Minimal Perfect Hashing

(Minimum) Perfect Hash Functions

We've been using the idea of *hashing* a lot in this lecture.

One class of hash functions that are particularly interesting are Minimum Perfect Hash Functions (MPHF).

S : set of keys

$f : S \rightarrow \{1,2,\ldots|S|\}$ s.t.

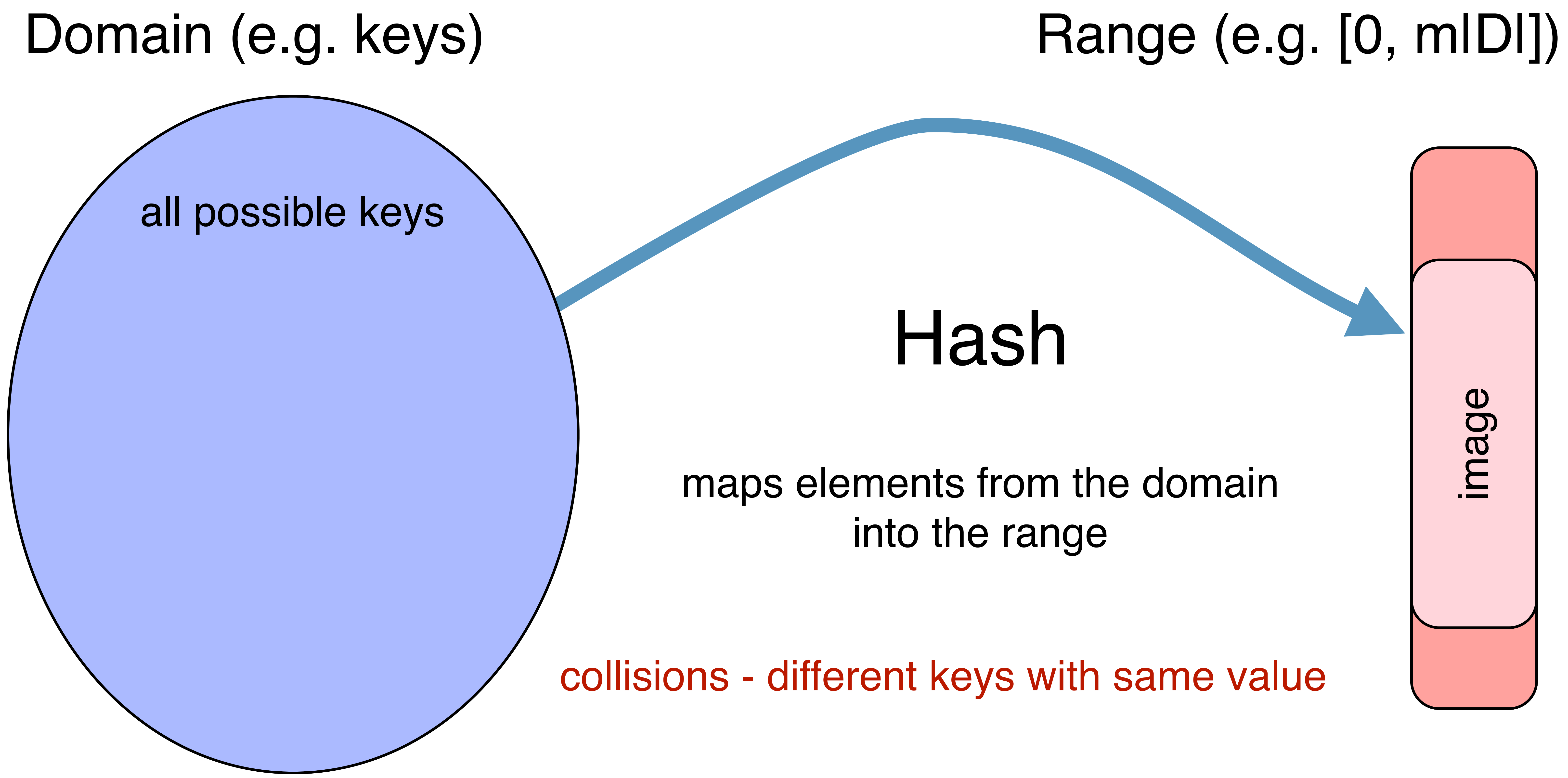
$\forall\, u,v \in S,\ u \neq v$ then $f(u) \in \{1,2,\ldots|S|\}$
and $f(v) \in \{1,2,\ldots|S|\}$ and $f(u) \neq f(v)$

In other words. f is an injective function from S to the integers 1..|S| (or 0..|S|-1) such that every element of S maps to some *distinct* integer.

Note: for $x \notin S$, no property is guaranteed about f(x)

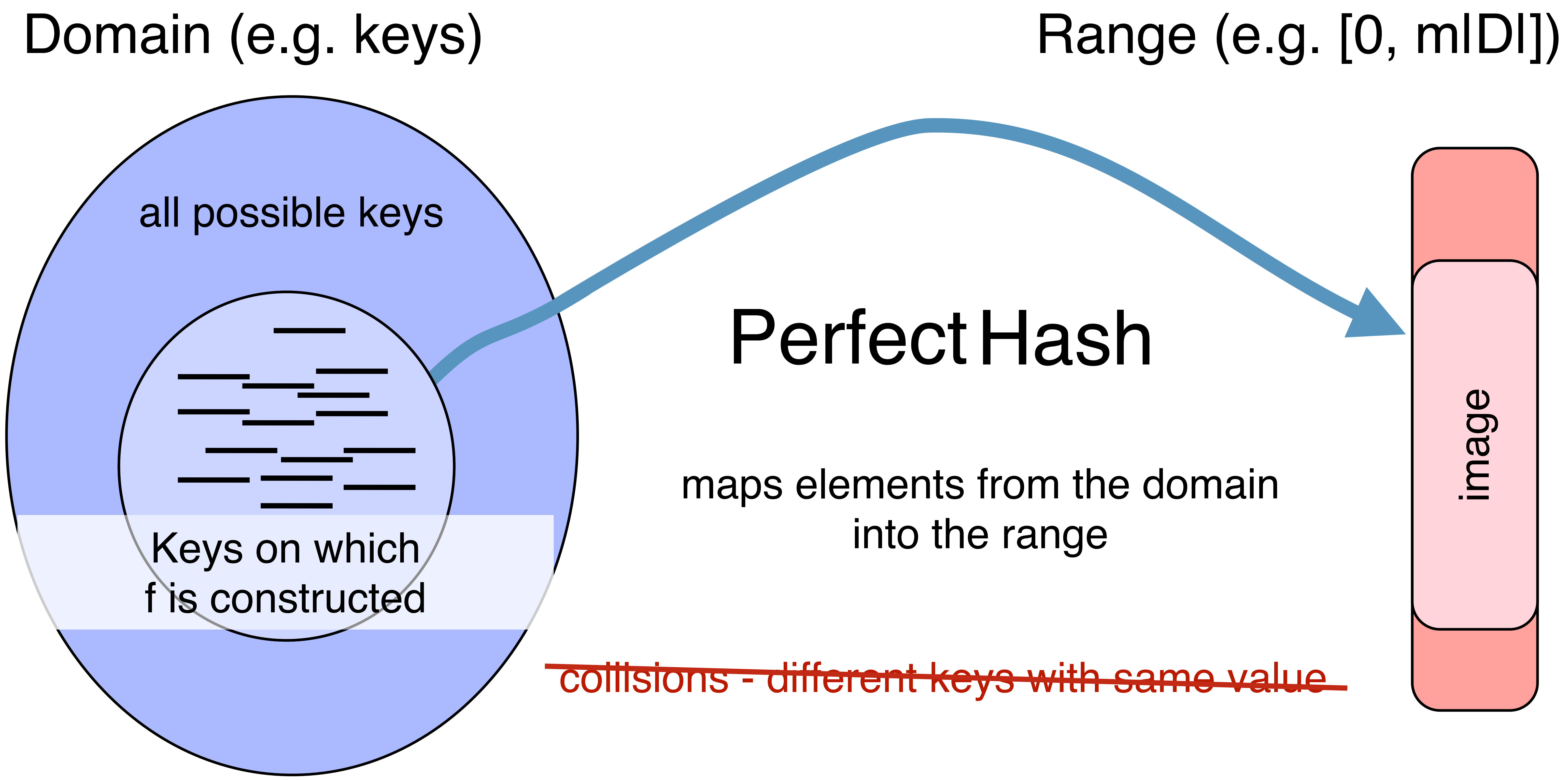
Construction of a Perfect Hash Index

Key: We know meaningful sub-patterns ahead of time


Domain (e.g. keys)
Range (e.g. [0, m|D|])
all possible keys
Hash
maps elements from the domain into the range
image
collisions - different keys with same value

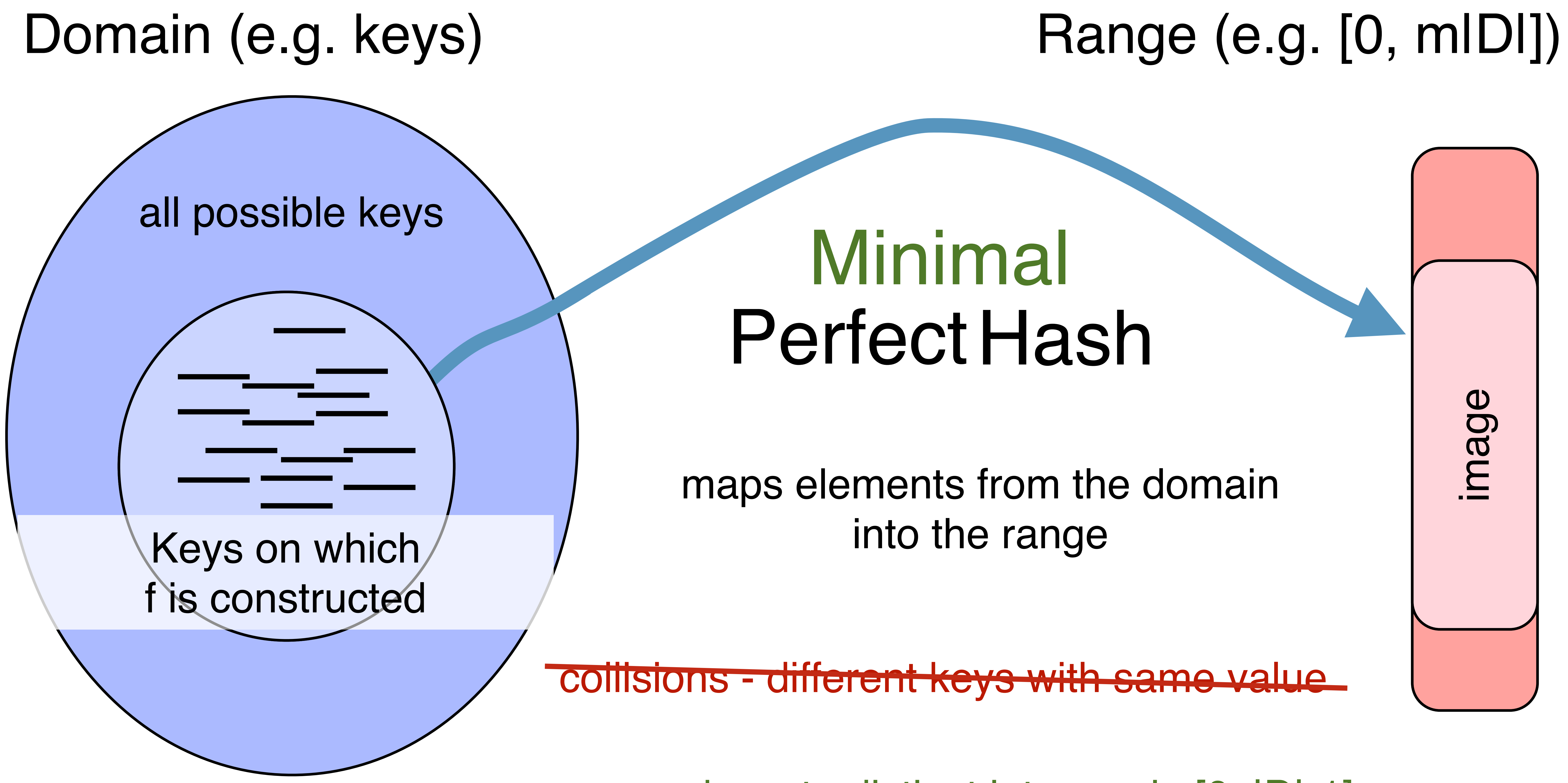
Construction of a Perfect Hash Index

Key: We know meaningful sub-patterns ahead of time


Domain (e.g. keys)
Range (e.g. [0, m|D|])
all possible keys
Keys on which f is constructed
Perfect Hash
maps elements from the domain into the range
~~collisions - different keys with same value~~
image

Construction of a Perfect Hash Index

Key: We know meaningful sub-patterns ahead of time



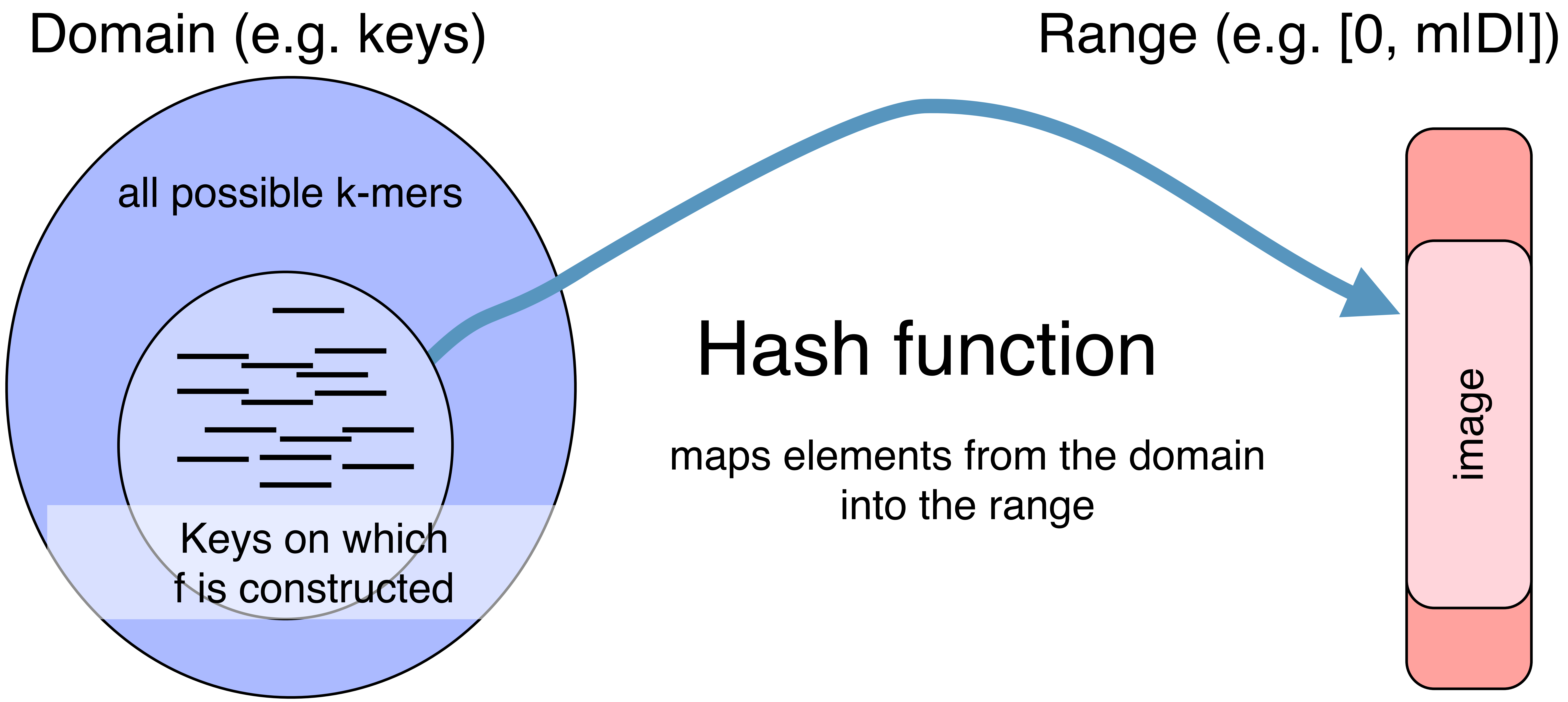
maps elements from the domain into the range

~~collisions - different keys with same value~~

maps keys to distinct integers in [0, |D|-1]

Construction of a Perfect Hash Index

Key: We know meaningful sub-patterns ahead of time



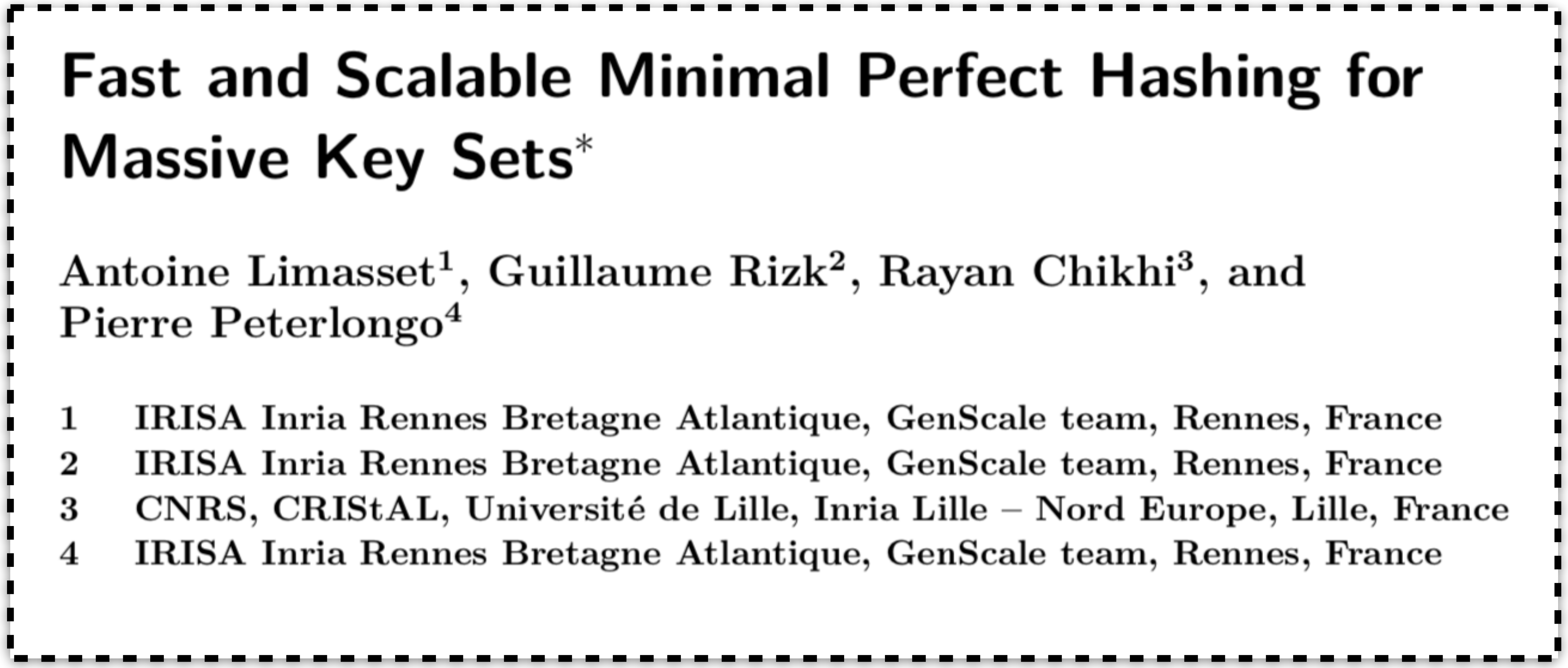
Minimal maps keys to distinct integers in [0, |D|-1]

Perfect no collisions - every key maps to its own value

(Minimum) Perfect Hash Functions

We'll talk about BBhash. My *favorite* algorithm for minimal perfect hash construction. It's *not* the most sophisticated algorithm in the literature, but it is by-far the most practical.

https://github.com/rizkg/BBHash



Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*

Antoine Limasset[1], Guillaume Rizk[2], Rayan Chikhi[3], and Pierre Peterlongo[4]

1 IRISA Inria Rennes Bretagne Atlantique, GenScale team, Rennes, France
2 IRISA Inria Rennes Bretagne Atlantique, GenScale team, Rennes, France
3 CNRS, CRIStAL, Université de Lille, Inria Lille – Nord Europe, Lille, France
4 IRISA Inria Rennes Bretagne Atlantique, GenScale team, Rennes, France

Observation: often MPHF are used to map keys (not stored) to values (stored). The set of values is often much larger, per-element than the MPHF. It's worth spending a few more bits per-element on the MPHF if we can construct it efficiently

Minimum Perfect Hash Functions

Theoretical minimum is $\log_2(e)N \approx 1.44N$ bits / key (regardless of # of keys).

Best methods, in practice, provide just under ~3 bits/key.

This approach provides a parameter γ to trade off between construction speed and final MPHF size and query time.

Successive hashing for construction

For a set of keys F_0 construct a bit-array of size $A_0 = |F_0|$.

Insert the keys into A_0 using hash function $h_0()$

$A_0[i] = 1$ if *exactly* one element from F_0 hashes to i

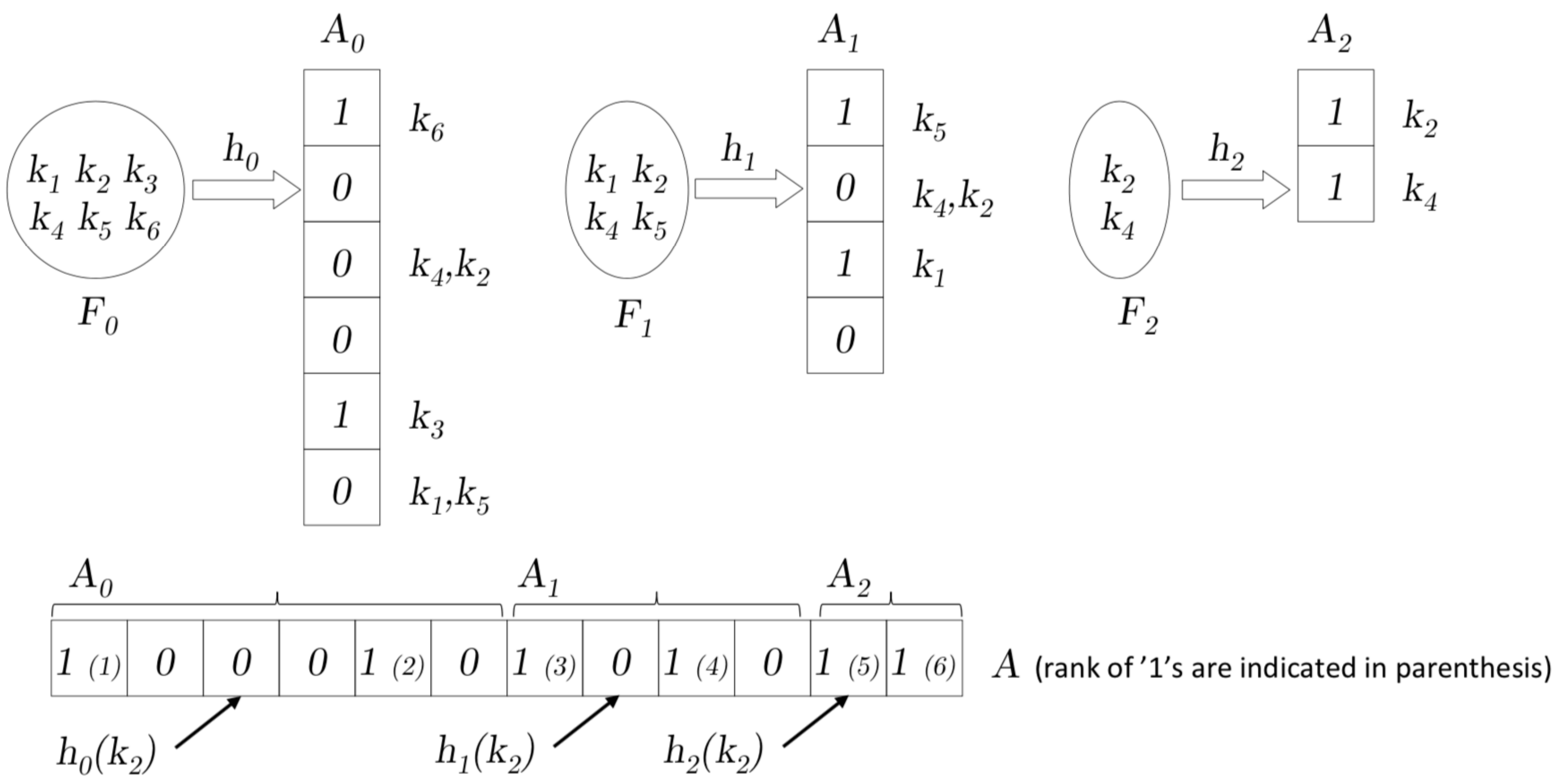
For all keys that collide under $h_0()$, create a new key set F_1 of size $|F_1|$

Create a corresponding new bit vector A_1.

Repeat this process until there are no collisions.

In practice: Repeat this process until F_k is sufficiently small, and use a traditional hash table to store it.

Successive hashing for construction



Detecting Collisions

During construction at each level d, collisions are detected using a temporary bit array C_d of size $|A_d|$. Initially all C_d bits are set to '0'. A bit of $C_d[i]$ is set to '1' if two or more keys from F_d have the same value i given by hash function h_d. Finally, if $C_d[i] = 1$, then $A_d[i] = 0$. Formally:

$$C_d[i] = 1 \Rightarrow A_d[i] = 0;$$
$$(h_d[x] = i \text{ and } A_d[i] = 0 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 1\,(\text{and } C_d[i] = 0)\,;$$
$$(h_d[x] = i \text{ and } A_d[i] = 1 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 0 \text{ and } C_d[i] = 1.$$

Querying & Minimality

A query of a key x is performed by finding the smallest d such that $A_d[h_d(x)] = 1$. The (non minimal) hash value of x is then $(\sum_{i<d} |F_i|) + h_d(x)$.

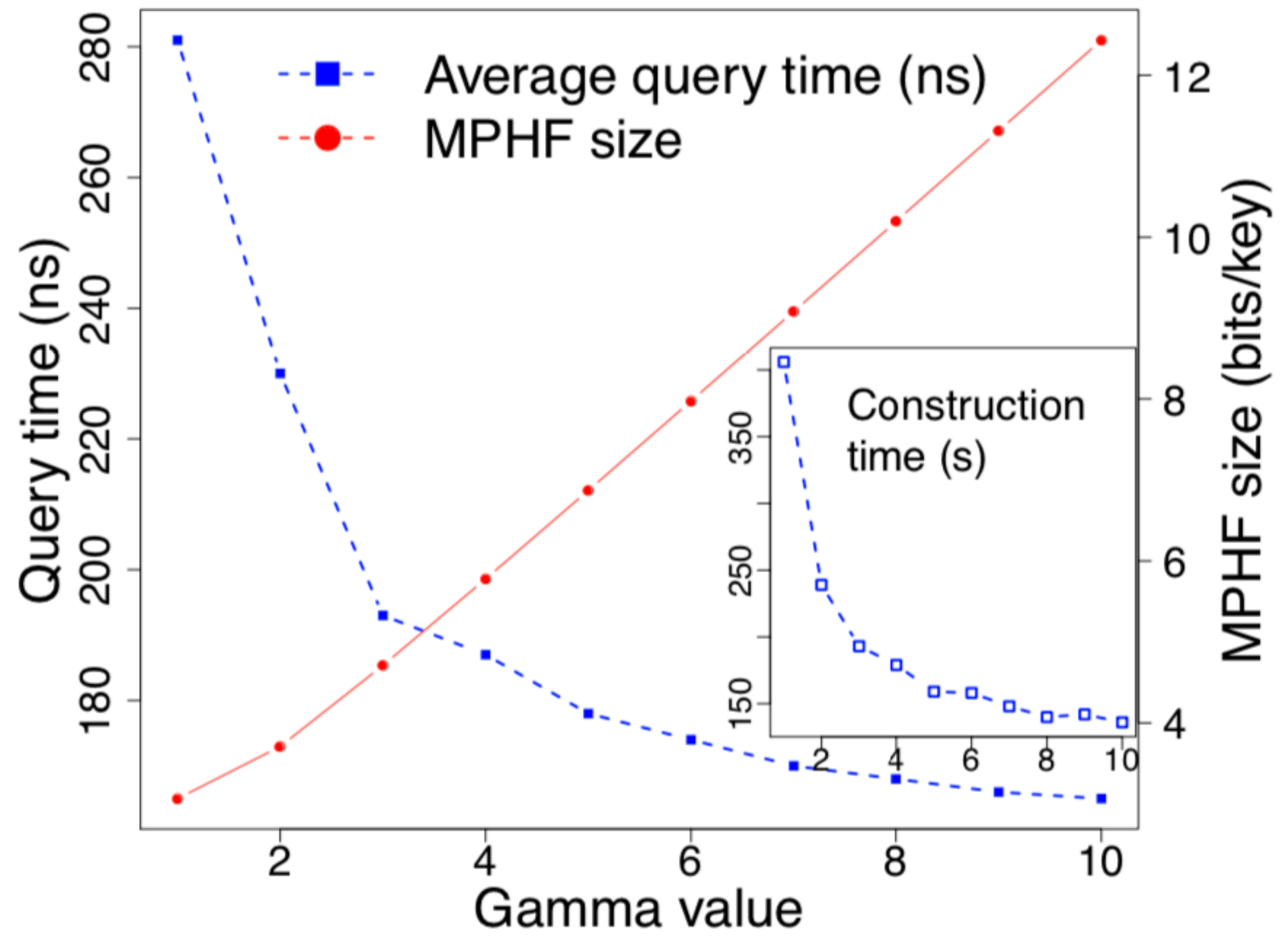
To ensure that the image range of the function is $[1, |F_0|]$, we compute the cumulative rank of each '1' in the bit arrays A_i. Suppose, that d is the smallest value such that $A_d[h_d(x)] = 1$. The minimal perfect hash value is given by $\sum_{i<d}(weight(A_i) + rank(A_d[h_d(x)])$, where $weight(A_i)$ is the number of bits set to '1' in the A_i array, and $rank(A_d[y])$ is the number of bits set to 1 in A_d within the interval $[0, y]$, thus $rank(A_d[y]) = \sum_{j<y} A_d[j]$. This is a classic method also used in other MPHFs [3].

Tradeoff with the γ parameter

The running time of the construction depends on the number of collisions on the A_d arrays, at each level d. One way to reduce the number of collisions, hence to place more keys at each level, is to use bit arrays (A_d and C_d) larger than $|F_d|$. We introduce a parameter $\gamma \in \mathbb{R}$, $\gamma \geq 1$, such that $|C_d| = |A_d| = \gamma|F_d|$. With $\gamma = 1$, the size of A is minimal. With $\gamma \geq 2$, the number of collisions is significantly decreased and thus construction and query times are reduced, at the cost of a larger `MPHF` structure size.

▶ **Lemma 1.** *For $\gamma > 0$, the space of our `MPHF` is $S = \gamma e^{\frac{1}{\gamma}} N$ bits. The maximal space during construction is S when $\gamma \leq \log(2)^{-1}$, and $2S$ bits otherwise.*

Tradeoff with the γ parameter



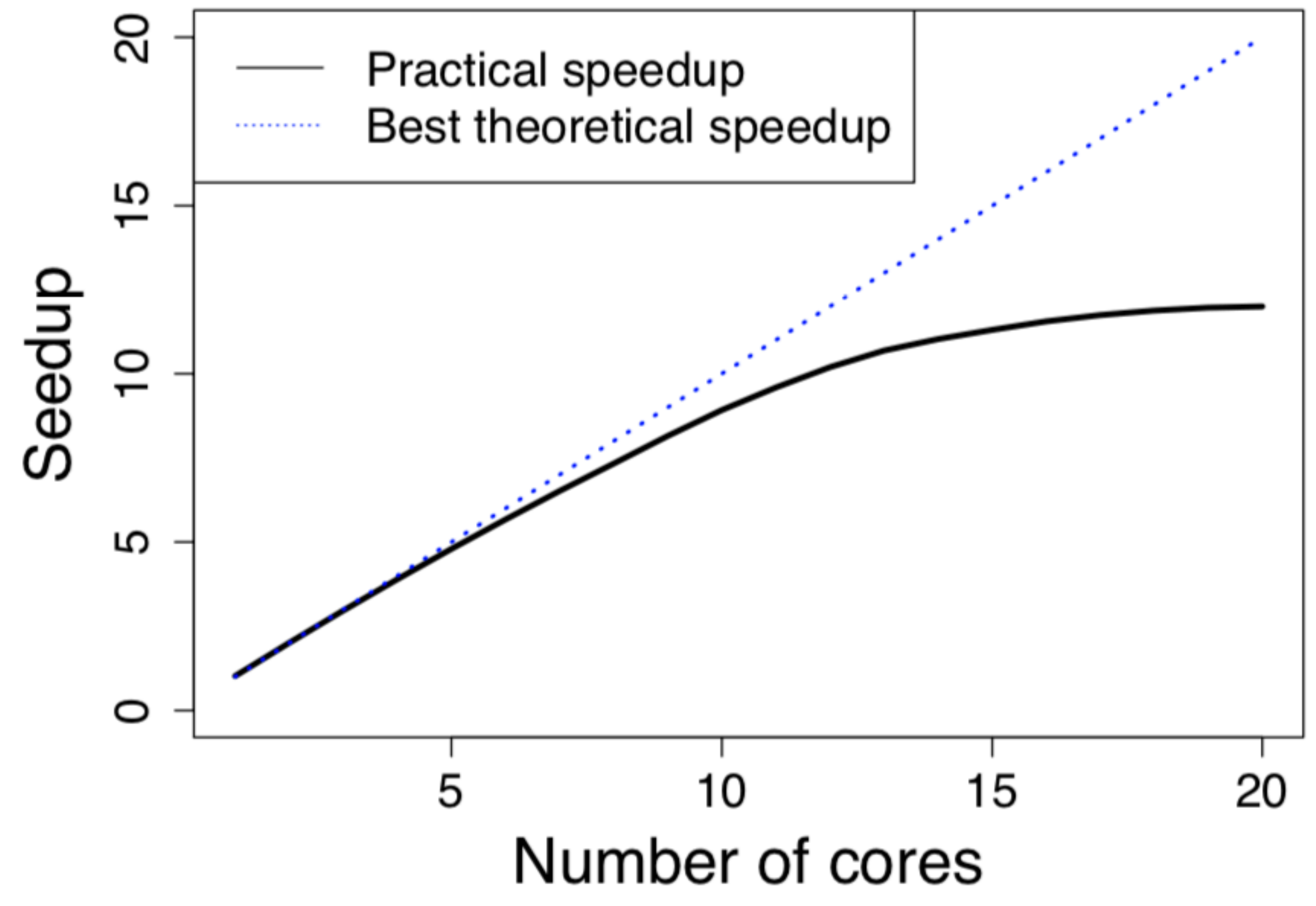
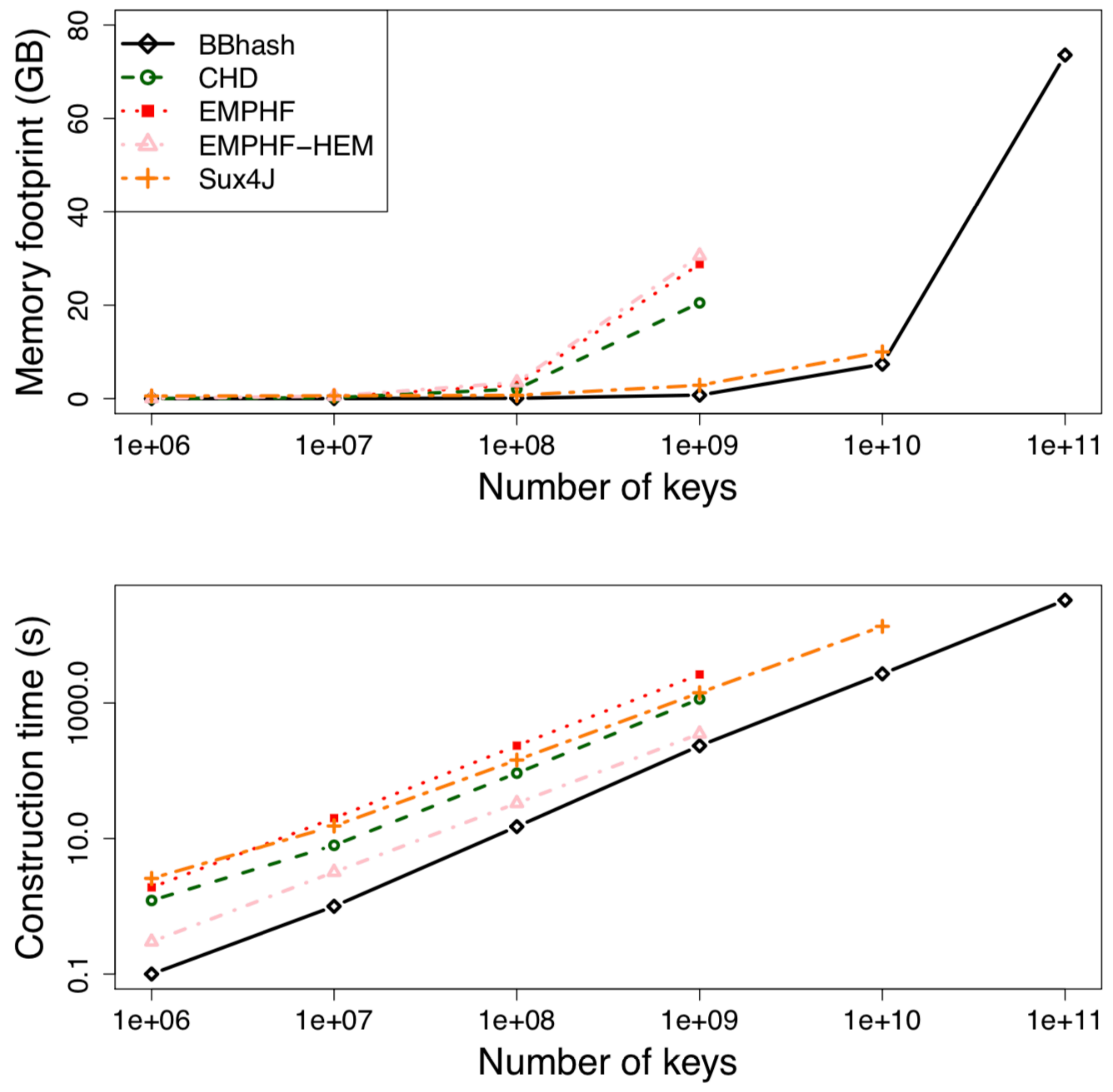


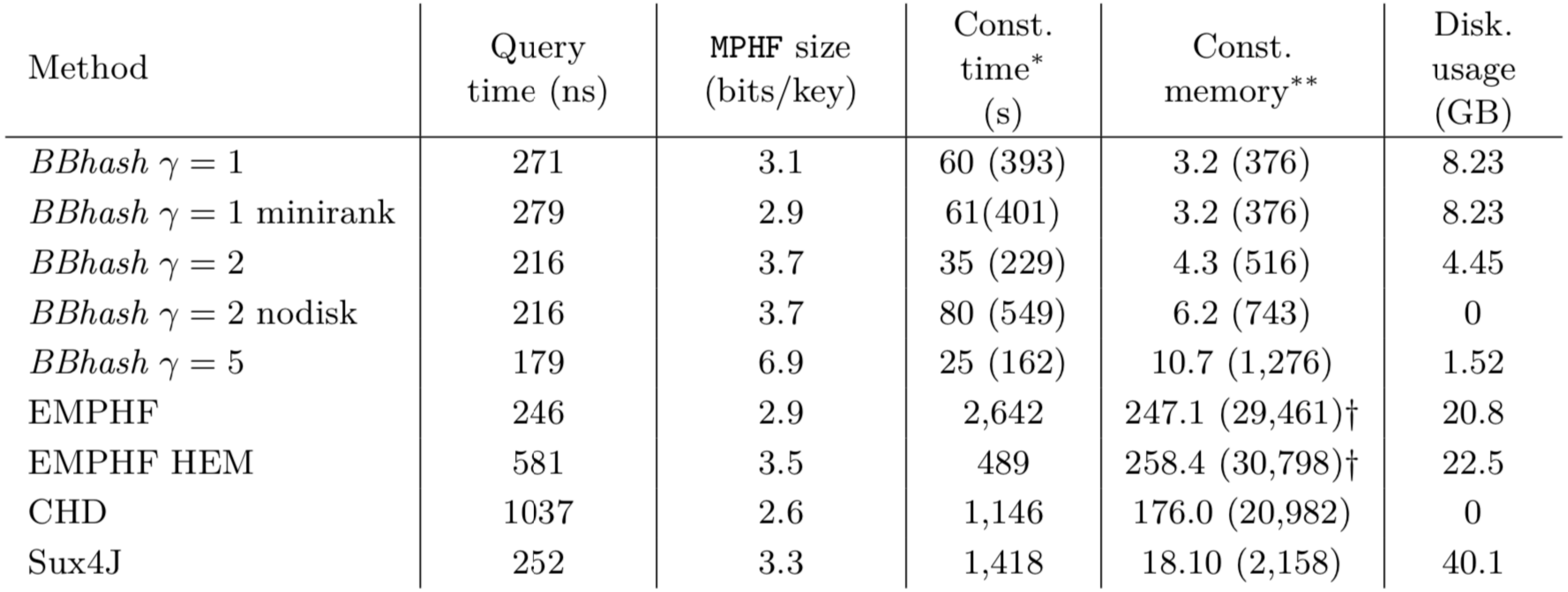
Figure 2 Left: Effects of the gamma parameter on the performance of *BBhash* when run on a set composed of one billion keys, when executed on a single CPU thread. Times and `MPHF` size behave accordingly to the theoretical analysis, respectively $O(e^{(1/\gamma)})$, and $O(\gamma e^{(1/\gamma)})$. Right: Performance of the *BBhash* construction time according to the number of cores, using $\gamma = 2$.

Comparison with other MPHF schemes



🟨 **Figure 3** Memory footprint and construction time with respect to the number of keys. All libraries were run using default parameters, including $\gamma = 2$ for *BBhash*. For a fair comparison, *BBhash* was executed on a single CPU thread. Except for Sux4J, missing data points correspond to runs that exceeded the amount of available RAM. Sux4J limit comes from the disk usage, estimated at approximately 4TB for 10^{11} keys.

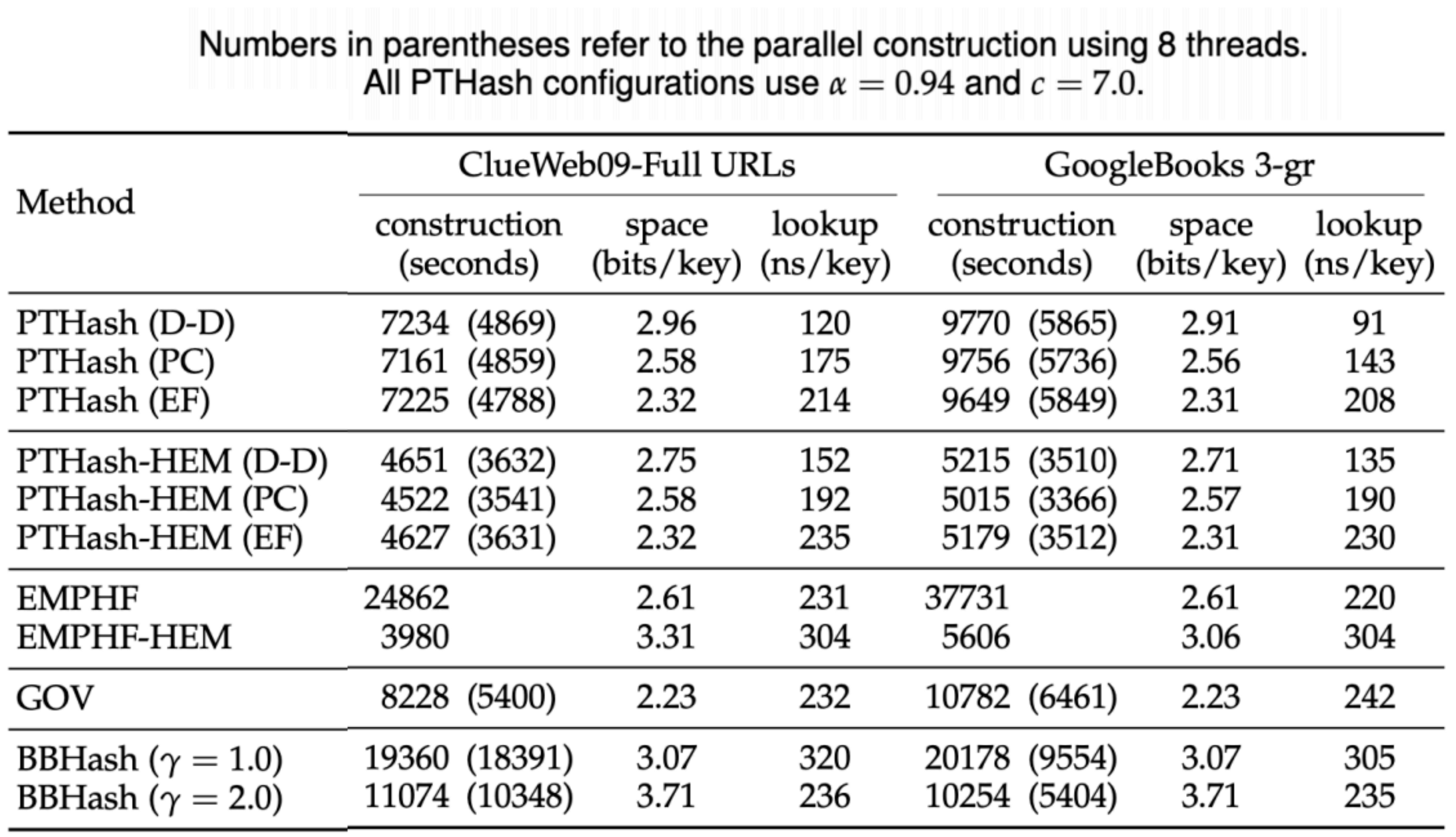
Comparison with other MPHF schemes



Method	Query time (ns)	MPHF size (bits/key)	Const. time* (s)	Const. memory**	Disk. usage (GB)
BBhash $\gamma = 1$	271	3.1	60 (393)	3.2 (376)	8.23
BBhash $\gamma = 1$ minirank	279	2.9	61(401)	3.2 (376)	8.23
BBhash $\gamma = 2$	216	3.7	35 (229)	4.3 (516)	4.45
BBhash $\gamma = 2$ nodisk	216	3.7	80 (549)	6.2 (743)	0
BBhash $\gamma = 5$	179	6.9	25 (162)	10.7 (1,276)	1.52
EMPHF	246	2.9	2,642	247.1 (29,461)†	20.8
EMPHF HEM	581	3.5	489	258.4 (30,798)†	22.5
CHD	1037	2.6	1,146	176.0 (20,982)	0
Sux4J	252	3.3	1,418	18.10 (2,158)	40.1

Even more recent work in this area : PTHash

Dataset	Number of strings
ClueWeb09-Full URLs	4 780 950 911
GoogleBooks 3-gr	7 384 478 110



Numbers in parentheses refer to the parallel construction using 8 threads.
All PTHash configurations use $\alpha = 0.94$ and $c = 7.0$.

Method	ClueWeb09-Full URLs			GoogleBooks 3-gr		
	construction (seconds)	space (bits/key)	lookup (ns/key)	construction (seconds)	space (bits/key)	lookup (ns/key)
PTHash (D-D)	7234 (4869)	2.96	120	9770 (5865)	2.91	91
PTHash (PC)	7161 (4859)	2.58	175	9756 (5736)	2.56	143
PTHash (EF)	7225 (4788)	2.32	214	9649 (5849)	2.31	208
PTHash-HEM (D-D)	4651 (3632)	2.75	152	5215 (3510)	2.71	135
PTHash-HEM (PC)	4522 (3541)	2.58	192	5015 (3366)	2.57	190
PTHash-HEM (EF)	4627 (3631)	2.32	235	5179 (3512)	2.31	230
EMPHF	24862	2.61	231	37731	2.61	220
EMPHF-HEM	3980	3.31	304	5606	3.06	304
GOV	8228 (5400)	2.23	232	10782 (6461)	2.23	242
BBHash ($\gamma = 1.0$)	19360 (18391)	3.07	320	20178 (9554)	3.07	305
BBHash ($\gamma = 2.0$)	11074 (10348)	3.71	236	10254 (5404)	3.71	235

1. Giulio Ermanno Pibiri and Roberto Trani. "*PTHash: Revisiting FCH Minimal Perfect Hashing*". In Proceedings of the 44th International Conference on Research and Development in Information Retrieval (SIGIR). 2021.

2. Giulio Ermanno Pibiri and Roberto Trani. "*Parallel and External-Memory Construction of Minimal Perfect Hash Functions with PTHash*". ArXiv. 2021.