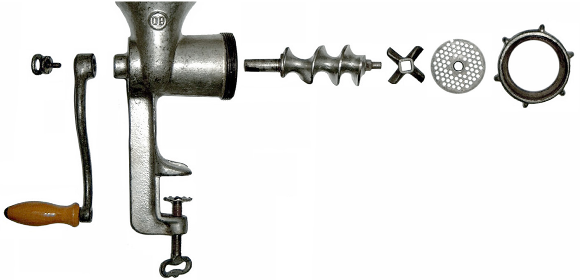
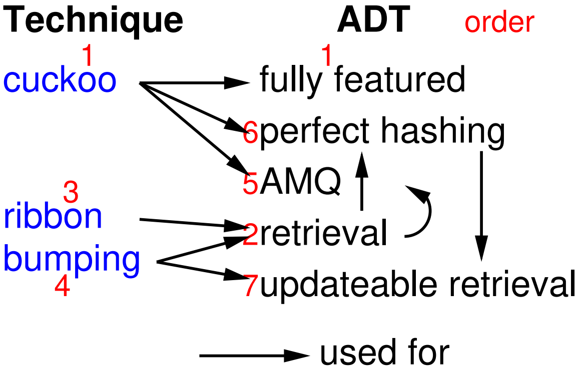
Space Efficient Hash Tables

ITI AG Sanders



Overview



Technique
ADT
order
1
cuckoo
1
fully featured
6perfect hashing
5AMQ
3
ribbon
2retrieval
bumping
4
7updateable retrieval
used for

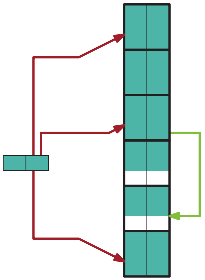
Cuckoo Hashing

GFDL 1.2 Chris Romeiks

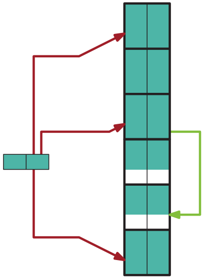
H-ary Bucket Cuckoo Hashing

based on
Pagh Rodler 01, Fotakis Pagh S Spirakis 03,
Dietzfelbinger Weidling 05

- H hash functions address H buckets
- Buckets can store B elements each
- find: check these $H \times B$ possible locations
- delete: find, then overwrite with $\perp$
- insert: can move elements around
  (BFS or random walk)



H-ary Bucket Cuckoo Hashing



- \+ Highly space efficient even for $H = 2$, $B = 4$
- \+ Worst case constant find, delete
- \+ Empirically $\approx 1/\epsilon$ average insertion time when not too close to capacity limit
- − reallocate when full

Capacity Limits $\hat{\alpha}$:

$H \backslash B$	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 on Cuckoo Hashing

Conjecture:
Cuckoo hashing achieves expected insertion time $O(1/\epsilon)$ when the load factor is below $\hat{\alpha}(H, B) - \epsilon$.

Retrieval

key	value
Godzilla	**
Ben Hur	***
Attack of the Killer Tomatoes	*
Three Gifts for Cinderella	****
Howl’s Moving Castle	****
Metropolis	****

Retrieval / Static Function Evaluation

For $S = \{s_1, \ldots, s_n\}$
allow evaluating $f : S \to \{0, 1\}^r$ where $S = \{s_1, \ldots, s_n\}$.

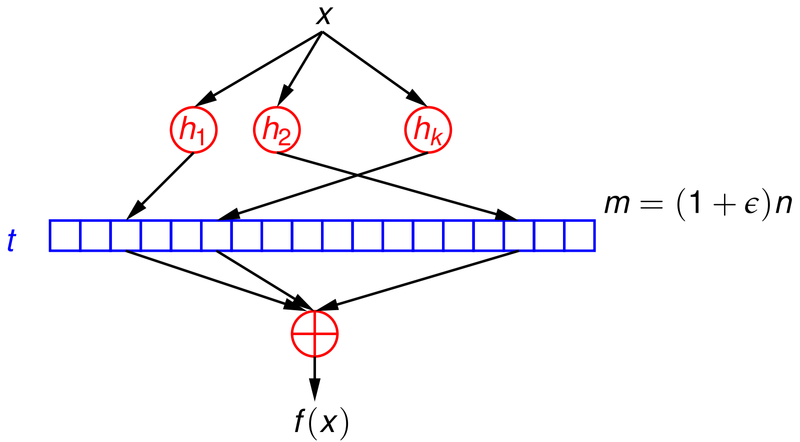
~~key~~	value
~~Godzilla~~	**
~~Ben Hur~~	***
~~Attack of the Killer Tomatoes~~	*
~~Three Gifts for Cinderella~~	****
~~Howl's Moving Castle~~	****
~~Metropolis~~	****

Space near $r \cdot n$ bits?

Retrieval by Linear Algebra

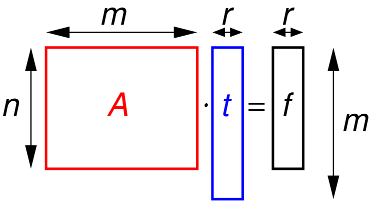
A key x is mapped to k hash functions with range $\mathbb{Z}_m$ and the computed output is

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

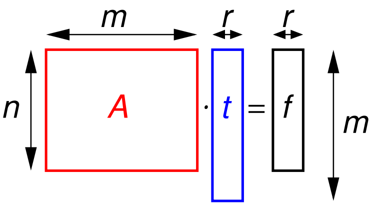


Finding t

Solve a system of linear equations over F_2 with kn nonzeroes determined by the hash values.



Brute Force



- $m = n$
- A is a random matrix

\+ A has full rank with constant probability (store a succeeding hash seed)

− Cubic construction time

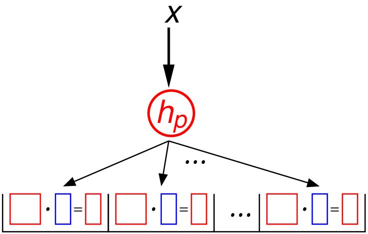
− Linear query time

Sharding – A Standard Trick

Assume $r = \mathrm{O}(1)$.

- Partitioning hash function h_p maps elements to shards of size $\Theta(\log n)$
- Constant time row operations using word parallelism
- $\frac{n}{\log n} \times \frac{\log^3 n}{\log n} = n \log n$ construction time
- Constant query time

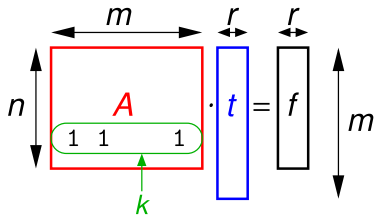
For $r = \mathrm{O}(\log n)$, word size w: Query time $\mathrm{O}\left(\frac{r \log n}{w}\right)$



Sparse Matrices

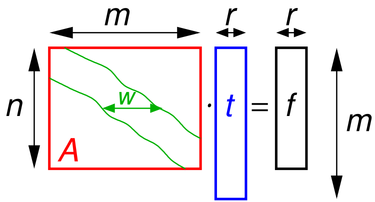
Most well known: $k \in 3..7$ random nonzeroes per row.

+ Linear time construction heuristics for sufficiently large m (typical value $m = 1.21n$)
– Bad locality for query and construction



Ribbon –
Sparse Matrices with Locality

Random bit pattern in a randomly placed window of width w



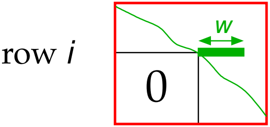
[Dietzfelbinger Weidling 19]:

For $m = (1 + \epsilon)n$ it works for some $w = \Omega\left(\frac{\log n}{\epsilon}\right)$.

+ High locality
+ Row operations can use word parallelism
– w large and dependent on n
  Sharding helps a bit.

Ribbon Solving

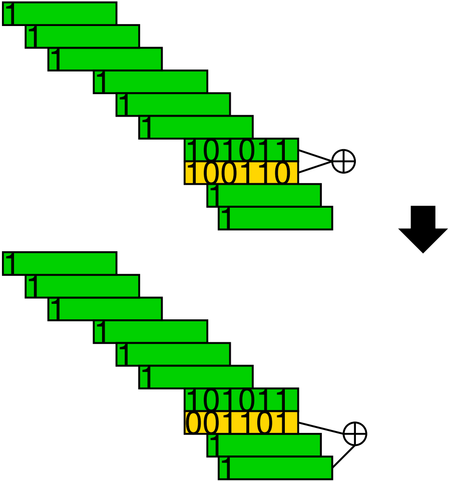
Function ribbonSolve(A, f, **var** $x = 0^m$) :
   bring A into row-echelon form (REM)
   backsubstitution



Ribbon Solving

```
Function ribbonSolve(A, f,  var x = 0^m) :
    placed = ⟨0, . . . , 0⟩ : Array 1..m of {0, 1}^w
    rhs = ⟨0^r, . . . , 0^r⟩ : Array 1..m of {0, 1}^r
    for i := 1 to n do                          -- bring A into row-echelon form
        loop
            if a_i = 0^m then
                if rhs_j = 0 then next iteration of for-loop
                else return "failed after i − 1 rows"
            j:= min {ℓ : a_iℓ = 1}
            if placed_j = 0 then exit loop
            (a_i, f_i) ⊕= (placed_j, rhs_j)
        (placed_j, rhs_j):= (a_i, f_i)
    for j := m to 1 do                          -- backsubtitution
        if placed_j ≠ 0 then x_j:= (x · placed_j) ⊕ rhs_j
```

Ribbon Solving



Ribbon Solving – Analysis

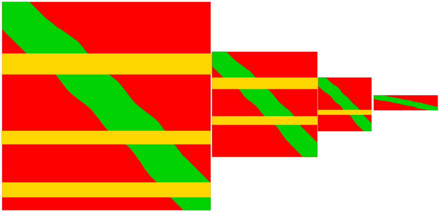
Assume $\max(r, w) = \mathrm{O}(\text{wordSize})$

- Constant time per row operation
- $\mathrm{O}(w)$ row operations per row (e.g., left-to-right processing)
- $\mathrm{O}(rn)$ time for backsubstitution

Overall $\mathrm{O}(n(w + r))$ time using bit parallelism.

Bumped Ribbon Retrieval (BuRR)

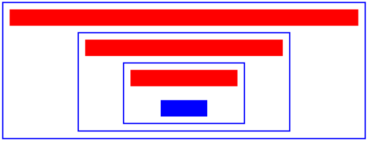
Problem of basic Ribbon: Even if a single row insertion fails.
the entire construction was in vain.
Idea: bump offending rows from the system and handle them separately.



Generic Bumped Retrieval (BuRe)

Class BuRe(E : **set of** *Element*)
  primary : ImperfectRetrieval
  fallback : Retrieval
  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)

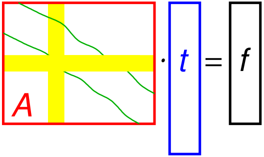


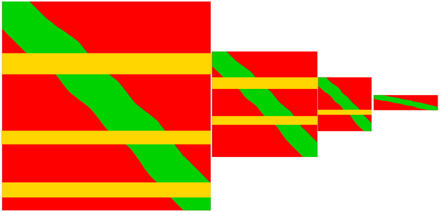
Originally used for filtered retrieval (FiRe) – simple, fast, updateable retrieval with $\approx$ 4 bits overhead per element.

[Müller, Sanders, Schulze, Zhou; Retrieval and Perfect Hashing Using Fingerprinting, SEA 2014]

Bumped Ribbon Retrieval (BuRR)

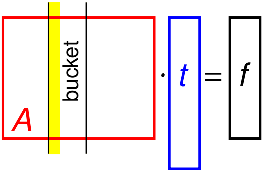
Central Observation:
Rather than identifying specific bumped rows, we can bump ranges of rows based on the position $h_0(x)$ of their window.



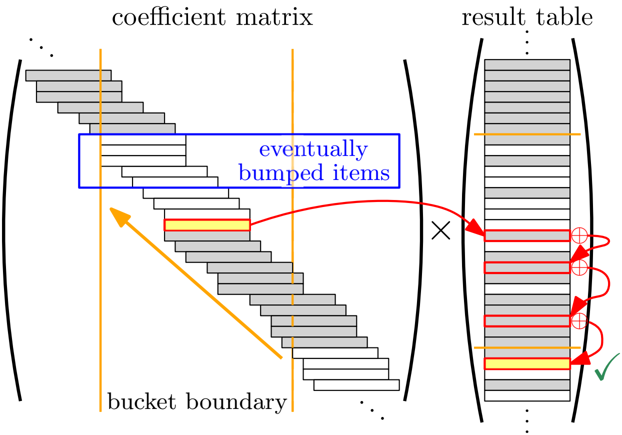


Bumped Ribbon Retrieval (BuRR)

- Partition columns into buckets of size B
- Allow some starting range of each bucket to be bumped
- Element x is mapped to bucket $h_0(x)$ –
  x is bumped if $h_0(x)$ is in the bumped range.
- Insert one bucket at a time from left to right
- Within a bucket, insert from right to left
- Bump remaining bucket when insertion fails
  (possibly more)

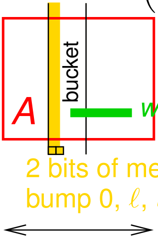


Bumped Ribbon Retrieval (BuRR)



BuRR – Design Choices

$B = \mathrm{O}\left(\frac{w^2}{\log w}\right)$ e.g., 128 or 256



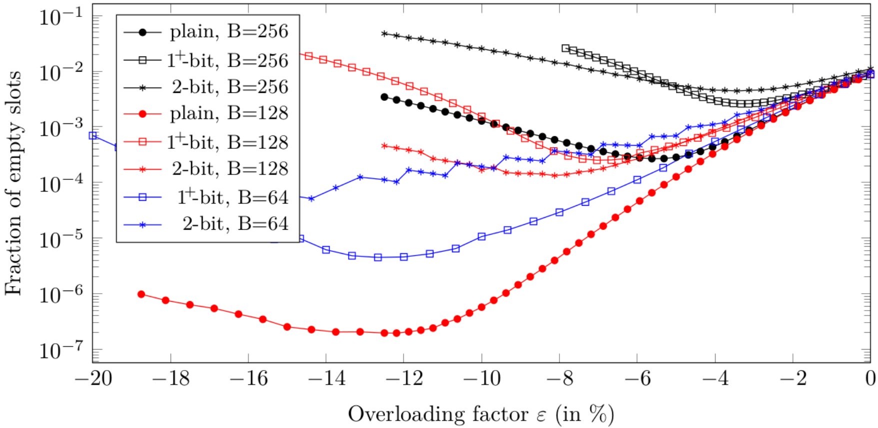
$w = 64$

2 bits of metadata per bucket, i.e.,
bump 0, ℓ, u, or B columns

$m = (1+\epsilon)m$

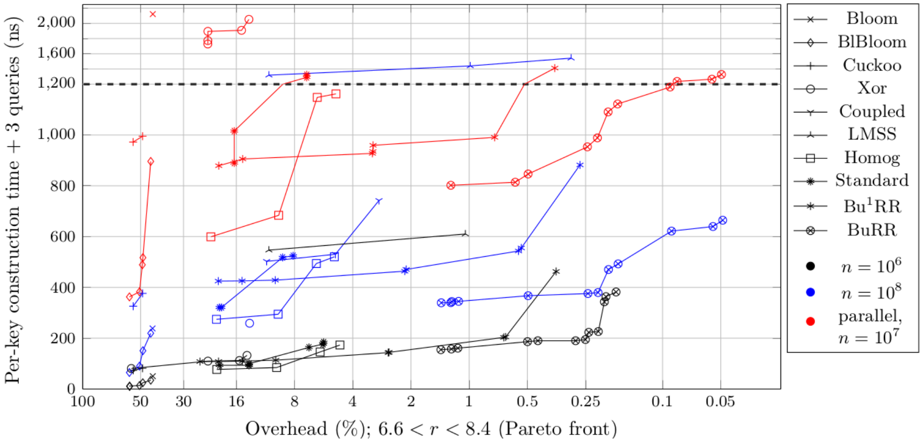
what should ϵ be?

BuRR – Choice of ϵ $(w = 64)$



$\Rightarrow$ overloading almost eliminates empty cells

Space–Performance Tradeoffs

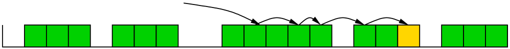


BuRR – Details and Variants

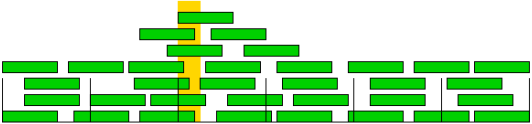
- Interleaved storage of table allows bit parallelism – essentially one population count instruction per retrieved bit.
- Use appropriate $\epsilon > 0$ for ultimate fallback
- Master Hash Codes: $e \to \overbrace{\mathrm{MHC}}^{64\mathrm{bit}} \overset{\text{fast hash function}}{\to}$ further "random" data e.g., use $h(x) = a \cdot x + b$, with $a \bmod 4 = 1$ and odd b.
- 1+ bit metadata: bump 0 or t columns plus exception table
- Sparse bit patterns: e.g. use 8 out of 64 bits per row. Faster for small r
- Bu^{1}RR: Each element is stored in 1 out of 2 layers.
- Parallelization: "implicit" sharding – bump segment of w columns
- Variable bitlength encoding: For prefix-free codes like Huffman this reduces to 1-bit retrieval. Query can be made very fast using specialized interleaving techniques.

BuRR Analysis – Basic Ideas

- Ribbon solving is analogous to a variant of linear probing hashing



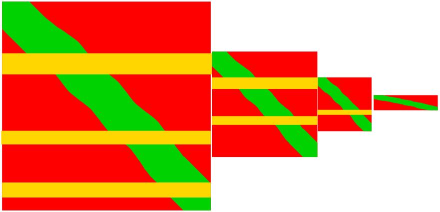
- Bumping mostly eliminates overloading



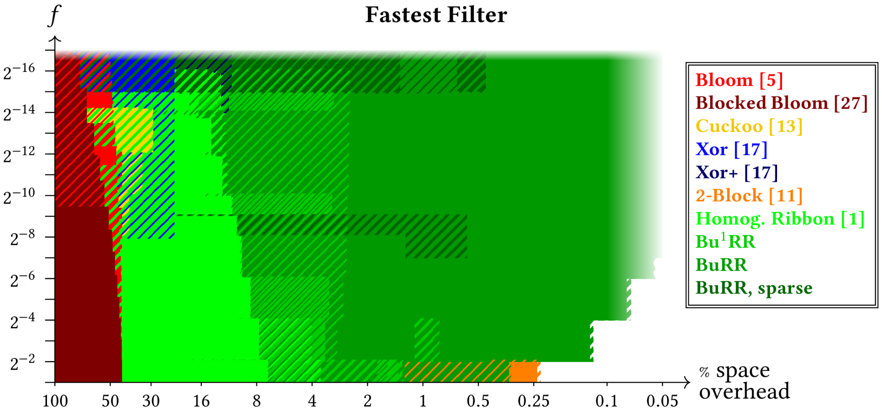
- $B = \mathrm{O}\left(\frac{w^2}{\log w}\right)$
  – larger buckets can have intra-bucket overloading
- Relative space overhead $B = \mathrm{O}\left(\frac{\log w}{rw^2}\right)$

BuRR/Retrieval – Open Problems

- Efficient use of bit-manipulation and SIMD instructions
- Parallelization without sharding
- Fast retrieval of numbers mod p for p not a power of two. (Algebraically this is easy but how to use word parallelism?)
- Dynamization (S available but small update on compressed data structure) for more space efficient variants than FiRe.



Approximate Membership Query Data Structure/Filter (AMQ) aka “Bloom” Filter



AMQs

Maintain approximation $\tilde{S}$ of a set $S = \{s_1, \ldots, s_n\}$.
Query contains$(x) \in \{0, 1\}$

Case $x \in S$, result 1: true positive query

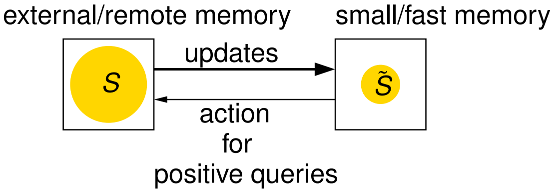
Case $x \notin S$, result 0: true negative query

Case $x \notin S$, result 1: false positive query

false positive rate f

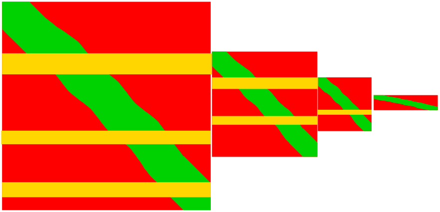
Lower space bound for $\tilde{S}$: 2^{-f}

Typical Application of AMQs



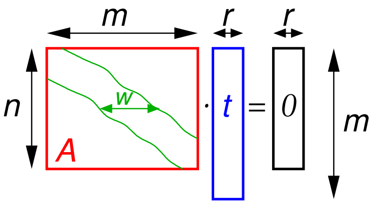
Static Retrieval Based AMQs

With BuRR, space $\log(1/f) + o(1)$ bits per entry.



Homogeneous Ribbon Filter

Solve a homogenous system of equations.
$\Rightarrow$ always solvable.
Take a random solution.

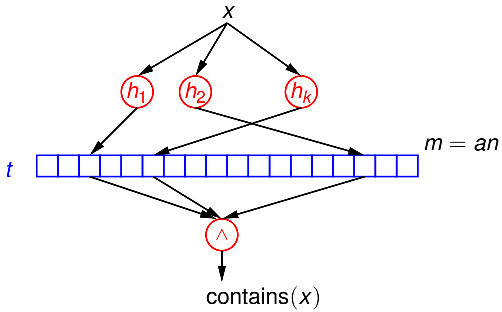


Bloom Filters – Simple Dynamic AMQs

Consider bit vector $b[1..an]$ and
hash functions $h_1, \ldots, h_k$ with range $1..an$.

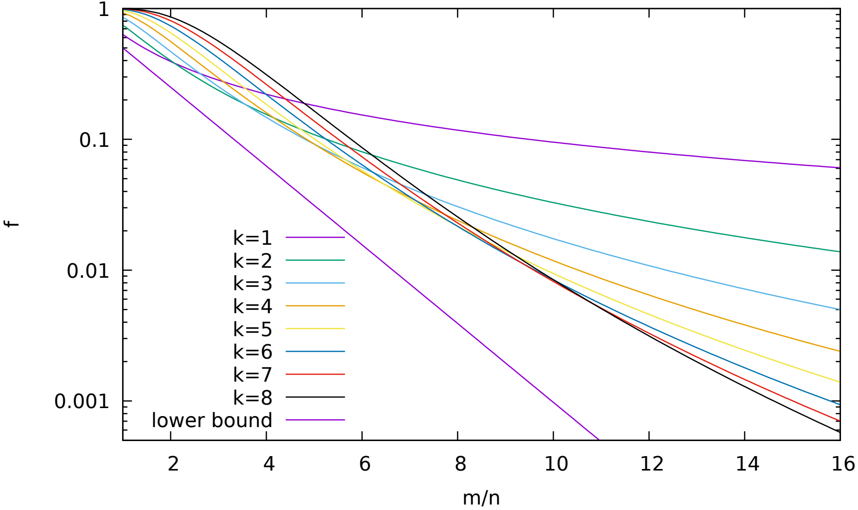
Inserting x: set $b[h_1(x)], \ldots, b[h_k(x)]$.

contains$(x) = b[h_1(x)] \wedge \cdots \wedge b[h_k(x)]$.



What about deletion?

Bloom Filters $f \geq 2^{-0.69a}$

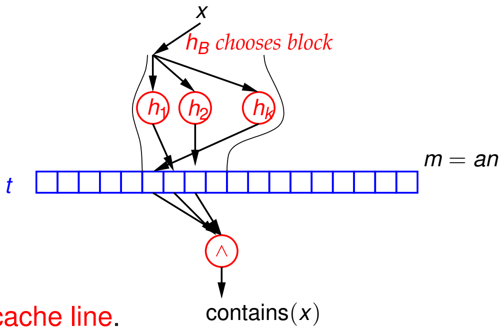


Blocked Bloom Filters

Consider bit vector $b[1..an]$,
a block selection function h_B with range $0..m/B$, and
hash functions $h_1, \ldots, h_k$ with range $1..B$.

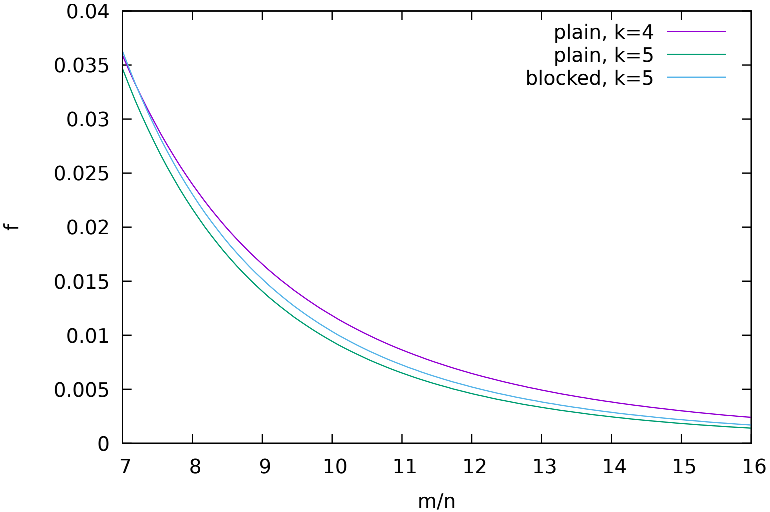
Inserting x: set $b[Bh_B(x) + h_1(x)], \ldots, b[Bh_B(x) + h_k(x)]$.

contains$(x) = b[Bh_B(x) + h_1(x)] \wedge \cdots \wedge b[Bh_B(x) + h_k(x)]$.

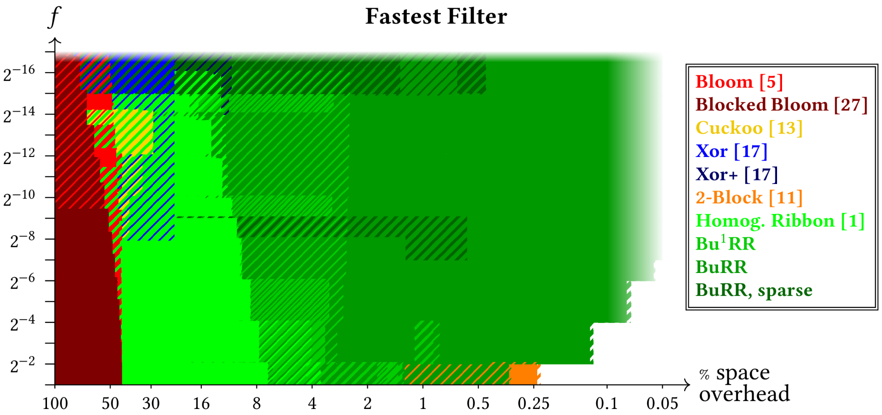


Typically B is one cache line.

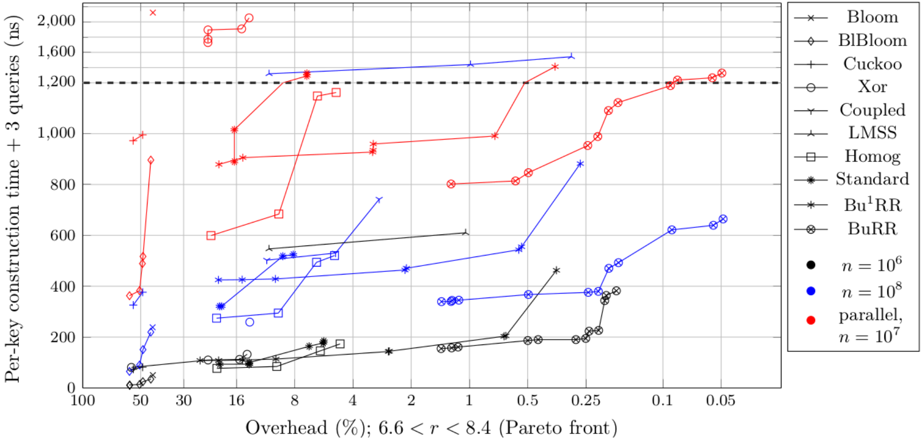
Blocked Bloom Filters *f*



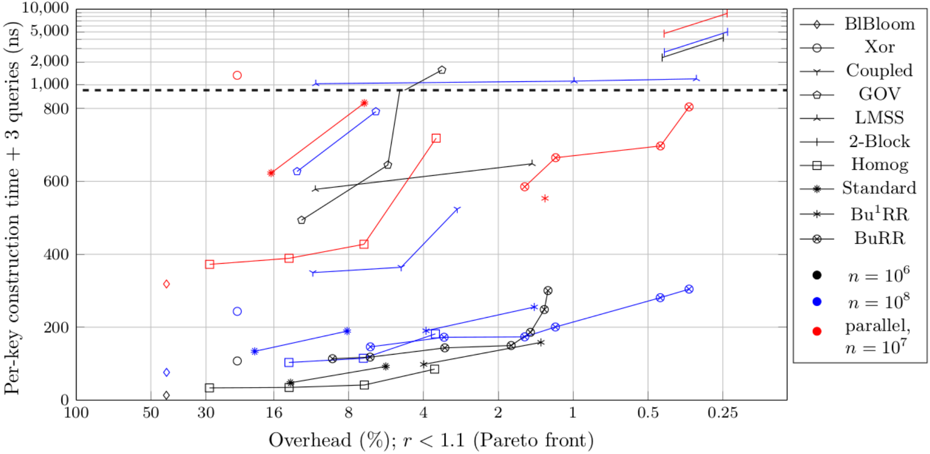
Tradeoff Speed, Space, f



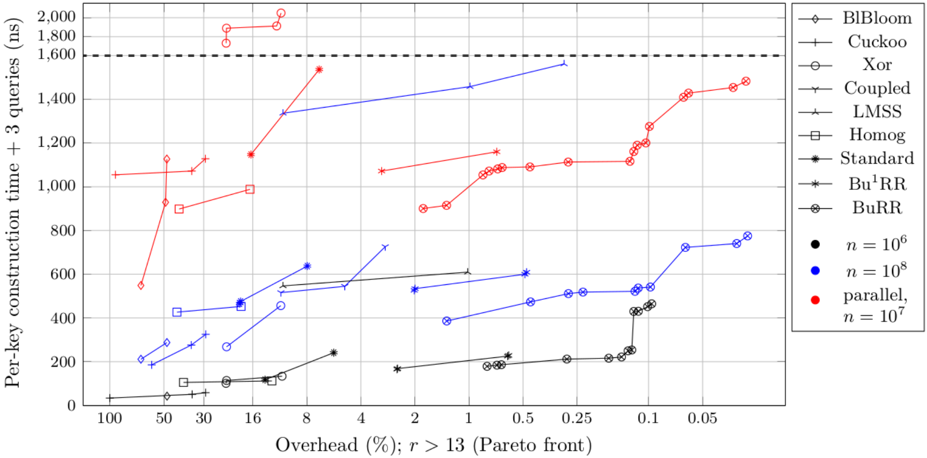
Tradeoff Speed, Space, *f*



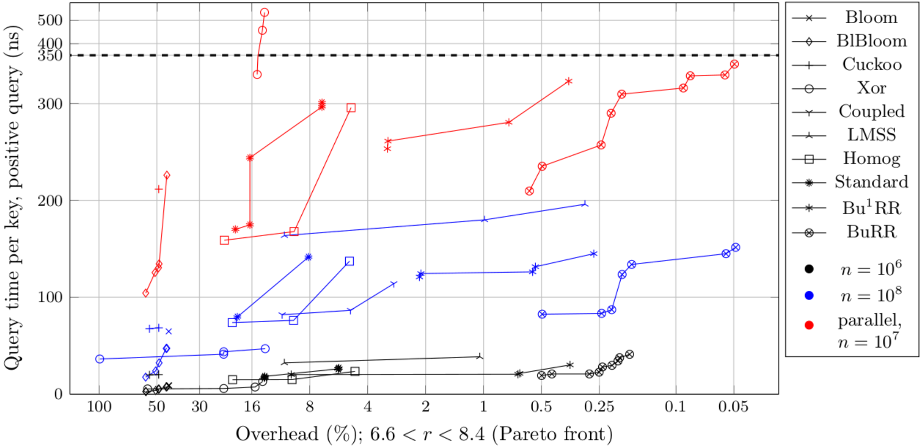
Tradeoff for small r



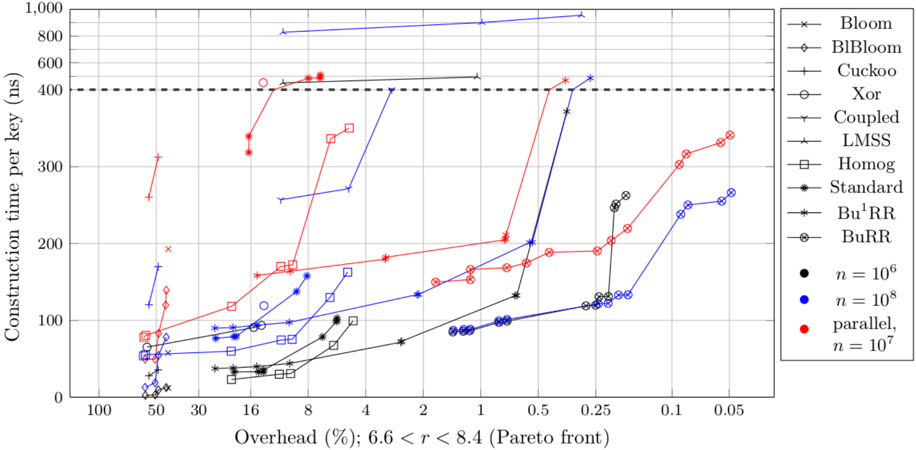
Tradeoff for large r



Tradeoff Query Time – Space ($r = 8$)



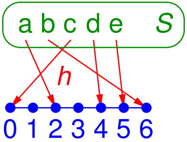
Tradeoff Constr. T. – Space ($r = 8$)



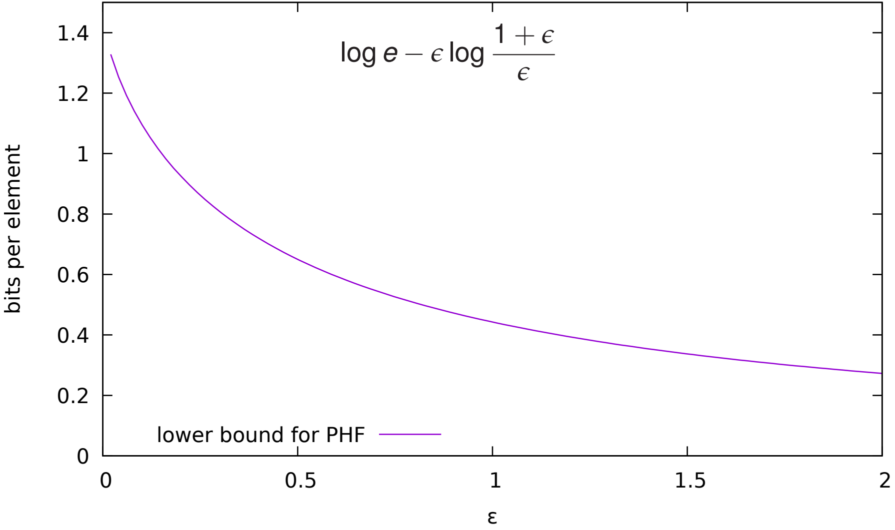
Perfect Hash Functions (PHF)

Given a set $S = \{s_1, \ldots, s_n\}$,
find a function $h : S \rightarrow \mathbb{Z}_m$.

Minimal Perfect Hash Functions (MPHF): $m = n$.



Space Lower Bound $m = (1+\epsilon)n$



Brute Force PHFs

Consider a sequence $h_1, h_2, \ldots$ of random hash functions.

for $i := 1$ **to** ∞ **do** **if** $|h_i(S)| = |S|$ **then** break loop
store i — variable bitlength encoding

$$p := \mathbf{P}[\text{success}] = \frac{n!\binom{m}{n}}{m^n}$$

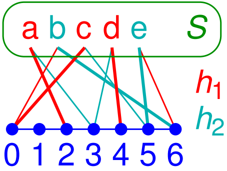
i has geometric distribution with parameter p
Its entropy is about $\log 1/p$. Let $m = (1+\epsilon)n$

$$\log\frac{1}{p} \approx n\log m - n\log\frac{n}{e} - n\log\frac{m}{n} - (m-n)\log\frac{m}{m-n}$$
$$= n\left(\log e - \epsilon\log\frac{1+\epsilon}{\epsilon}\right)$$

use $n! \sim n\ln\frac{n}{e}$, $\log\binom{m}{n} \sim n\log\frac{m}{n} + (m-n)\log\frac{m}{n-k}$ when $m = \Theta(n)$

PHFs via
Cuckoo-Hashing and Retrieval

Insert S into an m-cell cuckoo-hash-table using 2^r hash functions.
Store the choice of hash function for each $x \in S$ in an
r-bit retrieval data structure f.



$$h(x) := h_{f(x)}(x)$$

With BuRR:

r	m	bits per el.	lower bound
1	$\approx 2n$	≈ 1	0.443
2	$\approx 1.024n$	≈ 2	1.313