Space Efficient Hash Tables
Overview

Technique

1. cuckoo

ADT

1. fully featured

6. perfect hashing

5. AMQ

2. retrieval

7. updateable retrieval

used for

order
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 ⊥
- 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 $\approx \frac{1}{\epsilon}$ **average insertion time** when not too close to capacity limit

- **reallocate** when full

### Capacity Limits $\hat{\alpha}$:

<table>
<thead>
<tr>
<th>$H \backslash B$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>.5</td>
<td>.897</td>
<td>.959</td>
<td>.980</td>
<td>.989</td>
<td>.994</td>
<td>.996</td>
<td>.998</td>
</tr>
<tr>
<td>3</td>
<td>.918</td>
<td>.988</td>
<td>.997</td>
<td>.9992</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>.977</td>
<td>.998</td>
<td>.998</td>
<td>.99997</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conjecture:
Cuckoo hashing achieves expected insertion time $O(1/\epsilon)$ when the load factor is below $\hat{\alpha}(H, B) - \epsilon$. 
### Retrieval

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Godzilla</td>
<td>**</td>
</tr>
<tr>
<td>Ben Hur</td>
<td>***</td>
</tr>
<tr>
<td>Attack of the Killer Tomatoes</td>
<td>*</td>
</tr>
<tr>
<td>Three Gifts for Cinderella</td>
<td>****</td>
</tr>
<tr>
<td>Howl’s Moving Castle</td>
<td>****</td>
</tr>
<tr>
<td>Metropolis</td>
<td>****</td>
</tr>
</tbody>
</table>
For $S = \{s_1, \ldots, s_n\}$ allow evaluating $f : S \rightarrow \{0, 1\}^r$ where $S = \{s_1, \ldots, s_n\}$.

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Godzilla</td>
<td>**</td>
</tr>
<tr>
<td>Ben Hur</td>
<td>***</td>
</tr>
<tr>
<td>Attack of the Killer Tomatoes</td>
<td>*</td>
</tr>
<tr>
<td>Three Gifts for Cinderella</td>
<td>****</td>
</tr>
<tr>
<td>Howl’s Moving Castle</td>
<td>****</td>
</tr>
<tr>
<td>Metropolis</td>
<td>****</td>
</tr>
</tbody>
</table>

Space near $r \cdot n$ bits?
A key \( x \) is mapped to \( k \) hash functions with range \( \mathbb{Z}_m \) and the computed output is

\[
f(x) := t[h_1(x)] \oplus \cdots \oplus t[h_k(x)]
\]
Solve a system of linear equations over $F_2$ with $kn$ nonzeros 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 $r = O(1)$.

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

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

+ Linear time construction heuristics for sufficiently large $m$
  (typical value $m = 1.21n$)

- Bad locality for query and construction
Random bit pattern in a randomly placed window of width $w$

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

- High locality
- Row operations can use word parallelism
- $w$ large and dependent on $n$

Sharding helps a bit.
**Function** ribbonSolve($A$, $f$, \texttt{var} $x = 0^m$) :

- bring $A$ into **row-echelon form** (REM)
- backsubstitution

\[
\text{row } i
\]

\[
0
\]
Function ribbonSolve($A$, $f$, \textbf{var} $x = 0^m$):

placed $= \langle 0, \ldots, 0 \rangle$ : Array $1..m$ of $\{0,1\}^w$

rhs $= \langle 0^r, \ldots, 0^r \rangle$ : Array $1..m$ of $\{0,1\}^r$

for $i := 1$ to $n$ do
    \hspace{1cm} \text{-- bring $A$ into row-echelon form}
    loop
        \hspace{1cm} \text{if } a_i = 0^m \text{ then}
            \hspace{1cm} \text{if } \text{rhs}_j = 0 \text{ then next iteration of for-loop}
            \hspace{1cm} \text{else return} \text{ "failed after } i - 1 \text{ rows"}
        \hspace{1cm} j := \min \{ \ell : a_{i\ell} = 1 \}
        \hspace{1cm} \text{if placed}_j = 0 \text{ then exit loop}
        \hspace{1cm} (a_i, f_i) \oplus= (\text{placed}_j, \text{rhs}_j)
        \hspace{1cm} (\text{placed}_j, \text{rhs}_j) := (a_i, f_i)$
    \hspace{1cm} for $j := m$ to $1$ do
        \hspace{1cm} \text{-- backsubstitution}
        \hspace{1cm} \text{if placed}_j \neq 0 \text{ then } x_j := (x \cdot \text{placed}_j) \oplus \text{rhs}_j
Ribbon Solving
Assume \( \max(r, w) = O(\text{wordSize}) \)

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

Overall \( O(n(w + r)) \) time using bit parallelism.
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( \( E : \text{set of} \ Element \) )
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.

\[
A \cdot t = f
\]
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)

![Diagram of bucket insertion]

$t \cdot f = A$
Bumped Ribbon Retrieval (BuRR)

bucket boundary

coefficient matrix

eventually bumped items

result table

✓
⊕
⊕
⊕
BuRR – Design Choices

\[ B = O\left(\frac{w^2}{\log w}\right) \]

e.g., 128 or 256

\[ w = 64 \]

2 bits of metadata per bucket, i.e., bump 0, \( \ell \), \( u \), or \( B \) columns

\[ m = (1 + \epsilon)m \]

what should \( \epsilon \) be?
$\Rightarrow$ overloading almost eliminates empty cells
Space–Performance Tradeoffs

![Graph showing space-performance tradeoffs with different methods such as Bloom, BlBloom, Cuckoo, Xor, Coupled, LMSS, Homog, Standard, Bu^3RR, BuRR, with overhead ranging from 6.6 to 8.4 percent (Pareto front).]
Interleaved storage of table allows bit parallelism – essentially one population count instruction per retrieved bit.

Use appropriate $\varepsilon > 0$ for ultimate fallback

$\text{Master Hash Codes: } e \rightarrow \text{MHC fast hash function} \rightarrow \text{further “random” data}$

e.g., use $h(x) = a \cdot x + b$, with $a \mod 4 = 1$ 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 $r$

$Bu^1RR$: 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
- \( B = \mathcal{O}\left(\frac{w^2}{\log w}\right) \)
  - larger buckets can have intra-bucket overloading
- Relative space overhead \( B = \mathcal{O}\left(\frac{\log w}{rw^2}\right) \)
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

Fastest Filter

Bloom [5]
Blocked Bloom [27]
Cuckoo [13]
Xor [17]
Xor+ [17]
2-Block [11]
Homog. Ribbon [1]
Bu¹RR
BuRR
BuRR, sparse
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 $x \not\in S$, result 0: true negative query  
Case $x \not\in S$, result 1: false positive query  

false positive rate $f$

Lower space bound for $\tilde{S}$: $2^{-f}$
Typical Application of AMQs

external/remote memory  small/fast memory

$S$  $\tilde{S}$

updates  action for positive queries
Static Retrieval Based AMQs

With BuRR, space $\log(1/f) + o(1)$ bits per entry.
Solve a homogenous system of equations.
⇒ 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 $b[h_1(x)], \ldots, b[h_k(x)]$.

$\text{contains}(x) = b[h_1(x)] \land \cdots \land b[h_k(x)]$.

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

$\text{contains}(x) = b[Bh_B(x) + h_1(x)] \land \cdots \land b[Bh_B(x) + h_k(x)]$.

Typically $B$ is one cache line.
Blocked Bloom Filters $f$

- **plain, k=4**
- **plain, k=5**
- **blocked, k=5**
Tradeoff Speed, Space, $f$

Fastest Filter

- Bloom [5]
- Blocked Bloom [27]
- Cuckoo [13]
- Xor [17]
- Xor+ [17]
- 2-Block [11]
- Homog. Ribbon [1]
- Bu$^1$RR
- BuRR
- BuRR, sparse

% space overhead
Tradeoff Speed, Space, $f$
Tradeoff for small $r$

![Graph showing tradeoff for small $r$.](image)

Per-key construction time + 3 queries (ns)

Overhead (%); $r < 1.1$ (Pareto front)

Symbols and Legends:
- BlBloom
- Xor
- Coupled
- GOV
- LMSS
- 2-Block
- Homog
- Standard
- $Bu^1RR$
- $BuRR$

- $n = 10^6$
- $n = 10^8$
- parallel,
  $n = 10^7$

Peter Sanders, et al.
Tradeoff for large $r$

Per-key construction time + 3 queries (ns)

Overhead (%); $r > 13$ (Pareto front)

- BlBloom
- Cuckoo
- Xor
- Coupled
- LMSS
- Homog
- Standard
- Bu^1RR
- BuRR

- $n = 10^6$
- $n = 10^8$
- parallel,
- $n = 10^7$
Tradeoff Query Time – Space \((r = 8)\)
Tradeoff Constr. T. – Space \( (r = 8) \)
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**  
\[ m = (1 + \epsilon) n \]  

\[ \log e - \epsilon \log \frac{1 + \epsilon}{\epsilon} \]  

![Graph showing the lower bound for PHF](image)

**Bits per element**  

0  0.5  1  1.5  2  

**\( \epsilon \)**  

0  0.2  0.4  0.6  0.8  1  1.2  1.4  

**Lower bound for PHF**
Brute Force PHFs

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

\[
\text{for } i := 1 \text{ to } \infty \text{ do if } |h_i(S)| = |S| \text{ then break loop}
\]

store \( i \)  

\[ p := P[\text{success}] = \frac{n!(\begin{pmatrix} m \\ n \end{pmatrix})}{m^n} \]

\( i \) 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 \( n! \sim n \ln n - n \), \( \log (\begin{pmatrix} m \\ n \end{pmatrix}) \sim n \log \frac{m}{n} + (m - n) \log \frac{m}{m - n} \) when \( m = \Theta(n) \)
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:

<table>
<thead>
<tr>
<th>$r$</th>
<th>$m$</th>
<th>bits per el.</th>
<th>lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\approx 2n$</td>
<td>$\approx 1$</td>
<td>0.443</td>
</tr>
<tr>
<td>2</td>
<td>$\approx 1.024n$</td>
<td>$\approx 2$</td>
<td>1.313</td>
</tr>
</tbody>
</table>