Near-Optimal Space
Perfect Hashing Algorithm

Nivio Ziviani
Fabiano C. Botelho
Department of Computer Science
Federal University of Minas Gerais, Brazil

IR Course
CS Department, UFMG, February 19th, 2014
Objective of the Presentation

Present a perfect hashing algorithm that uses the idea of partitioning the input key set into small buckets:

- Key set fits in the internal memory
  - Internal Random Access Memory algorithm (RAM)

- Key set larger than the internal memory
  - External Memory (cache-aware) algorithm (EM)
Objective of the Presentation

Present a perfect hashing algorithm that uses the idea of partitioning the input key set into small buckets:

- Key set fits in the internal memory
  - Internal Random Access Memory algorithm (RAM)
- Key set larger than the internal memory
  - External Memory (cache-aware) algorithm (EM)

Theoretically well-founded, time efficient, highly scalable and near space-optimal
Perfect Hash Function

Static key set $S$ of size $n$

Hash Table

Perfect Hash Function

$S \subseteq U$, where $|U| = u$
Minimal Perfect Hash Function

Static key set $S$ of size $n$

$S \subseteq U$, where $|U| = u$
Where to use a PHF or a MPHF?

- Access items based on the value of a key is ubiquitous in Computer Science

- Work with huge static key sets:
  - In data warehousing applications:
    - On-Line Analytical Processing (OLAP) applications
  - In Web search engines:
    - Large vocabularies
    - Map long URLs in smaller integer numbers that are used as IDs
Crawling

Web

Crawling Pages

Pages

Parser

URLs To Be Crawled

Visited URLs

New URLs

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
Representing Visited URLs

- MPHFs is the most compact way of representing the set of visited URLs.
- Enable to keep much more URLs in main memory of each machine.
- When the set of new URLs becomes large, a new MPHF is generated for the whole set of URLs.
**Indexing**

### Vocabulary

<table>
<thead>
<tr>
<th>Term 1</th>
<th>Doc 1</th>
<th>Doc 5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term 2</td>
<td>Doc 1</td>
<td>Doc 2</td>
<td>...</td>
</tr>
<tr>
<td>Term 3</td>
<td>Doc 3</td>
<td>Doc 4</td>
<td>...</td>
</tr>
<tr>
<td>Term 4</td>
<td>Doc 7</td>
<td>Doc 9</td>
<td>...</td>
</tr>
<tr>
<td>Term 5</td>
<td>Doc 6</td>
<td>Doc 10</td>
<td>...</td>
</tr>
<tr>
<td>Term 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Term 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Term 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Term t</td>
<td>Doc 9</td>
<td>Doc 11</td>
<td>...</td>
</tr>
</tbody>
</table>

### Collection of documents

- Doc 1
- Doc 2
- Doc 3
- Doc 4
- Doc 5
- ...
- Doc n
Representing the Vocabulary

Vocabulary

Inverted List

| Term 1 | Doc 1 | Doc 5 | ...
|--------|-------|-------|---
| Term 2 | Doc 1 | Doc 2 | ...
| Term 3 | Doc 3 | Doc 4 | ...
| Term 4 | Doc 7 | Doc 9 | ...
| Term 5 | Doc 6 | Doc 10 | ...
| Term 6 | Doc 1 | Doc 5 | ...
| Term 7 | Doc 9 | Doc 11 | ...
| Term 8 |       |       | ---
| ...    |       |       | ---
| Term t |       |       | ---

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
Mapping URLs to Web Graph Vertices

URLS

| URL 1 |
| URL 2 |
| URL 3 |
| URL 4 |
| URL 5 |
| URL 6 |
| URL 7 |
| ... |
| URL n |

Web Graph Vertices

0
1
2
3
4
5
6
...
n-1

LATIN - LABoratory for Treating INformation (www.dcc.ufmg.br/latin)
Mapping URLs to Web Graph Vertices

URLS

URL 1
URL 2
URL 3
URL 4
URL 5
URL 6
URL 7
...
URL n

MPHF

Web Graph Vertices

0
1
2
3
4
5
6
...
n-1
Information Theoretical Lower Bounds for Storage Space

- PHFs ($m \approx n$): Storage Space $\geq \frac{n^2}{m} \log e$

- MPHFs ($m = n$): Storage Space $\geq n \log e$

$m < 3n$

$log_e \approx 1.4427$
Uniform Hashing Versus Universal Hashing

Key universe $U$ of size $u$ → Hash function → Range $M$ of size $m$
Uniform hashing

- # of functions from U to M?
  \[ m^u \]

- # of bits to encode each function
  \[ u \log m \]

- Independent functions with values uniformly distributed
Uniform Hashing Versus Universal Hashing

**Key universe**
- $U$ of size $u$

**Hash function**
- Mapping from $U$ to $M$

**Range M of size m**

### Uniform hashing
- # of functions from $U$ to $M$? $m^u$
- # of bits to encode each function: $u \log m$
- Independent functions with values uniformly distributed

### Universal hashing
- A family of hash functions $\mathcal{H}$ is universal if:
  - for any pair of distinct keys $(x_1, x_2)$ from $U$ and
  - a hash function $h$ chosen uniformly from $\mathcal{H}$ then:
    $$\Pr(h(x_1) = h(x_2)) \leq \frac{1}{m}$$
Intuition Behind Universal Hashing

- We often lose relatively little compared to using a completely random map (uniform hashing).
- If $S$ of size $n$ is hashed to $n^2$ buckets, with probability more than $\frac{1}{2}$, no collisions occur.
  - Even with complete randomness, we do not expect little $o(n^2)$ buckets to suffice (the birthday paradox).
  - So nothing is lost by using a universal family instead!
Related Work

- Theoretical Results
  (use uniform hashing)

- Practical Results
  (assume uniform hashing for free)

- Heuristics
# Theoretical Results

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schmidt and Siegel (1990)</td>
<td>Not analyzed</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Hagerup and Tholey (2001)</td>
<td>$O(n + \log \log u)$</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
## Theoretical Results – Use Uniform Hashing

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schmidt &amp; Siegel (1990)</td>
<td>Not analyzed</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Hagerup &amp; Tholey (2001)</td>
<td>O(n+log log u)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Theoretic EM (CIKM 2007)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

## Practical Results – Assume Uniform Hashing For Free

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Czech, Havas &amp; Majewski (1992)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Majewski, Wormald, Havas &amp; Czech (1996)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Pagh (1999)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Botelho, Kohayakawa, &amp; Ziviani (2005)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
</tbody>
</table>
## Practical Results – Assume Uniform Hashing For Free

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Czech, Havas &amp; Majewski (1992)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Majewski, Wormald, Havas &amp; Czech (1996)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Pagh (1999)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Botelho, Kohayakawa, &amp; Ziviani (2005)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>RAM (WADS 2007)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
## Practical Results – Assume Uniform Hashing For Free

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Czech, Havas &amp; Majewski (1992)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Majewski, Wormald, Havas &amp; Czech (1996)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Pagh (1999)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Botelho, Kohayakawa, &amp; Ziviani (2005)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>RAM (WADS 2007)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Heuristic EM (CIKM 2007)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
# Empirical Results

<table>
<thead>
<tr>
<th>Work</th>
<th>Application</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox, Chen &amp; Heath (1992)</td>
<td>Index data in CD-ROM</td>
<td>Exp.</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Lefebvre &amp; Hoppe (2006)</td>
<td>Sparse spatial data</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Chang, Lin &amp; Chou (2005, 2006)</td>
<td>Data mining</td>
<td>O(n)</td>
<td>O(1)</td>
<td>Not analyzed</td>
</tr>
</tbody>
</table>
Internal Random Access and External Memory Algorithms

- Near-optimal space
- Evaluation in constant time
- Function generation in linear time
- Simple to describe and implement
- Known algorithms with near-optimal space either:
  - Require exponential time for construction and evaluation, or
  - Use near-optimal space only asymptotically, for large $n$
- Acyclic random hypergraphs
  - Used before by Majewski et al (1996): $O(n \log n)$ bits
- We proceed differently: $O(n)$ bits
  (we changed space complexity, close to theoretical lower bound)
Random Hypergraphs (r-graphs)

- 3-graph:

  0  1  2  3  4  5

- 3-graph is induced by three uniform hash functions
Random Hypergraphs (r-graphs)

- 3-graph:

- 3-graph is induced by three uniform hash functions

\[ h_0(\text{jan}) = 1 \quad h_1(\text{jan}) = 3 \quad h_2(\text{jan}) = 5 \]
Random Hypergraphs (r-graphs)

- 3-graph:

- 3-graph is induced by three uniform hash functions

\[ h_0(\text{jan}) = 1 \quad h_1(\text{jan}) = 3 \quad h_2(\text{jan}) = 5 \]

\[ h_0(\text{feb}) = 1 \quad h_1(\text{feb}) = 2 \quad h_2(\text{feb}) = 5 \]
Random Hypergraphs (r-graphs)

- 3-graph:

\[ \begin{align*}
  h_0(\text{jan}) &= 1 \quad h_1(\text{jan}) = 3 \quad h_2(\text{jan}) = 5 \\
  h_0(\text{feb}) &= 1 \quad h_1(\text{feb}) = 2 \quad h_2(\text{feb}) = 5 \\
  h_0(\text{mar}) &= 0 \quad h_1(\text{mar}) = 3 \quad h_2(\text{mar}) = 4
\end{align*} \]

- 3-graph is induced by three uniform hash functions
- Our best result uses 3-graphs
The MPHF Proposed by Czech, Havas e Majewski ...

MPHF by Czech, Havas and Majewski
MPHF by Czech, Havas and Majewski

\[ S \]
\[
\begin{array}{c}
\text{jan} \\
\text{feb} \\
\text{mar} \\
\text{apr}
\end{array}
\]

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3
\end{array}
\]

\[
\begin{array}{c}
4 \\
5 \\
6 \\
7
\end{array}
\]

\[
\begin{array}{c}
\text{mar} \\
\text{jan} \\
\text{feb} \\
\text{apr}
\end{array}
\]

\[
\begin{align*}
h_0(\text{jan}) &= 2 & h_1(\text{feb}) &= 5 \\
h_0(\text{feb}) &= 2 & h_1(\text{feb}) &= 6 \\
h_0(\text{mar}) &= 0 & h_1(\text{mar}) &= 5 \\
h_0(\text{apr}) &= 2 & h_1(\text{feb}) &= 7
\end{align*}
\]
Acyclic 2-graph

\[ G_r; \]

\[ L:Ø \]

0 1 2 3 \( h_0 \)

4 5 6 7 \( h_1 \)

mar jan feb apr
Acyclic 2-graph

\[ G_r : \]

\[ L : \{0,5\} \]
Acyclic 2-graph

$G_r$: 

$0 \quad 1 \quad 2 \quad 3 \quad h_0$

$4 \quad 5 \quad 6 \quad 7 \quad h_1$

L: \{0,5\} \{2,6\}
Acyclic 2-graph
Acyclic 2-graph

$G_r$: 0 1 2 3 $h_0$

$G_r$ is acyclic

4 5 6 7 $h_1$
MPHF by Czech, Havas and Majewski (1992)

$S$

$G_r$: 

0 → 1 → 2 → 3 → $h_0$

4 → 5 → 6 → 7 → $h_1$

Mapping

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
MPHF by Czech, Havas and Majewski (1992)

L: \{0,5\} \{2,6\} \{2,7\} \{2,5\}
MPHF by Czech, Havas and Majewski (1992)

\[ g[2] = N = 4 \]

\[ g[5] = i_a - g[2] \mod N = 0 - 4 \mod 4 = 0 \]
MPHF by Czech, Havas and Majewski (1992)

$S$

Jan  Feb  Mar  Apr

Mapping

$G_r$:  

0  1  2  3  $h_0$

Mar  Jan  Feb  Apr

Assigning

$L$:  

0  1  2  3

\{0,5\}  \{2,6\}  \{2,7\}  \{2,5\}

$g$

0  1  2  3  4  5  6  7

-1  -1  4  -1  -1  0  -1  3

$g[2] = N = 4$

$g[7] = i_a - g[2] \mod N = 3 - 4 \mod 4 = 3$
MPHF by Czech, Havas and Majewski (1992)
MPHF by Czech, Havas and Majewski (1992)

\[
\text{Phf}(\text{feb}) = (g[h_0(\text{feb})] + g[h_1(\text{feb})]) \mod N = (g[2] + g[6]) \mod 4 = 1
\]

Order is preserved
MPHF by Czech, Havas and Majewski (1992)

Mapping: $G_f$: $h_0$

Assigning: $h_1$

$\log(N)$ bits for each entry
The Internal Random Access Memory Algorithm ...
(from log(N) to O(1) bits)

Acyclic 2-graph

$G_r$: $h_0$

L: $\emptyset$

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin) 47
Acyclic 2-graph

$G_r$: $h_0$

$L$: $\{0,5\}$
Acyclic 2-graph

$G_r:\ h_0 \quad h_1$

$\{0,5\} \quad \{2,6\}$

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
Acyclic 2-graph

\[ G_r: \]

0 \quad 1 \quad 2 \quad 3 \quad h_0

4 \quad 5 \quad 6 \quad 7 \quad h_1

L:

0 \quad \{0,5\} \quad 1 \quad \{2,6\} \quad 2 \quad \{2,7\}
Acyclic 2-graph

$G_r$ is acyclic

$G_r$: 

\begin{align*}
0 & \quad 1 & \quad 2 & \quad 3 & h_0 \\
4 & \quad 5 & \quad 6 & \quad 7 & h_1
\end{align*}

$\text{L: } \{0,5\} \{2,6\} \{2,7\} \{2,5\}$
Internal Random Access Memory Algorithm (r=2)
Internal Random Access Memory Algorithm (r=2)

\[ S \]
\[ \text{Jan} \quad \text{Feb} \quad \text{Mar} \quad \text{Apr} \]

\[ \text{Mapping} \]

\[ G_r: \]

\[ 0 \quad 1 \quad 2 \quad 3 \quad h_0 \]
\[ 4 \quad 5 \quad 6 \quad 7 \quad h_1 \]
Internal Random Access Memory Algorithm (r=2)

\[ G_r : \]
\[
\begin{array}{c}
0 & 1 & 2 & 3 \\
\text{mar} & \text{jan} & \text{feb} & \text{apr} \\
4 & 5 & 6 & 7 \\
\end{array}
\]

Mapping: \( S \)

Assigning: \( g \)

\( L : \{0,5\} \{2,6\} \{2,7\} \{2,5\} \)

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
Internal Random Access Memory Algorithm (r=2)

Mapping

Assigning

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
Internal Random Access Memory Algorithm (r=2)

Mapping:

Assigning:

\[ g \]

\[
\begin{array}{c}
0 & 1 & 2 & 3 \\
r & r & 0 & 1 \\
\end{array}
\]

\[ L: \{0,5\} \{2,6\} \{2,7\} \{2,5\} \]
Internal Random Access Memory Algorithm (r=2)

Mapping

Assigning
Internal Random Access Memory Algorithm (r=2)

\[ i = \left( g[h_0(\text{feb})] + g[h_1(\text{feb})] \right) \mod r = (g[2] + g[6]) \mod 2 = 1 \]
Internal Random Access Memory Algorithm: PHF

\[ i = (g[h_0(feb)] + g[h_1(feb)]) \mod r = (g[2] + g[6]) \mod 2 = 1 \]

\[ \text{phf}(feb) = h_{i=1}(feb) = 6 \]
Internal Random Access Memory Algorithm: MPHF

\[ i = (g[h_0(feb)] + g[h_1(feb)]) \mod r = (g[2] + g[6]) \mod 2 = 1 \]

\[ \text{phf}(\text{feb}) = h_{i=1}(\text{feb}) = 6 \]

\[ \text{mphf}(\text{feb}) = \text{rank}(\text{phf}(\text{feb})) = \text{rank}(6) = 2 \]
Space to Represent the Function

Order is not preserved

2 bits for each entry

LATIN - LABoratory for Treating INformation (www.dcc.ufmg.br/latin)
Space to Represent the Functions ($r = 3$)

- **PHF** $g: [0, m-1] \rightarrow \{0, 1, 2\}$
  - $m = cn$ bits, $c = 1.23 \rightarrow 2.46n$ bits
  - $(\log 3)cn$ bits, $c = 1.23 \rightarrow 1.95n$ bits (arith. coding)
  - Optimal: $0.89n$ bits

- **MPHF** $g: [0, m-1] \rightarrow \{0, 1, 2, 3\}$ (ranking info required)
  - $2m + \varepsilon m = (2 + \varepsilon)cn$ bits
  - For $c = 1.23$ and $\varepsilon = 0.125 \rightarrow 2.62n$ bits
  - Optimal: $1.44n$ bits.
Use of Acyclic Random Hypergraphs

- Sufficient condition to work
- Repeatedly selects $h_0, h_1, ..., h_{r-1}$
- For $r = 2$, $m = cn$ and $c \geq 2.09$: $Pr_a = 0.29$
- For $r = 3$, $m = cn$ and $c \geq 1.23$: $Pr_a$ tends to 1
- Number of iterations is $1/Pr_a$
  - $r = 2$: 3.5 iterations
  - $r = 3$: 1.0 iteration
The External Memory Cache-Aware Algorithm ...
External Memory Algorithm (EM)

- First MPHF algorithm for very large key sets (in the order of billions of keys)

- This is possible because
  - Deals with external memory efficiently (cache-aware)
  - Generates compact functions (near space-optimal)
  - Uses a little amount of internal memory to scale
  - Works in linear time

- Two implementations:
  - Theoretical well-founded EM (uses uniform hashing)
  - Heuristic EM (uses universal hashing)
External Memory Algorithm (EM)

\[ \text{MPHF}(x) = \text{MPHF}_i(x) + \text{offset}[i]; \]
Key Set Does Not Fit In Internal Memory

Partitioning

Key Set $S$ of $\beta$ bytes

$\mu$ bytes of Internal memory

$\mu$ bytes of Internal memory

$N = \beta/\mu$

$b =$ Number of bits of each bucket address

Each bucket $\leq 256$
Important Design Decisions

- We map long URLs to a fingerprint of fixed size using a hash function.
- Use our RAM linear time and near-optimal space algorithm to generate the MPHF of each bucket.
- How do we obtain a linear time complexity?
  - Using internal radix sorting to form the buckets.
  - Using a heap of N entries to drive a N-way merge that reads the buckets from disk in one pass.
Use the Internal Random Access Memory Algorithm for Each Bucket

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
Why the EM Algorithm is Well-Founded?

First Point:

Pool of uniform hash function on each bucket

Sharing

h_{i0} h_{i1} h_{i2}  h_{i0} h_{i1} h_{i2}

0 1 2 2^b - 1

Buckets
Why the EM Algorithm is Well-Founded?

Second Point:

We have shown how to create that pool based on the linear hash functions proposed by Alon et al (JACM 1999)

\[
f(x, s, \Delta) = \left( \sum_{j=1}^{k} t_j [y_j(x) \oplus \Delta] + s \sum_{j=k+1}^{2k} t_j [y_{j-k}(x) \oplus \Delta] \right) \mod p
\]

\[
h_{i0}(x) = f(x, s, 0) \mod |B_i|
\]

\[
h_{i1}(x) = f(x, s, 1) \mod |B_i| + |B_i|
\]

\[
h_{i2}(x) = f(x, s, 2) \mod |B_i| + 2|B_i|
\]
Why the EM Algorithm is Well-Founded?

Second Point:

We have shown how to create that pool based on the linear hash functions proposed by Alon et al (JACM 1999)

\[ f(x, s, \Delta) = \left( \sum_{j=1}^{k} t_j [y_j(x) \oplus \Delta] + s \sum_{j=k+1}^{2k} t_j [y_{j-k}(x) \oplus \Delta] \right) \mod p \]

\[ h_{i0}(x) = f(x, s, 0) \mod |B_i| \]

\[ h_{i1}(x) = f(x, s, 1) \mod |B_i| + |B_i| \]

\[ h_{i2}(x) = f(x, s, 2) \mod |B_i| + 2|B_i| \]
Why the EM Algorithm is Well-Founded?

Second Point:

We have shown how to create that pool based on the linear hash functions proposed by Alon et al (JACM 1999)

\[
f(x, s, \Delta) = \left( \sum_{j=1}^{k} t_j \left[ y_j(x) \oplus \Delta \right] + s \sum_{j=k+1}^{2k} t_j \left[ y_{j-k}(x) \oplus \Delta \right] \right) \mod p
\]

\[
h_{i0}(x) = f(x, s, 0) \mod |B_i|
\]

\[
h_{i1}(x) = f(x, s, 1) \mod |B_i| + |B_i|
\]

\[
h_{i2}(x) = f(x, s, 2) \mod |B_i| + 2|B_i|
\]
Why the EM Algorithm is Well-Founded?

Second Point:

We have shown how to create that pool based on the linear hash functions proposed by Alon et al (JACM 1999)

\[ f(x, s, \Delta) = \left( \sum_{j=1}^{k} t_j [y_j(x) \oplus \Delta] + s \sum_{j=k+1}^{2k} t_j [y_{j-k}(x) \oplus \Delta] \right) \text{mod } p \]

Computed by a linear hash function

\[ h_{i0}(x) = f(x, s, 0) \text{mod}|B_i| \]
\[ h_{i1}(x) = f(x, s, 1) \text{mod}|B_i| + |B_i| \]
\[ h_{i2}(x) = f(x, s, 2) \text{mod}|B_i| + 2|B_i| \]
Why the EM Algorithm is Well-Founded?

Second Point:

Computing fingerprints of 128 bits with the linear hash functions

\[ h'(x) = 100101011110011011010000111000110 \]
\[ h_0(x) = h'(x)[96,127] \gg (32 - b) \]
\[ y_6(x) = h'(x)[80,95] \]
\[ \cdot \]
\[ \cdot \]
\[ \cdot \]
\[ y_1(x) = h'(x)[0,15] \]
Why the EM Algorithm is Well-Founded?

Third Point:
How to keep maximum bucket size smaller than $l = 256$?

\[
b \leq \log(n) - \log(l/\log(l)) + O(1)
\]
\[
l \geq \log n \log \log n
\]
The Heuristic EM Algorithm

- Uses a universal pseudo random hash function proposed by Jenkins (1997):
  - Faster to compute
  - Requires just one random integer number as seed
Experimental Results

- **Metrics:**
  - Generation time
  - Storage space
  - Evaluation time

- **Collection:**
  - 1.024 billions of URLs collected from the web
  - 64 bytes long on average

- **Experiments**
  - Commodity PC with a cache of 4 Mbytes
  - 1.86 GHz, 1 GB, Linux, 64 bits architecture
### Generation Time of MPHFs (in Minutes)

<table>
<thead>
<tr>
<th>n (millions)</th>
<th>32</th>
<th>128</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Theoretic EM</strong></td>
<td>1.3 ± 0.002</td>
<td>6.2 ± 0.02</td>
<td>27.6 ± 0.09</td>
<td>57.4 ± 0.06</td>
</tr>
<tr>
<td><strong>Heuristic EM</strong></td>
<td>0.95 ± 0.02</td>
<td>5.1 ± 0.01</td>
<td>22.0 ± 0.13</td>
<td>46.2 ± 0.06</td>
</tr>
</tbody>
</table>
Related Algorithms

- Botelho, Kohayakawa, Ziviani (2005) - BKZ
- Fox, Chen and Heath (1992) – FCH
- Czech, Havas and Majewski (1992) – CHM
- Pagh (1999) - PAGH

All algorithms coded in the same framework
## Generation Time

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Generation Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM ((r = 3))</td>
<td>6.7 ± 0</td>
</tr>
<tr>
<td>Theoretic EM</td>
<td>9.0 ± 0.3</td>
</tr>
<tr>
<td>Heuristic EM</td>
<td>6.4 ± 0.3</td>
</tr>
<tr>
<td>BKZ</td>
<td>12.8 ± 1.6</td>
</tr>
<tr>
<td>CHM</td>
<td>17.0 ± 3.2</td>
</tr>
<tr>
<td>FCH</td>
<td>2,400.1 ± 711.6</td>
</tr>
<tr>
<td>PAGH</td>
<td>42.8 ± 2.4</td>
</tr>
</tbody>
</table>

3,541,615 URLs
# Generation Time and Storage Space

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Generation Time (sec)</th>
<th>Space (bits/key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM (r = 3)</td>
<td>6.7 ± 0</td>
<td>2.6</td>
</tr>
<tr>
<td>Theoretic EM</td>
<td>9.0 ± 0.3</td>
<td>3.3</td>
</tr>
<tr>
<td>Heuristic EM</td>
<td>6.4 ± 0.3</td>
<td>3.1</td>
</tr>
<tr>
<td>BKZ</td>
<td>12.8 ± 1.6</td>
<td>21.8</td>
</tr>
<tr>
<td>CHM</td>
<td>17.0 ± 3.2</td>
<td>45.5</td>
</tr>
<tr>
<td>FCH</td>
<td>2,400.1 ± 711.6</td>
<td>4.2</td>
</tr>
<tr>
<td>PAGH</td>
<td>42.8 ± 2.4</td>
<td>44.2</td>
</tr>
</tbody>
</table>

3,541,615 URLs
## Generation Time, Storage Space and Evaluation Time

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Generation Time (sec)</th>
<th>Space (bits/key)</th>
<th>Evaluation time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM (r = 3)</td>
<td>6.7 ± 0</td>
<td>2.6</td>
<td>2.1</td>
</tr>
<tr>
<td>Theoretic EM</td>
<td>9.0 ± 0.3</td>
<td>3.3</td>
<td>4.9</td>
</tr>
<tr>
<td>Heuristic EM</td>
<td>6.4 ± 0.3</td>
<td>3.1</td>
<td>2.7</td>
</tr>
<tr>
<td>BKZ</td>
<td>12.8 ± 1.6</td>
<td>21.8</td>
<td>2.3</td>
</tr>
<tr>
<td>CHM</td>
<td>17.0 ± 3.2</td>
<td>45.5</td>
<td>2.3</td>
</tr>
<tr>
<td>FCH</td>
<td>2,400.1 ± 711.6</td>
<td>4.2</td>
<td>1.7</td>
</tr>
<tr>
<td>PAGH</td>
<td>42.8 ± 2.4</td>
<td>44.2</td>
<td>2.3</td>
</tr>
</tbody>
</table>

3,541,615 URLs  
Key length = 64 bytes
Why to build a library?

- Lack of similar libraries in the free software community
- Test the applicability of our algorithm out there

Feedbacks:

- 3,069 downloads (until March 16th, 2009)
- Incorporated by Debian

Library address: [http://cmph.sourceforge.net](http://cmph.sourceforge.net)
Comparing Our MPHF With Other Methods

- Open Addressing
- Chaining
Open Addressing

- Items Stored in a contiguous array
- Empty entries are used to resolve collisions
- Methods used in comparison
  - Linear Hashing
  - Quadratic Hashing
  - Double Hashing
  - Cuckoo Hashing
  - Hopscotch Hashing
  - Sparse Hashing
Chaining

- Use linked lists to resolve collisions
- New items can always be added
- Methods used in comparison
  - Chaining with move to front heuristics
  - Exact fit
  - Exact fit with move to front heuristics
## Vocabularies

<table>
<thead>
<tr>
<th>Collection</th>
<th>n</th>
<th>Shortest Key</th>
<th>Largest Key</th>
<th>Avg Key Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllTheWeb</td>
<td>5,424,923</td>
<td>2</td>
<td>31</td>
<td>17.46</td>
</tr>
<tr>
<td>URLs-37</td>
<td>37,294,116</td>
<td>8</td>
<td>496</td>
<td>58.77</td>
</tr>
</tbody>
</table>
Successful and Unsuccessful Searches

<table>
<thead>
<tr>
<th>Collection</th>
<th>$n$</th>
<th>Average Key Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllTheWeb</td>
<td>10,000,000</td>
<td>17.46</td>
</tr>
<tr>
<td>URLs-37</td>
<td>250,000,000</td>
<td>58.77</td>
</tr>
</tbody>
</table>

- **Successful searches**
  - Follow real access patterns (power law)
  - Represents the best case

- **Unsuccessful searches**
  - Randomly generated keys
  - Uniform distribution

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
### MPH vs Linear, Quadratic and Double Hashing

#### Successful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>α(%)</td>
<td>T(s)</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
</tr>
<tr>
<td>LH</td>
<td>20</td>
<td>3.38</td>
</tr>
<tr>
<td>QH</td>
<td>20</td>
<td>3.41</td>
</tr>
<tr>
<td>DH</td>
<td>20</td>
<td>3.46</td>
</tr>
</tbody>
</table>
Successful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>α(%)</td>
<td>T(s)</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
</tr>
<tr>
<td>LH</td>
<td>20</td>
<td>3.38</td>
</tr>
<tr>
<td>QH</td>
<td>20</td>
<td>3.41</td>
</tr>
<tr>
<td>DH</td>
<td>20</td>
<td>3.46</td>
</tr>
</tbody>
</table>
### MPH vs Linear, Quadratic and Double Hashing

#### Successful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>α(%)</td>
<td>T(s)</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
</tr>
<tr>
<td>LH</td>
<td>20</td>
<td>3.38</td>
</tr>
<tr>
<td>QH</td>
<td>20</td>
<td>3.41</td>
</tr>
<tr>
<td>DH</td>
<td>20</td>
<td>3.46</td>
</tr>
</tbody>
</table>
## MPH vs Linear, Quadratic and Double Hashing

### Unsuccessful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\alpha$ (%)</td>
<td>$T(s)$</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.23</td>
</tr>
<tr>
<td>LH</td>
<td>20</td>
<td>3.92</td>
</tr>
<tr>
<td>QH</td>
<td>20</td>
<td>3.79</td>
</tr>
<tr>
<td>DH</td>
<td>20</td>
<td>3.85</td>
</tr>
</tbody>
</table>
MPHF vs Cuckoo Hashing

Successful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th></th>
<th></th>
<th></th>
<th>URLs-37</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>α(%)</td>
<td>T(s)</td>
<td>S₀</td>
<td></td>
<td>α(%)</td>
<td>T(s)</td>
<td>S₀</td>
<td></td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
<td>2.62</td>
<td></td>
<td>100</td>
<td>261.50</td>
<td>2.62</td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>50</td>
<td>4.06</td>
<td>128</td>
<td></td>
<td>50</td>
<td>264.54</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>40</td>
<td>3.92</td>
<td>160</td>
<td></td>
<td>40</td>
<td>263.86</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>30</td>
<td>3.76</td>
<td>213</td>
<td></td>
<td>30</td>
<td>263.30</td>
<td>213</td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>20</td>
<td>3.67</td>
<td>320</td>
<td></td>
<td>20</td>
<td>263.20</td>
<td>320</td>
<td></td>
</tr>
</tbody>
</table>
## MPHF vs Cuckoo Hashing

### Successful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\alpha$ (%)</td>
<td>$T(s)$</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
</tr>
<tr>
<td>CH</td>
<td>50</td>
<td>4.06</td>
</tr>
<tr>
<td>CH</td>
<td>40</td>
<td>3.92</td>
</tr>
<tr>
<td>CH</td>
<td>30</td>
<td>3.76</td>
</tr>
<tr>
<td>CH</td>
<td>20</td>
<td>3.67</td>
</tr>
</tbody>
</table>
## MPH vs Cuckoo Hashing

### Unsuccessful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th></th>
<th></th>
<th>URLs-37</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>α(%)</td>
<td>T(s)</td>
<td>S₀</td>
<td>α(%)</td>
<td>T(s)</td>
<td>S₀</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.23</td>
<td>2.62</td>
<td>100</td>
<td>271.42</td>
<td>2.62</td>
</tr>
<tr>
<td>CH</td>
<td>50</td>
<td>5.13</td>
<td>128</td>
<td>50</td>
<td>281.23</td>
<td>128</td>
</tr>
<tr>
<td>CH</td>
<td>40</td>
<td>4.99</td>
<td>160</td>
<td>40</td>
<td>275.80</td>
<td>160</td>
</tr>
<tr>
<td>CH</td>
<td>30</td>
<td>4.80</td>
<td>213</td>
<td>30</td>
<td>273.02</td>
<td>213</td>
</tr>
<tr>
<td>CH</td>
<td>20</td>
<td>4.70</td>
<td>320</td>
<td>20</td>
<td>272.52</td>
<td>320</td>
</tr>
</tbody>
</table>
MPHF vs Hopscotch, Sparse and Dense Hashing

Successful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>α(%)</td>
<td>T(s)</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
</tr>
<tr>
<td>HoH</td>
<td>65</td>
<td>3.32</td>
</tr>
<tr>
<td>SH</td>
<td>65</td>
<td>3.95</td>
</tr>
<tr>
<td>DeH</td>
<td>65</td>
<td>2.88</td>
</tr>
</tbody>
</table>
### MPH vs Hopscotch, Sparse and Dense Hashing

#### Successful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>α(%)</td>
<td>T(s)</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
</tr>
<tr>
<td>HoH</td>
<td>65</td>
<td>3.32</td>
</tr>
<tr>
<td>SH</td>
<td>65</td>
<td>3.95</td>
</tr>
<tr>
<td>DeH</td>
<td>65</td>
<td>2.88</td>
</tr>
</tbody>
</table>
MPHF vs Hopscotch, Sparse and Dense Hashing

Successful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\alpha$</td>
<td>T(s)</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
</tr>
<tr>
<td>HoH</td>
<td>65</td>
<td>3.32</td>
</tr>
<tr>
<td>SH</td>
<td>65</td>
<td>3.95</td>
</tr>
<tr>
<td>DeH</td>
<td>65</td>
<td>2.88</td>
</tr>
</tbody>
</table>
MPHF vs Hopscotch, Sparse and Dense Hashing

Successful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>α(%)</td>
<td>T(s)</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
</tr>
<tr>
<td>HoH</td>
<td>65</td>
<td>3.32</td>
</tr>
<tr>
<td>SH</td>
<td>65</td>
<td>3.95</td>
</tr>
<tr>
<td>DeH</td>
<td>65</td>
<td>2.88</td>
</tr>
</tbody>
</table>
Unsuccessful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th></th>
<th></th>
<th>URLs-37</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\alpha(%))</td>
<td>(T(s))</td>
<td>(S_o)</td>
<td>(\alpha(%))</td>
<td>(T(s))</td>
<td>(S_o)</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.23</td>
<td>2.62</td>
<td>100</td>
<td>271.42</td>
<td>2.62</td>
</tr>
<tr>
<td>HoH</td>
<td>65</td>
<td>3.88</td>
<td>83.69</td>
<td>56</td>
<td>271.51</td>
<td>107.43</td>
</tr>
<tr>
<td>SH</td>
<td>65</td>
<td>4.60</td>
<td>4.10</td>
<td>56</td>
<td>271.76</td>
<td>4.76</td>
</tr>
<tr>
<td>DeH</td>
<td>65</td>
<td>3.84</td>
<td>98.46</td>
<td>56</td>
<td>271.75</td>
<td>114.29</td>
</tr>
</tbody>
</table>
## MPHF vs Chaining Methods

Sucesfull searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\alpha$ (%)</td>
<td>$T(s)$</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.07</td>
</tr>
<tr>
<td>CHMTF</td>
<td>40</td>
<td>3.56</td>
</tr>
<tr>
<td>EFH</td>
<td>100</td>
<td>3.71</td>
</tr>
<tr>
<td>EFHMTF</td>
<td>200</td>
<td>3.64</td>
</tr>
</tbody>
</table>
## MPH vs Chaining Methods

### Unsuccessful searches

<table>
<thead>
<tr>
<th>Data Struc.</th>
<th>AllTheWeb</th>
<th>URLs-37</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\alpha$(%)</td>
<td>$T(s)$</td>
</tr>
<tr>
<td>MPH</td>
<td>100</td>
<td>2.23</td>
</tr>
<tr>
<td>CHMTF</td>
<td>40</td>
<td>4.02</td>
</tr>
<tr>
<td>EFH</td>
<td>40</td>
<td>4.09</td>
</tr>
<tr>
<td>EFHMTF</td>
<td>40</td>
<td>4.06</td>
</tr>
</tbody>
</table>
Conclusions

- Three implementations were developed:
  - Theoretic EM (external memory)
  - Heuristic EM (external memory)
  - RAM (internal memory, used in the EM algorithm)

- Near-optimal space functions in linear time

- Function evaluation in time $O(1)$

- First theoretically well-founded algorithm that is practical and will work for every key set from $U$ with high probability
Conclusions

- MPHFs provide a gain of $O(\log n)$ bits when compared to other hashing methods
- MPHFs benefits from cache effects
- MPHFs lookup times are of the same order or smaller
Parallel Version of the EM Algorithm

<table>
<thead>
<tr>
<th>PCs</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.8</td>
<td>3.5</td>
<td>7.0</td>
<td>8.7</td>
<td>12.2</td>
</tr>
</tbody>
</table>

1 billion URLs using 14 PCs in 5 minutes
References


- F.C. Botelho, R. Pagh and N. Ziviani, Simple and Space-Efficient Minimal Perfect Hash Functions *10th International Workshop on Algorithms and Data Structures (WADS07), Halifax, Canada, August 2007*, 139-150.