hat{\alpha} - \frac{i}{m}}\, di = n \ln\left(\hat{\alpha} - \frac{n}{m}\right) - n\ln(\hat{\alpha}) \leq n \ln \frac{1}{\varepsilon}.$$

∎

such nodes are rare anyway, this speeds up the search.

[6] Even larger thresholds can be achieved for overlapping buckets [324]. However, this implies more complex insertion operations and is not readily compatible with fixed cache lines.

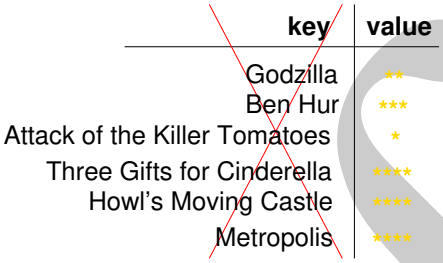

Figure 8.2: A retrieval data structure for movie ratings.

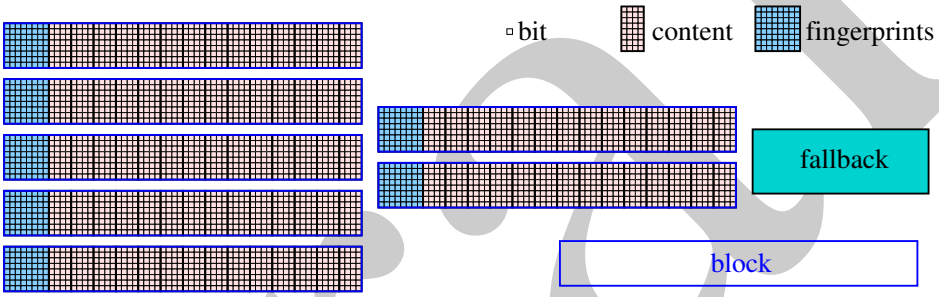

Figure 8.3: FiRe data structure with $r = 32$, $B = 14$, $k = 64$, and $L = 2$. Note that one block fits into a 64-byte cache line.

8.10 Retrieval Data Structures

A basic and frequently needed functionality of a hash table is to access a value based on a key. We will now see that this can be supported without actually storing the keys. If the keys are long, while the associated information is short, this can drastically reduce the required space. Figure 8.2 illustrates this with a simple example – if you have a database rating movies with 1–4 stars, you just need 2 bits to store the ratings even if the movies have long names. The price we pay is that this only works if we use keys actually present in the table. Trying to access the value associated with a nonexistent key will give an arbitrary result.

These *retrieval data structures* are typically static, i.e., they are constructed for a fixed set of keys given as input. We can distinguish two important subvariants – those allowing dynamic update of the associated information and *immutable* (or *fully static*) ones that don't.

More formally, an r-bit retrieval data structure allows evaluating a function

$f : K \to \{0,1\}^r$ on a set $S = \{s_1, \ldots, s_n\} \subseteq K$. In the following, we present several techniques for supporting retrieval together with examples that usually use a combination of several techniques. Section 8.10.1 begins with the simple approach to first break down the problem into many small subproblems. Another simple technique is to filter out collisions of a hash function (section 8.10.2). We can also use hash functions that do not produce collisions in the first place, i.e., delegate the problem of retrieval to that of constructing a perfect hash function (see section 8.10.3 and section 8.12). Finally, section 8.10.4 reduces the retrieval problem to computing a linear combination of several table entries (usually xoring bits). Constructing the data structure then amounts to solving a system of linear equations.

Applications. Static retrieval data structures have a wide range of applications. For example, by storing a random signature for each key, we get an approximate membership query data structure (AMQ or Bloom filter replacement) with information-theoretically optimal false positive probability 2^{-r}; see section 8.11.4. This is useful in applications involving memory hierarchies or distributed computing [58]. The cases $r = 1$ and $r = 2$ can be used to represent perfect hash functions; see section 8.10.3. In turn, this can be used to obtain *updateable retrieval data structures* where the function value can be changed. To exemplify the deep chain of involved applications, consider the case where updateable retrieval supports semi-external DFS, which is needed for cycle detection in large graphs, which is important for model checking, which in turn can be used for various purposes in hardware and software verification [101].

Retrieval data structures can also be used to directly store compact names of objects, e.g., in column-oriented databases [226]. This takes more space than perfect hashing but allows to encode the ordering of the keys into the names.

In several of these applications, retrieval data structures occupy a considerable fraction of RAM in large server farms. Even small reductions (say 10 %) in their space consumption thus translate into sizable cost savings. Whether or not these space savings should be pursued at the price of increased access costs depends on the number of queries per second. The lower the access frequency, the more worthwhile it is to occasionally spend increased access costs for a permanently lowered memory budget. Sophisticated implementations use multiple variants of compressed data structures at once based on known access frequencies of different parts of the database [224]. Thus, the entire set of Pareto-optimal variants with respect to the space–access cost tradeoff is relevant for applications.

We remark that once we can do 1-bit retrieval with low overhead, we can use

that to store data with prefix-free **variable-bit-length encoding** (e.g. Huffman or Golomb codes). We can store the k-th bit of element x as data to be retrieved for the input tuple (x,k). This can be further improved by storing R 1-bit retrieval data structures where R is the largest number of bits needed for representing an input [156, 37, 128]. By interleaving these data structures, one can make queries almost as fast as the case of fixed r.

8.10.1 Using Partitioning

A trick that works for many hashing based problems is to first partition the elements into m' buckets using a primary hash function h_p with range $\mathbb{Z}_{m'}$. This has several advantages. We only need to allocate temporary space for constructing the retrieval data structure on a per bucket manner. We can parallelize over the buckets, and we can use external memory construction. This also makes it practical to use construction algorithms that use superlinear time – the superlinearity will only be in terms of the bucket size and not in terms of the overall input size. The downside is that there is a space overhead per bucket so that the overall space consumption grows with the number of buckets.

Porat [248] exploits the partitioning approach to its extremes. He first partitions the input into blocks of $\mathrm{O}\left(\log^2 n\right)$ expected size. Then each block is partitioned once more into subblocks of size $\mathrm{O}(\sqrt{\log n})$. This problem size is so small that the random matrix approach from Section 8.10.4.2 can be used in constant time using table lookup. With a few additional tricks, preprocessing time is $\mathrm{O}(n)$ and query time is $\mathrm{O}(1)$ using bit parallelism. The space overhead is asymptotically dominated by storing a constant number of bits per subbucket for selecting the hash functions to be used there. This is $\mathrm{O}(n/\sqrt{\log n}) = o(n)$.

Belazzougui and Venturini [37] use slightly larger buckets of size $\mathrm{O}((1+\log\log(n)/r)\log\log(n)/\log n)$. Using carefully designed random lookup tables they show that linear construction time, constant lookup time, and overhead $\mathrm{O}\left((\log\log n)^2/\log n\right)$ is possible. This even applies to result sets with arbitrary distributions.

8.10.2 Filtering out Collisions

A very simple and useful technique for designing retrieval data structures is to use a method that does not work perfectly and to repair it by bumbing colliding elements to a fallback data structure. This approach can also be cascaded to multiple levels. We first exemplify this approach using a very simple and fast variant in

```
Class FiReBucket(E : Array of Element, var bumped : Sequence of Element)
    fingerPrint = 0^k : BitArray[0..k−1]                          // fingerprints
    t : Array [0..B−1] of {0,1}^r                                 // content
    static count = 0^|E| : Array [0..k−1] of 0..|E|   // # of occ. of each fingerprint
    for i := 1 to |E| do count[h_f(E[i])]++           // 1st scan: count occurrences
    for (i = 1, j = 1;  i ≤ |E|;  i++)                // 2nd scan: place each element
        if count[h_f(E[i])] = 1 then                  // store noncolliding element
            t[j++] := f(E[i])                         // store function value
            fingerPrint[fPos[i]] := 1                 // remember fingerprint
            if j > |E| then move E[i+1..|E|] to bumped;  return   // bucket full
        else bumped.pushBack(E[i])        // bump colliding element to next layer

    Function retrievable(e) return fingerPrint[h_f(e)] = 1
    Function retrieve(e) return t[fingerPrint.rank(h_f(e))]
```

Figure 8.4: Pseudocode for a bucket of the FiRe data structure. Note that the array *count* can be reused between calls of the constructor but deallocated after all buckets are constructed. Double evaluation of h_f can be avoided by caching the values computed in the first scan of E.

section 8.10.2.1. We generalize this in section 8.10.2.2 presenting a highly generic approach that is amenable to highly space efficient variants.

8.10.2.1 Filtered Retrieval (FiRe)

In the FiRe approach [226], elements are mapped to buckets with B slots for r-bit values. They are also mapped to fingerprints from $\mathbb{Z}_k$. Preprocessing for a bucket selects up to B elements that have unique fingerprint among the elements mapped to that bucket and stores their associated value in the slots of the bucket. Figure 8.3 shows an example of the data structure. Figure 8.4 gives pseudocode for constructing and querying buckets. This pseudocode is intentionally formulated in a fairly low-level fashion in order to underline how fast and simple the construction is. In practice, the dominating cost will be for sorting the elements into buckets. Both sorting and bucket construction can be parallelized well. Evaluation of hash functions can be vectorized.

The bumped elements are delegated to the next layer of the data structure using

```
layer : Array [1..L] of Array of FiReBucket
fallback : RetrievalDataStructure          // for elements bumped from level L
Function FiRe::retrieve(e : Element) : {0,1}^r
    for i := 1 to L do                     // try one layer after the other
        bucket:=layer[i].b[h_b^i(e)]       // h_b^i(e) adresses buckets in layer i
        if bucket.retrievable(e) then return bucket.retrieve(e)
    return fallback.retrieve(e)
```

Figure 8.5: Pseudocode for retrieval from a FiRe data structure

fresh hash functions. Assuming that we keep the fraction of bumped elements constant, subsequent layers shrink geometrically. In order to ensure worst case constant access time, one can stop the cascade after a fixed number L of layers. The remaining bumbed elements are then stored using some other retrieval data structure. Since this affects only a tiny fraction of the origninal input and since only few queries can affect it, the choice for this fallback layer is uncritical – it need not be particulaly fast nor easy to construct.

Perhaps the main complication in the FiRe data structure is the considerable number of parameters: B, k, L, and the number b of buckets allocated to a layer. One approach is to first decide on the size of a bucket, e.g., a cache line size like $C = 512$ bits. Assuming an uncompressed representation of the fingerprints, this imposes the constraint $k + Br \leq C$. Hence, when we choose k, we can infer $B = \lfloor (C-k)/r \rfloor$. We want to choose $b = \alpha n/B$ for a tuning parameter α. The number X of elements mapped to a particular fingerprint is approximately Poisson distributed with parameter $\lambda = n/kb = B/k\alpha$. Thus, the probability $p_1 = \text{prob}(X = 1) = \lambda e^{-\lambda}$ gives the probability that a particular fingerprint is noncolliding. In total, the expected number of noncolliding elements in a bucket is

$$B_1 = k \cdot p_1 = k\lambda e^{-\lambda} = \frac{B}{\alpha} e^{-\frac{B}{\alpha k}}.$$

B_1 is maximized for $\alpha^*(k) = B/k$ and then takes the value k/e. Since, for reasons of space efficiency, we want to have $B_1 \geq B$, we should therefore usually choose $k \geq eB$. This implies that $B \leq C/(r+e)$.

For fixed k, $\alpha = \alpha^*$ maximizes space efficiency since a larger expected number of noncolliding elements also means that less buckets are underloaded and hence waste space. Choosing a larger value for α will reduce the number of

bumped elements and thus reduces the expected number of accessed layers during queries – a classical space–time tradeoff.

To assess the actual space overhead, we also need to know the expected number of empty cells. The number of noncolliding elements in a bucket approximately has the Binomial distribution $B(k, p_1)$. Hence, the expected number of empty cells is approximately

$$B_0 = \sum_{i=0}^{B-1} (B-i) \binom{k}{i} p_1^i (1-p_1)^{k-i}.$$

Therefore, the space overhead per element is about

$$s = (rB_0 + k)/(B - B_0).$$

The expected number of noncolliding elements bumped from a bucket is

$$B_> = \sum_{i=B+1}^{k} (i-B) \binom{k}{i} p_1^i (1-p_1)^{k-i}.$$

The probability for an element to be colliding is

$$p_{\text{coll}} := 1 - \left(1 - \frac{1}{bk}\right)^{n-1} \approx 1 - e^{-B/\alpha k}.$$

Thus, the expected number of colliding elements is about np_{coll} and the expected number of colliding elements per bucket is $p_{\text{coll}}B/\alpha$. Overall, the expected number of bumped elements per bucket is $B_{\text{bump}} = B_> + p_{\text{coll}}B/\alpha$.

Together with the colliding elements, the expected number of bumped elements per bucket is $B_{\text{bump}} = B_> + (1 - p_0 - p_1)k$ where $p_0 = e^{-B/\alpha k}$ is the probability that a particular fingerprint is not observed at all. Then, bB_{bump} is the overall expected number of bumped elements and $p_{\text{bump}} = bB_{\text{bump}}/n = \alpha B_{\text{bump}}/B$ is the expected fraction of bumped elements. This results in

$$\ell = 1/(1 - p_{\text{bump}})$$

expected layer accesses (for $L = \infty$).

Figure 8.6 plots the resulting space-time tradeoff for 512-bit cache lines. The chosen values for α start at $\alpha^* = B/k$ since smaller values are dominated. We see that choosing $\alpha > \alpha^*$ steeply improves ℓ. Choosing k such that $k \approx eB$ is indeed the most space efficient choice. However, from a performance point of

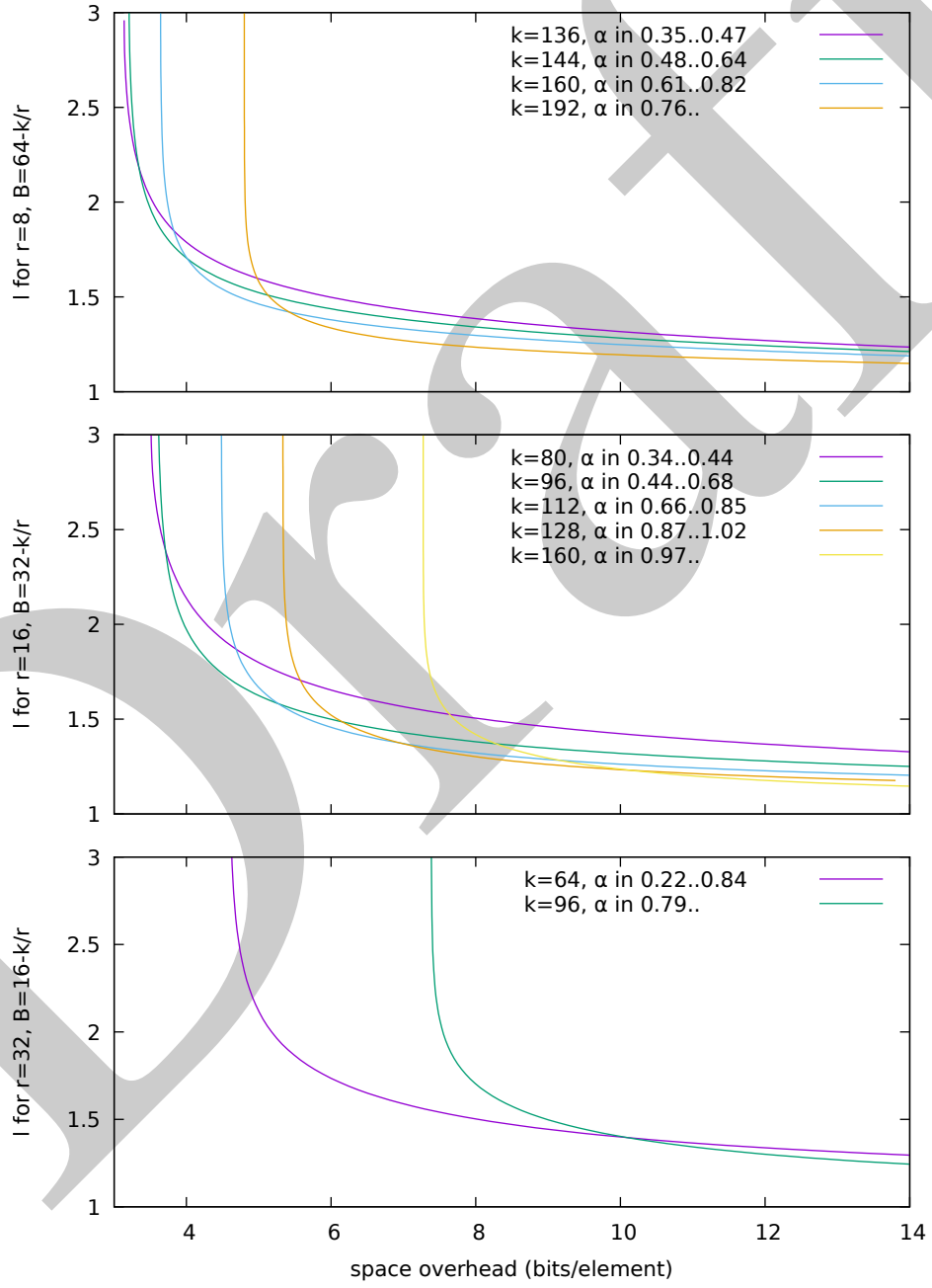

Figure 8.6: Space versus time (expected number of layer accesses ℓ) for different parametrizations of the FiRe data structure assuming 512 bit cache lines. The α-ranges given in the key indicate values where this values of k yields a Pareto-optimal configuration.

```
Class BuRe(E : Set of Element)
    primary : ImperfectRetrieval                    // stores most data
    fallback : Retrieval                    // for elements bumped from primary
    build primary from E and let b indicate the bumped elements
    build fallback from b

    Function retrieve(e)
        if primary.isBumped(e) then return fallback.retrieve(e)
        else return primary.retrieve(e)
```

Figure 8.7: Bumped retrieval

view, this choice is only good for rather large r ($r = 32$ in the plots). For smaller r, the considerable number of bumped elements implies around $\ell \approx 3$ required layer accesses in expectation. To get ℓ close to one, configurations with larger k are better.

In general, FiRe is fast and simple and yields high relative space efficiency for large r. However, for small r (≤ 4), the space overhead easily becomes larger than the "payload" of stored data. There are more space efficient methods with slower construction and retrieval.

Open Problem 48 (Engineering Filtered Retrieval (FiRe)) It might be worth studying different implementation tradeoffs for FiRe. In which situtations are they competitive with other methods even for small r? In that case k is large and the cost of rank operations becomes an issue. Can they be accelerated using SIMD instructions? Should we rather use buckets smaller than a cache line? How can sparse fingerprint vectors be compressed? Perhaps using Elias–Fano encoding as in section 8.11.3.2? (Initial calculations indicate that Elias–Fano encoding as in section 8.11.3.2 does not provide siginificant compression for the relatively high densitiy bit vectors under consideration.)

8.10.2.2 Generic Bumped Retrieval (BuRe)

Let us now complement the very specific FiRe data structure with a highly generic approach BuRe that perhaps better illuminates the basic idea behind the filtering-out approach. Figure 8.7 gives pseudocode for this approach that is parameterized with two data types we are free to choose: A *primary* retrieval data structure that

is supposed to contain most elements, and should be both fast and space efficient. To facilitate this, it is allowed to *bump* some elements, i.e., when *primary* is constructed, it can return a (hopefully small) set of elements b that cannot be retrieved from primary. These elements are then placed into a *fallback* retrieval data structure. At query time, a function *isBumped* decides whether the data can be retrieved from *primary* or from *fallback*. Since bumped retrieval itself implements a reliable retrieval data structure, we can recurse, i.e., *fallback* can have data type *BuRe* itself etc. This recursion can be continued until no more elements are bumped or until the there are so few bumped elements that a more expensive reliable retrieval data structure can be used as the ultimate fallback.

In section 8.10.4.2 we apply the BuRe approach to immutable retrieval, and obtain a practical approach that can get the space requirement arbitrily close to the information theoretical limit. This also yields an near-optimal AMQ data structure

8.10.3 Using Perfect Hashing

In section 8.12 we will see ways to build injective hash functions $h : S \to \mathbb{Z}_m$ that need constant time to evaluate and $O(|S|)$ bits of space – typically around 2 bits per element. This reduces (updateable) retrieval to plain array access. We simply store the value $f(x)$ in an array a at position $h(x)$ In comparison to FiRe, this allows considerably smaller space overhead, in particular for small r. The price is in increased construction time and query time. In particular, evaluating $f(x)$ implies first accessing the data structure representing the hash function h and then accessing the array a.

8.10.4 Solving Systems of Linear Equations

We now turn to a particularly space efficient yet immutable family of retrieval data structures that allows to eliminate space overheads almost entirely. The idea is to use a table t of m entries with r bits each. A key x is mapped to k hash functions with range $\mathbb{Z}_m$ and the computed output is

$$f(x) := t[h_1(x)] \oplus \cdots \oplus t[h_k(x)]$$

where "$\oplus$" is (usually) the bit-wise xor operation. There is a complex tradeoff between the necessary size of t (i.e., space overhead), how difficult it is to find the entries of t (i.e., construction time), and how expensive function evaluations are (how large is k? how many cache lines are touched?).

In general, finding the entries during construction can be modelled as solving a set of linear equations over the field $\mathbb{F}_2 = \{0,1\}$ with r right-hand sides. We are solving a system $At = b$. A is an $n \times m$ 0/1-matrix where the ones in row i are at positions $h(s_i)$, t is an $m \times r$ matrix representing the table to be computed, and F is an $n \times r$ matrix specifying the values to be retrieved. Solving this system is possible if A has full rank n, i.e., if all rows are linearly independent. Brute force solution of $At = F$ is too slow for most applications so that we look for fast solutions.

8.10.4.1 Hypergraph Peeling

In some cases, the equation $At = F$ can be solved in linear time by a simple greedy algorithm. Suppose there is a column j of A that only has a single nonzero entry $a_{ij} = h_z(s_i) = 1$. Then we can temporarily remove column i and row j from the equation, solve the remaining system, reinsert row i and column j, and then set

$$t[j] := \bigoplus_{y \in 1..k \setminus \{z\}} t[h_y(s_i)].$$

This process is called *hypergraph peeling* based on viewing the constraint matrix as a hypergraph $H = (V = 1..m, E = \{e_1, \ldots, e_n\})$ with $e_i = \{j : a_{ij} = 1\}$. It turns out that for many randomized constructions of the constraint matrix A, the peeling process manages to solve the full system with constant probability. A natural choice are $k = 3$ random hash functions and $m > 1.23n$ columns [54]. Walzer [325] showed that by choosing *coupled* $h_1(x), \ldots, h_k(x)$ from a limited, randomly choosen window, even better space efficiency can be achieved using peeling. For example, space overheads 0.11, 0.05, and 0.03 are possible for $k = 3$, $k = 4$, and $k = 7$, respectively at $n = 10^7$. Evidence is given that the window size should scale like $n^{2/3}$ for optimal space efficiency.

Figure 8.8 gives pseudocode for the peeling process. The algorithm represents the nonzeroes of each column as the xor of the row indices of these nonzeroes. Thus, when the number of nonzeroes reaches one, the index of that nonzero can be retrieved from this entry. This saves time and space compared to a more explicit representation of a dynamically changing hypergraph. In particular, space consumption is $O(m)$ independent of the number of used hash functions k.

```
Procedure peel((x_0, f_0), ..., (x_{n-1}, f_{n-1}), var t : (\{0,1\}^r)^m, h_1, ..., h_k)
    loop
        choose fresh hash functions h_1, ..., k_k
        t := 0^m                                          // output retrieval data structure
        I := 0^m                                          // xor of rows incident to each column
        C := 0^m                                          // # of rows incident to each column
        for i := 0 to n - 1 do
            for ℓ := 1 to k do  j := h_ℓ(x_i);   I[j] ⊕= i;   C[j]++
        Q = {j : C[j] = 1} : Stack of Z_m                 // peelable columns
        s = ⟨⟩ : Stack of Z_n × Z_m                       // peeled columns
        repeat n times                                    // peel one column
            if Q = ⟨⟩ then next iteration of outer loop   // construction failed
            j := Q.pop                                    // peel column j
            i := I[j]                                     // the only remaining row incident to column j
            s.push((i, j))
            for ℓ := 1 to k do
                j' := h_ℓ(x_i);   I[j'] ⊕= i;   C[j']--
                if C[j'] = 1 then Q.push(j')              // new peelable column
        foreach (i, j) ∈ s do                             // pop one entry at a time
            t[j] := f_i ⊕ ⨁_{ℓ=1}^{k} t[h_ℓ(x_i)]         // assign entry j
        return                                            // success
```

Figure 8.8: Using hypergraph peeling to build a retrieval data structure t such that $\forall i : \oplus_{1 \le \ell \le k} t[h_\ell(x_i)] = f_i$.

8.10.4.2 Fast Solvers for the Equation Systems

Solving the systems of equations directly is potentially more space efficient than the peeling approach from section 8.10.4.1. For example, a random square matrix has full rank with constant probability. Thus, it suffices to store a constant number of bits (in expectation) to encode which attempt at finding a full rank matrix suceeded in order to achieve an otherwise perfectly space efficient retrieval data structure requiring only rn bits of space to store r bits for n input elements. There is a number of approaches to make equation solving more practical:

Partitioning: Using the partitioning trick from section 8.10.1 can drastically reduce the size of the equation systems to be solved at the expense of more overhead for a data structure navigating to the individual buckets and per bucket data to encode which attempts worked there.

Sparse, structured systems: With the right solvers, solving the system of equations can be accelerated by choosing systems with few nonzeros, perhaps having a structure that additionally simplifies the solution.

Word parallelism: Recall that the constraint matrix just has single bit entries. Thus key operations like row operations can be accelerated by a factor w on a machine with word size w. Similarly, if the number of bits r is not too large, also the retrieval operation can be accelerated if it addresses multiple close-by positions in the table t.

For example, we can use a partitioning hash function h_p to map elements to buckets of size $\Theta(\log p)$ with high probability (at least that works for random h_p). For a bucket b with n_b elements mapped to it, we can then set up an $n_b \times n_b$ system of linear equations $Ax = F$. Using word parallelism, each of these $O(n/\log n)$ systems can be solved in time $O\big(\log^3 n/w\big) = O\big(\log^2 n\big)$, i.e., overall construction costs are $O(n \log n)$. The involved constant factors may be quite favorable since the operations on each matrix are highly cache efficient. Retrieving $f(x)$ amounts to xoring a selection of $O(\log n)$ consecutive r-bit entries of a table which can be done in time $O(r \log n/w) = O(r)$. Also retrieval is cache efficient when the buckets fit into cache lines.

8.10.4.3 Ribbon Retrieval

The linear algebra approach can be further improved by using matrices that are more sparse and more structured. A very effective approach was introduced by

Dietzfelbinger and Walzer [97] and can be viewed as an extreme form of the coupled scheme from section 8.10.4.1 [325]. Each element is mapped to a random window of width $L = \Omega(\log(n)/\varepsilon)$ among $m = (1+\varepsilon)n$ columns of the constraint matrix. Within this window, nonzeroes are choosen using fair coin tosses. During construction, one can then reorder the rows of the matrix to obtain a constraint matrix with a narrow band of nonzeroes around the diagonal. Die leads to a retrieval data structure with construction time $O\big(n\log n/(w\varepsilon^2)\big)$ and query time $O(\log(n)/(w\varepsilon))$. For large n and small ε too slow for practical use. One but one can mitigate this problem by using the partitioning approach discussed above. Using buckets of size polylogarithmic in n, one gets construction time $O\big(n\log\log n/(w\varepsilon^2)\big)$ and query time $O(\log\log n/(w\varepsilon))$. The experiments below indicate that this makes the approach practical for space overhead around 10 %.

In the following we will further develop this approach into a methods allowing space overhead below 1 % [**?**].

Ribbon Solving. We first introduce the faster and more flexible *Ribbon*[12] approach to solve the linear system. This approach incrementally builds a matrix in *row-echelon* form that allows solving the original system by backsubstitution. Figure 8.9 gives pseudocode. The arrays *placed* and *rhs* represent the incrementally built row-echelon form. The main for-loop inserts row i into the row-echelon matrix. This form requires that the first nonzero of row i must be the only nonzero in column j. The process to achieve that is similar to insertion into a linear probing hash table. Let j denote the first nonzero in row i. If column j is still unoccupied, row i is assigned to column j. Otherwise, a row operation with the row placed at column j cancels the nonzero at a_{ij}. This process is repeated until a place for row i is found or a_i contains only zeroes which indicates that a_i is linearly dependent of rows that are already placed.

Bumped Ribbon Retrieval. We now describe a simple and efficient retrieval data structure that combines the ribbon retrieval approach from the previous section with the idea of bumbed retrieval from section 8.10.2.2, [226].

A problem with ribbon retrieval is that we must provide enough space and ribbon width so that no linear dependencies arise in the solution process. This problem can be mitigated by using many buckets with individually chosen hash functions for each bucket. However, this introduces additional complications and introduces per-bucket space overheads. Overall, it becomes difficult to achieve both high space efficiency and high performance within this difficult tradeoff.

[12] **R**apid **I**ncremental **B**oolean **B**anding **ON** the fly

```
Function ribbonSolve(A, F,   var x = 0^m) :
    placed = ⟨0,...,0⟩ : Array 1..m of {0,1}^w   // REM row starting at column j
    rhs = ⟨0^r,...,0^r⟩ : Array 1..m of {0,1}^r   // corresponding right-hand sides
    for i := 1 to n do                                   // Place row a_i.
        loop                                             // placement loop
            if a_i = 0^m then                            // linear dependence found
                if rhs_j = 0 then next iteration of for-loop   // redundant equation
                else return "failed after i − 1 rows"          // here, BuRR will bump
            j := min{ℓ : a_iℓ = 1}                       // first nonzero in row i
            if placed_j = 0 then exit loop               // position available
            (a_i, f_i) ⊕= (placed_j, rhs_j)   // row operation cancels first nonzero in R
        (placed_j, rhs_j) := (a_i, f_i)                  // assign row i to column j
    for j := m downto 1 do                               // backsubstitution
        if placed_j ≠ 0 then x_j := (x · placed_j) ⊕ rhs_j
```

Figure 8.9: Solving a banded $n \times m$ system $Ax = F$ using the ribbon solving approach. Let a_i denote row i of the 0/1-matrix A which can be represented as an offset j_0 and an w bit integer indicating possible nonzeroes at positions $j_0..j_0+w-1$. Operator '$\cdot$' is the scalar product here that indicates the r-bit xors of those entries of x with nonzeroes indicated by $placed_j$. Failures can also be avoided by backtracking a number of recently inserted rows.

The approach we use now avoids hard partitioning into buckets[13] and allows for both high performance and high space efficiency by eliminating linear dependencies by bumping some elements to the next layer of the data structure. The main design task is now to design a bumping scheme that, on the one hand avoids unnecessarily bumping a large number of elements (rows), and, on the other hand does not need too much space to represent the bumping information.

A crucial observation enabling efficient bumping is that most linear dependencies in ribbon retrieval arise from *overloaded ranges* of columns. In other words, for sufficiently large L, the particular L-bit pattern chosen for a row matters much less than *where* this pattern is located. We can view the bumping problem as the problem of eliminating overloaded ranges. Moreover, small overloaded ranges are unlikely. Ranges smaller than L are impossible. The number of rows having all their nonzeroes within a range of width $R \geq L$ is approximately Poisson distributed with parameter

$$\lambda = n\frac{R-L+1}{m-L} \approx \frac{R-L+1}{1+\varepsilon}.$$

From this, one can infer, that overloaded ranges of size $o(L^2)$ are rare.

This motivates our general approach to bumping. We divide the columns into "virtual" buckets of size $B = \mathrm{O}(L^2/\log L)$. We now say that a row is allocated to a bucket if its range of nonzeroes starts within that bucket. We store metadata for each bucket that allows us to bump some rows allocated to that bucket.

There is a large design space for how exactly to perform bumping. We describe one simple and useful approach. We first sort the rows i by the bucket containing their starting position $j(i)$. Within each bucket b, we place the rows *backwards* by decreasing starting position. Let $j_b \in 0..B$ denote the rightmost column within the bucket for which placement fails ($j_b = 0$ if insertion does not fail). Then we bump[14] all rows allocated to b which start at a column $j \leq j_b$. We store j_b as metadata. At query time, if we try to retrieve an element e whose row is allocated to column j of the matrix, we check whether $j \leq j_b$ to test whether e is bumped.

We can further compress j_b by rounding it up to one of a small number of thresholds. Both experiments and analysis [**?**] indicate that even the three pos-

[13] (Quite large) buckets (or *shards*) are useful for parallelization or external construction though.

[14] Note that all rows that are bumped in a later query should also be bumped by the construction. For example, if several rows start at the same column and only the last of them triggers a linear dependency, then all these rows have to be removed from the equation system. This is even more important with the compression schemes described below that rounds j_b upwards.

sible thresholds 0 (bump nothing), t, and B (bump all) suffice when t is chosen appropriately. Our actual implementation use two slight generalization: Either we store 2 bits of metadata per bucket allowing 4 different threshold values. Or we (mostly) store a single bit deciding between the threshold values 0 and t while (rarely needed) threshold $> t$ are stored in a hash table of exceptions.

The tuning parameter $\varepsilon = 1 - m/n$ is also important for space efficiency. In particular, *negative* values are a good choice because they have the potential to radically reduce the fraction of empty slots. For small B, we can make this fraction arbitrarily small.

Baustelle

8.11 Approximate Dictionaries (AMQs)

We now consider hash tables that support the operations *insert* and *contains*. Furthermore, the operation *contains* is allowed to give *false positive* answers, i.e., it may claim that an object is contained in the table although this is not true. Such a wrong answer will only happen with a certain probability – the *false-positive rate* ϕ. This data structure is known as an *approximate membership query data structure (AMQ)*. AMQs are also commonly known as *Bloom filters* which are one particular implementation of AMQs (see section 8.11.1).

Applications. Many applications can tolerate this kind of error. These fall in two categories: The most frequent case is that a positive query triggers an expensive operation that will anyway check whether the response was correct. For example, the AMQ A may approximate a set of data objects stored on disk (or any storage medium that is slower than the one used for A). When the user requests an object, A is queried first. A negative response is definitive and avoids checking the disk. A positive response triggers a disk access anyway and will reveal whether the object is actually available. Since the false-positive rate is small, disk access will happen almost exclusively when necessary. A similar scenario applies when the objects are stored on multiple computers connected by a network where AMQs help so avoid communication.

An example that does not involve memory hierarchies or communication could be a data base relation with a text attribute a. If one wants to support fast searching for words in a, one can additionally store an AMQ A_a with each of these texts that approximates the set of words contained in the text. When searching for a word w in a row r of this relation, one first computes $r.A_a.contains(w)$. If the result is false, $r.a$ does not contain w. Otherwise, one actually has to search $r.a$.

Note that in this case, we are likely to have a large number of small AMQs (one for each row of the relation).[15]

The second class of applications arises when a false positive reply can be tolerated without repairing it. For example, the Medium online publishing platform uses AMQs to avoid showing search results on articles which the user has already read [306]. The consequence of a false positive result then means that a small percentage of potential search results are omitted although they have not yet been read. Compared to other uncertainties in the search process, this source of errors can be made diminishingly small.

The false-positive rate ϕ is connected with the number of bits per object we are willing to invest. The simple and widely used *Bloom filters* we discuss in section 8.11.1 need about $\log(1/\phi)/\ln 2$ bits of space per object. *Blocked Bloom filters* (section 8.11.2) are a more cache-efficient variant. An information theoretical lower bound for the space consumption is $\log(1/\phi)$ bits [71]. In section 8.11.3 we approach this bound up to an additive constant by combining data compression with an extreme variant of Bloom filters (using a single hash function). At the price of restricting ourselves to static AMQs (no insertions), section 8.11.4 gets aribitrarily close to the lower bound.

In section 8.11.6 we explain parallel AMQs while in section 8.11.7 we outline various generalizations that support additional functionality like deletion, counting, or size adaptation. Many more variants of AMQs have been considered. For this we refer to a number of survey papers [58, 308, 129, 195]. Also note that some applications like networking use hardware implementations.

8.11.1 Bloom Filters

A Bloom filter [48] of size m consists of an m-bit bit array b and uses k hash functions $h_1,\ldots, h_k$ with range $\mathbb{Z}_m$. To insert an element e, the bits $b[h_1(e)],\ldots, b[h_k(e)]$ are set. Correspondingly, a *contains*-query checks the presence of these bits. A false positive can occur when these checked bits stem from one or several other inserted objects.

The false-positive rate is approximately [221, pages 107–112]

$$\phi \approx \left(1 - e^{-kn/m}\right)^k \tag{8.5}$$

[15] Even faster text search can be done with appropriate index data structures like inverted indices. However, the described solution will be useful if the searches involve other criteria that may be more selective than the text search. For example, one might look for all rows entered in a narrow range of dates that also contain w.

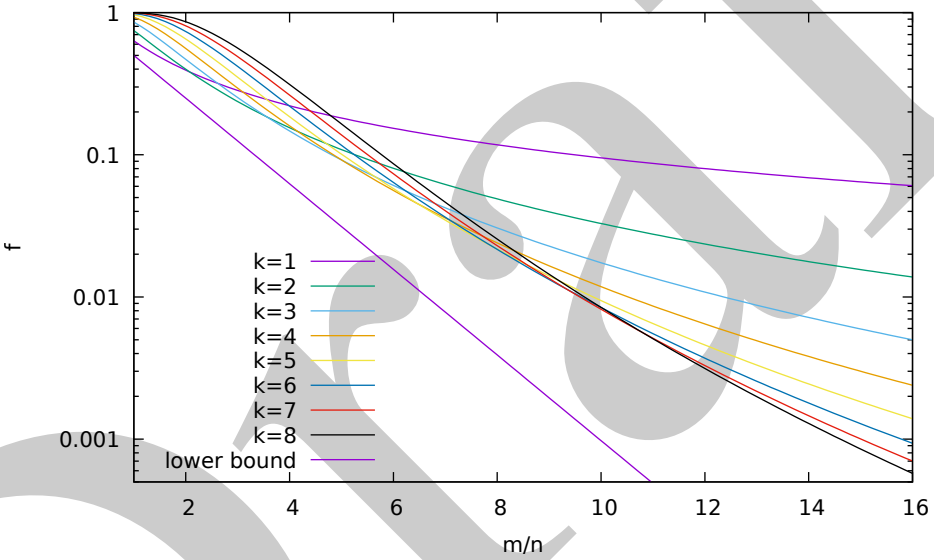

Figure 8.10: Approximate false-positive rate ϕ according to eq. (8.5) for an m-bit Bloom filter with n stored elements for $k \in 1..8$ hash functions. The line for the lower bound gives the value one would obtain for an information-theoretically optimal data structure.

where n is the number of stored elements. Figure 8.10 plots these values for different k as a function of m/n, the number of bits available per inserted element. Optimizing k based on eq. (8.5) yields $k = m/n \cdot \ln 2$ (which then has to be rounded to an integer). The effect of this choice is the intuitive result that a Bloom filter yields the lowest false-positive rate if about half of its bits are set, i.e., if it is in a state of maximum information content. An interesting consequence is that a *contains*-query with negative outcome has to probe only two bits in expectation. Insertions and positive queries have cost growing linearly with k. Although we can obtain multiple hash values cheaply (section 8.1.3), the k memory accesses and possible cache faults will have a significant cost. Therefore, one often chooses a smaller value of k in order to improve running time. Note that this often incurs only a small penalty in the form of increased false-positive rate. For example, at $m/n = 8$, the optimal k is 6 ($\phi \approx 2.16$ %) but $k = 5$ ($\phi \approx 2.17$ %) or $k = 4$ ($\phi \approx 2.4$ %) are very close. Even $k = 3$ ($\phi \approx 3.1$ %) might be a reasonable choice.

8.11.1.1 Union and Size Estimation

Given Bloom filters b_A and b_B representing two sets A and B respectively, we can compute the Bloom filter representing the union $A \cup B$ as the bit-wise-or $b_A | b_B$ of b_A and b_B. Note that this can be done in time $O(m/w) = O(m/\log m)$ using the bit-parallelism inherent in our standard machine model with word size w (see section 2.2.1). The union operation is also easy to parallelize with multiple processors.

We can also estimate the size of a set A represented by a Bloom filter b_A as $n(b_A) = -\frac{m}{k} \ln\left(1 - \frac{a}{m}\right)$ where a is the number of set bits in b_A [305]. Once more, this is possible in time $O(m/\log m)$ using bit-parallelism – in this case employing the population-count instruction available in most microprocessors.[16] Although we cannot compute the Bloom filter of the intersection $A \cap B$ from the Bloom filters b_A and b_B, we can estimate the size of $A \cap B$ as $n(b_A) + n(b_B) - n(b_A | b_B)$ [305].

Open Problem 49 (Fast union and size estimation of Bloom filters)
Although the above bit-parallel operations are rather straightforward algorithmically, it seems that current algorithm libraries do not support them efficiently. In particular, an implementation using the best available SIMD instructions (such as VPOPCNT in AVX-512) and multicore parallelism could be useful. Also one

[16] The population-count operation can also be implemented using lookup tables.

might want to adapt the the size estimators to the blocked Bloom filters in section 8.11.2 or to the space optimal variants in section 8.11.3. A related theoretical issue might be do derive error margins for these estimators.

8.11.2 Blocked Bloom Filters

A major performance bottleneck of large Bloom filters is that accessing them incurs up to k cache misses for insertions and positive queries. This can be reduced to one cache miss by splitting the Bloom filter into its individual cache blocks and setting all k bits for an element in the same cache block [254]. Thus, inserting an elements amounts to first choosing a block b based on one hash function (or from some bits of a larger hash value) and then to set k bits in block b based on further hash function values. Often, the blocks will be entire cache lines. Currently, for x86 processors, this implies a block size of $B = 512$.[17]

The downside of these *blocked Bloom filters* is that they have a somewhat larger false-positive rate than plain Bloom filters. Roughly, the reason is that fluctuations in the number of elements allocated to a block imply that some blocks get significantly more 1-bits than others. Increasing k amplifies this effect because one additional element already means k additional bits in one block. Indeed, for $k = 1$, blocked Bloom filters and plain Bloom filters are equivalent.

Our paper [254] derives an approximate value for the false-positive rate as

$$\phi \approx \sum_{i=0}^{\infty} \frac{\left(\frac{m}{n}\right)^i e^{-\frac{m}{n}}}{i!} \cdot \left(1 - e^{-\frac{ki}{B}}\right)^k. \tag{8.6}$$

and experimentally validates that this approximation is close to simulation results. This equation sums over the number of elements allocated to a bucket. The left factor in the summand is the likelihood of this event as approximated by the Poisson distribution while the right factor is the approximation of the false positive rate for that bucket as given by eq. (8.5). For fixed k, this infinite sum can be evaluated into a closed-form expression, e.g., using a computer algebra system. For example, fig. 8.11 plots this sum for $k = 5$ and relevant values of m/n. In comparison to plain Bloom filters with $k = 5$, the false-positive rate ϕ is slightly larger. However, it is smaller than that rate for plain Bloom filters with $k = 4$.

[17] One could also consider blocks consisting of two cache lines since hardware prefetching algorithms of many processors imply that the cost of accessing two consecutive blocks can be considerably cheaper than twice the time for fetching one block.

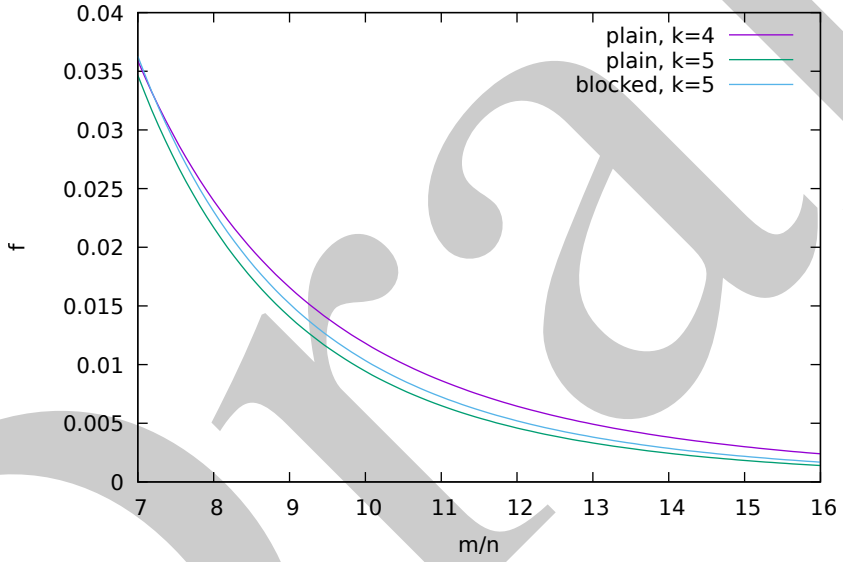

Figure 8.11: Approximate false-positive rate ϕ according to eq. (8.5) (plain) and eq. (8.6) (blocked with block size $B = 512$) for an m-bit plain/blocked Bloom filter with n stored elements for $k = 5$ hash functions.

Note that the measure to decrease k below the optimal value is often taken in order to improve performance. We can see that it may be more effective to switch to blocked Bloom filters instead.

Open Problem 50 (Refining blocked Bloom filters) There are a number of interesting loose ends in blocked Bloom filters. For one, in order to reduce the cost for computing hash functions, one would like to use the techniques from section 8.1.3. However, no theoretical guarantees are known for the linear congruential technique and the analysis of the linear combination techniques from [178] is only valid for $B \to \infty$.

One can also take up the *pattern refinement* [254]: Rather than setting k bits by computing k individual positions, just compute the bitwise or of the block with a bit pattern retrieved from a lookup table. This might be further improved by choosing the table entries carefully. Does it help if we ensure that each position in a block occurs equally frequently in the table? Does it help if we ensure that never two table entries contain all 1-bits of another table entry (or, when this is impossible to minimize the number of pairs with this property)? Does it help to use SIMD-instructions for the implementation? ■

The paper [254] describes further refinements that are useful when m/n and k are large. For example, it might then make sense to set k bits overall in $k' < k$ different blocks.

8.11.3 Compressed Single-Shot Bloom Filters (sBFs)

The Bloom filters we have seen need a factor about $\log e \approx 1.44$ more space than the information theoretic lower bound of about $\log 1/\phi$ bits [71]. We can see that in the growing gap in fig. 8.10. There has been considerable work on closing this gap (e.g., [71, 217, 240, 254, 277, 108]). Here we outline several variants of one possible approach and discuss their practicality. Section 8.11.4 discusses another approach that needs even less space but lacks some features like dynamic insertions, union, and size-estimation of the intersection.

Somewhat surprisingly, the starting point for several of these approaches can be viewed as using the *least* space-efficient variant of Bloom filters that sets only a single random bit in an m-bit array. Such *single-shot Bloom filters (sBFs)* have false positive probability $\phi = m/n$ when storing n bits. This paradox can be resolved by observing that there are plenty of techniques to *compress* sparse bit vectors in an information-theoretically near optimal way. There are $\binom{m}{n}$ possible

exactly implement single-shot Bloom filters (sBFs). A simple and fast alternative is to make the blocks self-sufficient – they store an Elias–Fano Filter that makes the best use of the available space. Depending on how many elements are allocated to a block, the number of bits used to represent each element is adapted. In the extreme case, the block degenerates to an uncompressed sBFs with B bits. Thus the false positive rate varies from block to block. This situation is similar to the blocked Bloom filter described in section 8.11.2 and overall will deteriorate the false positive rate. What we gain is simplicity and speed due to saved indirections. Since there is no space overhead for references to overflow blocks and garbage collection, the overall space efficiency may also be quite good for carefully chosen configurations. Implementing and analyzing this approach seems a worthwile project.

8.11.4 Signature Based AMQs (SAMQs)

Another view at AMQs is to hash objects to short bitstrings, i.e., *signatures*. A membership query for key k entails comparing its signature $s(k)$ with a set of candidate signatures representing objects in the AMQ data structure. If any of these signatures match $s(k)$ then key k is reported to be a member of the represented set. The false positive rate ϕ depends on the number of candidate signatures and the length r of the signatures. Ideally there is only a single candidate and we get $\phi = 2^{-r}$. Quotient filters and Elias–Fano filters can be viewed as SAMQs with a single candidate. Linear probing filters (see section 8.11.3.1) have a number of candidates depending on the fill-degree of the table.

Even more interesting SAMQs are not directly related to single-shot Bloom filters (sBFs).

Cuckoo filters [108, 55] adopt the idea of cuckoo hashing (see section 8.5) to map signatures to one of H buckets of size B. Cuckoo filters can be filled to a higher degree than quotient filters. On the other hand, they have increased false positive rate because HB candidate signatures have to be checked.

Retrieval filters. Space consumption close to the lower bound of $n \log 1/\phi$ bits can be achieved by storing a function mapping elements to signatures using one of the retrieval data structures described in section 8.10.

8.12 Perfect Hash Functions (PHFs))

Much of what is discussed in this chapter is about handling collisions between hash function values. One appealing approach is to look for hash functions that do not produce collisions at all. We call $h : S = \{s_1, \ldots, s_n\} \to \mathbb{Z}_m$ *perfect hash function (PHF)* if h is injective. We call it *minimal perfect* (MPHF) if, in addition, $m = n$. This is possible and practical when S is known in advance. Then h can be specifically designed for S.

In section 8.12.1 we will first establish a lower bound of $n\left(\log e - \varepsilon \log \frac{\varepsilon}{1+\varepsilon}\right)$ bits to represent a PHF. We will also show that this bound is tight by giving a brute-force construction algorithm that achieves this bound (using exponential construction time). For a long time, it has been considered unrealistic to find a practical algorithm with similar space efficiency. However, a long sequence of both theoretical and practical results has changed this view dramatically. Of course there is still a space-time tradeoff, but generally it can be said that perfect hash functions can be constructed in linear time with moderate constant factors so that they consume only a few bits per element and can be evaluated in constant time. After mentioning some applications, we first discuss a technique for converting PHFs to MPHFs and then outline several concrete approaches to constructing PHFs with decreasing amount of required space, increasing level of sophistication, and increasing cost for construction and retrieval. Section 8.12.2 modifies the FiRe approach for retrieval from section 8.10.2.1 to obtain MPHFs. FiPHa is simple and fast but requires about 4 bits per element to be fast. Section 8.12.3 combines cuckoo hashing (section 8.5) with a retrieval data structures to obtain PHFs. For example, one can use a 2-bit retrieval data structure to obtain PHFs with range $\mathbb{Z}_{1.024n}$. Sections 8.12.4 and 8.12.5 describe methods that approach the information theoretic bound on space consumption by breaking the problem of building a PHF into small subproblems for which a brute-force approach becomes feasible.

Applications. PHFs can be used to assign short, unique names to objects (e.g. to identify them in a database). They can also be used to address a table of associated values, see, i.e., they can implement an updateable retrieval data structure (see section 8.10.3).

Converting PHFs to MPHFs. Suppose h is a PHF with range $\mathbb{Z}_m$ where $m = (1+\varepsilon)n$. Then we can compute a bitvector $b[0..m-1]$ such that $b[i] = 1$ if and only if $h(e) = i$ for some element $e \in S$. Then $h'(x) = b.rank(h(x))$ is an MPHF. Using the techniques from section 7.4, b can be represented with $m + o(m)$ bits such that *rank* works in constant time. For small ε we might employ a com-

pressed reprentation of b that reduces the space overhead to $O(n\varepsilon \log 1/\varepsilon)$ bits. For example using Elias–Fano coding one would need $\varepsilon n(2+\lceil \log(1/\varepsilon)\rceil)$ bits.

8.12.1 A Tight Lower Bound

There is a lower bound of

$$n\left(\log e - \varepsilon \log \frac{\varepsilon}{1+\varepsilon}\right)$$

bits for representing a PHF with a range of size $m = (1+\varepsilon)n$. This bound holds as the size u of the universe S is drawn from goes to infinity. We obtained this bound taking the limit of a bound for general u [36, 210].

A Brute Force Upper Bound. The above lower bound is also an upper bound and there is a surprisingly simple proof. Consider a sequence $h_1, h_2, \ldots$ of random hash functions $S \to \mathbb{Z}_m$ and an algorithm that tries these functions one after the other, until h_i turns out to be perfect. Then it suffices to store i to represent a PHF. A random hash function has probability

$$p := \text{prob(success)} = \frac{n!\binom{m}{n}}{m^n}$$

to be perfect. Hence, the stored index i has a geometric distribution with parameter p. The entropy of this distribution is

$$\frac{-p\log p - (1-p)\log(1-p)}{p} = \log\frac{1}{p} + \log e + O(p)$$

where the simplification uses the Taylor series of the logarithm function. Now setting $m = (1+\varepsilon)n$ we can approximate

$$\begin{aligned}
\log\frac{1}{p} &\approx n\log m - n\log\frac{n}{e} - n\log\frac{m}{n} - (m-n)\log\frac{m}{m-n} \\
&= n\left(\log m - \log n + \log e - \log m + \log n - \varepsilon\log\frac{(1+\varepsilon)n}{\varepsilon n}\right) \\
&= n\left(\log e - \varepsilon\log\frac{1+\varepsilon}{\varepsilon}\right)
\end{aligned}$$

using $m - n = \varepsilon n$ and the approximations $n! \sim n\ln\frac{n}{e}$ (eq. (B.1)) and $\log\binom{m}{n} \sim n\log\frac{m}{n} + (m-n)\log\frac{m}{n-k}$ when $m = \Theta(n)$ (eq. (B.3)). This is not a practical algorithm since, in expectation, it needs an exponential number of trials. However, it will turn out to be a basis for practical algorithms that apply brute-force techniques to small subsets of the input.

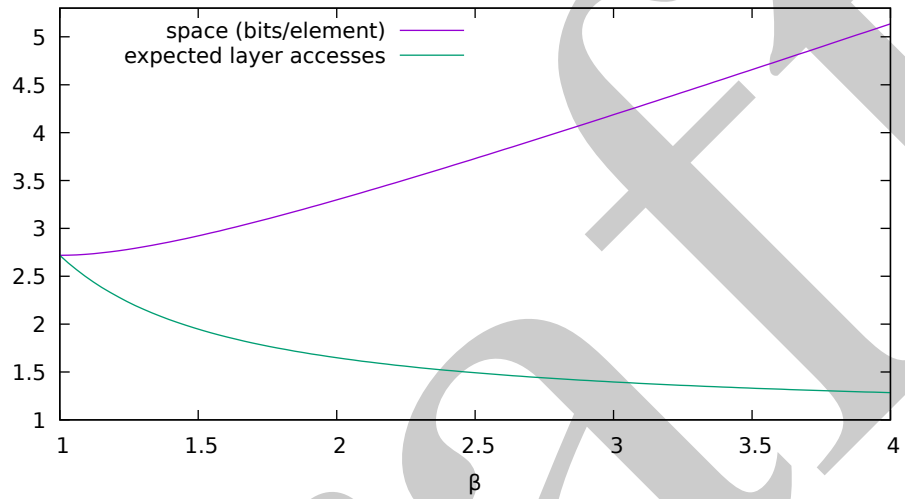

Figure 8.13: Space time tradoff of FiPha ignoring the $o(n)$ term for supporting fast rank operations.

8.12.2 Filtering out Collisions (FiPHa)

The FiRe approach for retrieval described in section 8.10.2.1 can be adapted to compute PHFs. We describe a variant *FiPHa* that computes a MPHF with little additional overhead (see also [226]). The approach uses a bitvector b with $m = \beta n$ bits for some tuning parameter β. Elements are mapped to a random bit in b using a hash function h_b. Bit $b[i]$ is set to one if and only if exactly one element e of S is mapped to position i, i.e., when e is *noncolliding*. The perfect hash function h maps noncolliding elements to $b.rank(h_b(e))$. Colliding elements are bumped to the next layer of the data structure whose hash function value is offset by the total number of noncolliding elements.

FiPHa is easier to configure and analyze than FiRe since we do not need to bother with the number of empty slots. Indeed, β is the only important tuning parameter. The probability that an element is not colliding with any other elements is

$$p_1 := \left(1 - \frac{1}{\beta n}\right)^{n-1} \approx e^{-1/\beta}.$$

Thus, the expected number of accessed layers is $\ell = 1/p_1 = e^{1/\beta}$. The expected space overhead per element is $s = \beta/p_1 = \beta e^{1/\beta}$. This value is minimized for $\beta = 1$. In this case, both ℓ and s take the value e. Since ℓ decreases with β, only values $\beta \geq 1$ make sense. Figure 8.13 plots this tradoff. We see that already for s around 4, ℓ approaches one – leading to a good space-time tradoff for many applications.

Supporting fast rank operations incurs an additional small overhead factor. We discusss a few cases for cache line sizes of 512 bits: For $n \leq 2^{32}$ we can store a 32-bit offset within each cache line and obtain an additional space overhead of $32/(512-32) \approx 7$ %. By using the two-level approach from section 7.4 we can store rank information in separate cache lines – a 64-bit global offset and 31 local offsets with 14 bits each within one cache lines addressing blocks of 2^{14} bits yield a space overehead of $512/2^{14} = 2^{-5} \approx 3$ %. With somewhat higher space overhead we can also reduce the required subblock sizes from 512 to 64 or 128 to speed up the local computations.

8.12.3 Using Retrieval and Cuckoo Hashing

We describe a generic version of a well known approach (e.g., [201, 54, 325] and then show one way to instantiate it with state-of-the-art components.

Consider H hash function $h_1, \ldots, h_H$ with range $\mathbb{Z}_m$. The idea is to select one of these hash functions $i(s)$ for each element $s \in S$ such that $h(s) = h_{i(s)}(s)$ is a perfect hash function. The indices $i(s)$ can be stored in a $\log H$ bit retrieval data structure (see section 8.10).

Of course, the difficulty is to find i. This problem is closely related to the problem of building a cuckoo hash table for S with m entries with H hash functions and block size $B = 1$; see section 8.5 [116, 177]. Suppose we have built this hash table. Rather than storing it, we just remember where each element was stored – when element s is stored at position $h_i(s)$ using the cuckoo hash table, we store i for s in the retrieval data structure. Recall, that such a cuckoo hash table can be build in linear expected time. In the literature, this has also been discussed as the equivalent problem of finding an *edge orientation* in a hypergraph. When using coupled hash functions, one can also use hypergraph peeling (section 8.10.4.1) [325].

A particularly useful choice of parameters are $H = 4$ where it suffices to store 2 bits in the retrieval data structure and it suffices to choose $m > 1.024n$. Using the highly space efficient retrieval data structure from **??**, we get perfect hash

functions with close to 2 bits per element. Using an additional bit-vector data structure allows us to convert this to a MPHF using space just over 3 bits per element. Using compression, e.g., using Elias–Fano coding one gets down to about 2.2 bits per element.

For $H = 2$ we can use a 1 bit retrieval data structure to obtain a PHF with $m > 2n$ using close to 1 bit of space per element. Adding a rank-select data structure of size $2n$ bit then yields a MPHF using about 3 bits per element.

(author?) [325] describes a variant with $H = 3$ that integrates a rank-select data structure – retrieved values 0–2 indicate hash function and value 3 indicates an empty cell. This yields an MPHF with about 2.18 bits per element. This approach is also partitularly fast, since a single peeling process computes both the MPHF and the data structure needed to retrieve it.

8.12.4 Hash, Displace, and Compress (CHD)

There is a very simple information theoretically optimal way to find perfect hash function – try random functions h_1, h_2, ...until a function h_k is found that is perfect and store k. This is extremely slow and impractical but can be viewed as one high level idea behind the practical method we now present. The trial-and-error approach can be made practical by building the function piece by piece.

Hash, Displace, and Compress (CHD) [36] first maps a key $s \in S$ to a *bucket* $g(s) \in 1..n/\lambda$ for some constant $\lambda > 1$ using a primary hash function g. Then, $g(s)$ is used to select a secondary hash function from a sequence ϕ_1, ϕ_2,.... The overall PHF is then $h(s) = \phi_{\sigma(g(s))}(s)$ where the function σ needs to be stored in an array that supports variable bitlength encoding [121].

In order to find σ, the buckets are first sorted by decreasing size and (dis)placed in that order. For each bucket B in that order, the functions ϕ_1, ϕ_2,... are tried until there are no collisions between the elements of B or of previously placed buckets. This check can be made in time $|B|$ by keeping an array $T[0..m-1]$ of flags indicating whether any element is already mapped to that position. Using appropriate variable-bitlength encoding, the expected number of bits to encode σ is constant.

Actually, CHD has the potential to approach an information-theoretically optimal representation. Space decreases with growing λ but running time increases exponentially. The concrete configuratations chosen in the experiments need about half a bit more than information theoretically optimal space.

8.12.5 Splitting plus Brute Force (RecSplit)

8.12.6 Experimental Comparison

8.13 Summary

There is a bewildering variety of results on hash tables which however reflect their actual importance in applications. There is hardly a nontrivial application of computers that does not need a dictionary data structure somewhere and more often than not, hash tables are the method of choice to implement them. Of course this does not mean that every programmer needs to become an expert on hash tables. It is usually OK to use existing library implementations. However, it is often not sufficient to just drop in *any* implementation or even the most frequently used standard libraries. When hash tables figure prominently in inner loops of a compute-intensive application or when a profiler indicates that they have significant impact on performance. One should have a closer look.

In our experience, linear probing tables (section 8.4) often perform substantially better than those based on chaining. However, it is important to use them in the right way. Table size m beyond 3 times the input size n is helpful if the space overhead is affordable. On the other hand, even m close to n can work well when positive queries and bulk-operations dominate the operation mix. Cuckoo hashing (section 8.5) is an interesting alternative in space-constrained situations. Nonstandard implementation features like efficient dynamic growing (section 8.6) or fast bulk operations can make a difference in some applications.

The situtation is more complicated for advanced models of computation – parallel, distributed, or external. Here, using an efficient, well configured library is often *not* sufficient to remove performance bottlenecks. Rather, it may be important to engineer the application itself to reduce the impact of expensive operations. For example, approximate dictionaries (section 8.11) may help to avoid external or remote hash table accesses. Instead of directly using a distributed or concurrent hash table (sections 8.8 and 8.9), it may be better to work with local tables as long as possible and to later combine the local results using bulk operations. The highly compressed but more specialized data structures for retrieval (section 8.10), approximate dictionaries (section 8.11) and perfect hashing (section 8.12) are a way to squeeze as much data as possible in some fast/local memory.

The hash functions themselves (section 8.1) are often taken for granted. This is justified to the extent that good library implementations seem to work well in a huge spectrum of applications. However, here the gaps between theory and

practice are perhaps most critical since the probabilitstic performance guarantees available for hash tables all hinge on hash function families that are actually rarely used in practice. Profiling for some concrete inputs cannot mend the resulting loss in robustness for the applications since bad inputs may be just around the corner. Thus, here one has to be careful with profiling driven performance engineering. Perhaps the right compromise is to use the highest quality hash functions available that do not severely impact performance – even when faster, low quality hash functions work well for the inputs tried.