Near Space-Optimal Perfect Hashing Algorithm

Nivio Ziviani

Fabiano C. Botelho

Department of Computer Science
Federal University of Minas Gerais, Brazil

International Conference on the Analysis of Algorithms
Maresias, Brazil, April 17, 2008

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
  - **I**nternal **R**andom **A**ccess memory algorithm (**IRA**)
- Key set larger than the internal memory
  - **E**xternal **C**ache-**A**ware memory algorithm (**ECA**)

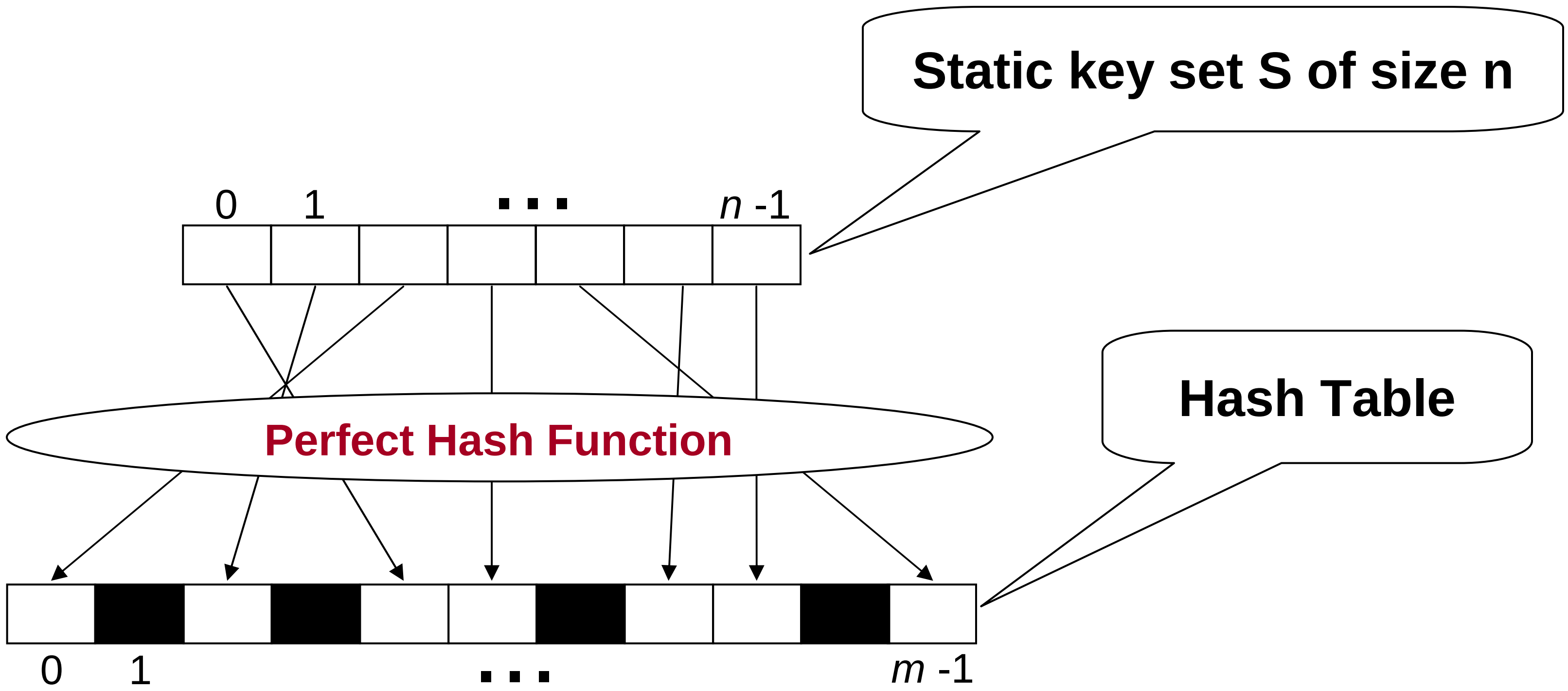
Objective of the Presentation

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
  - **I**nternal **R**andom **A**ccess memory algorithm (**IRA**)
- Key set larger than the internal memory
  - **E**xternal **C**ache-**A**ware memory algorithm (**ECA**)

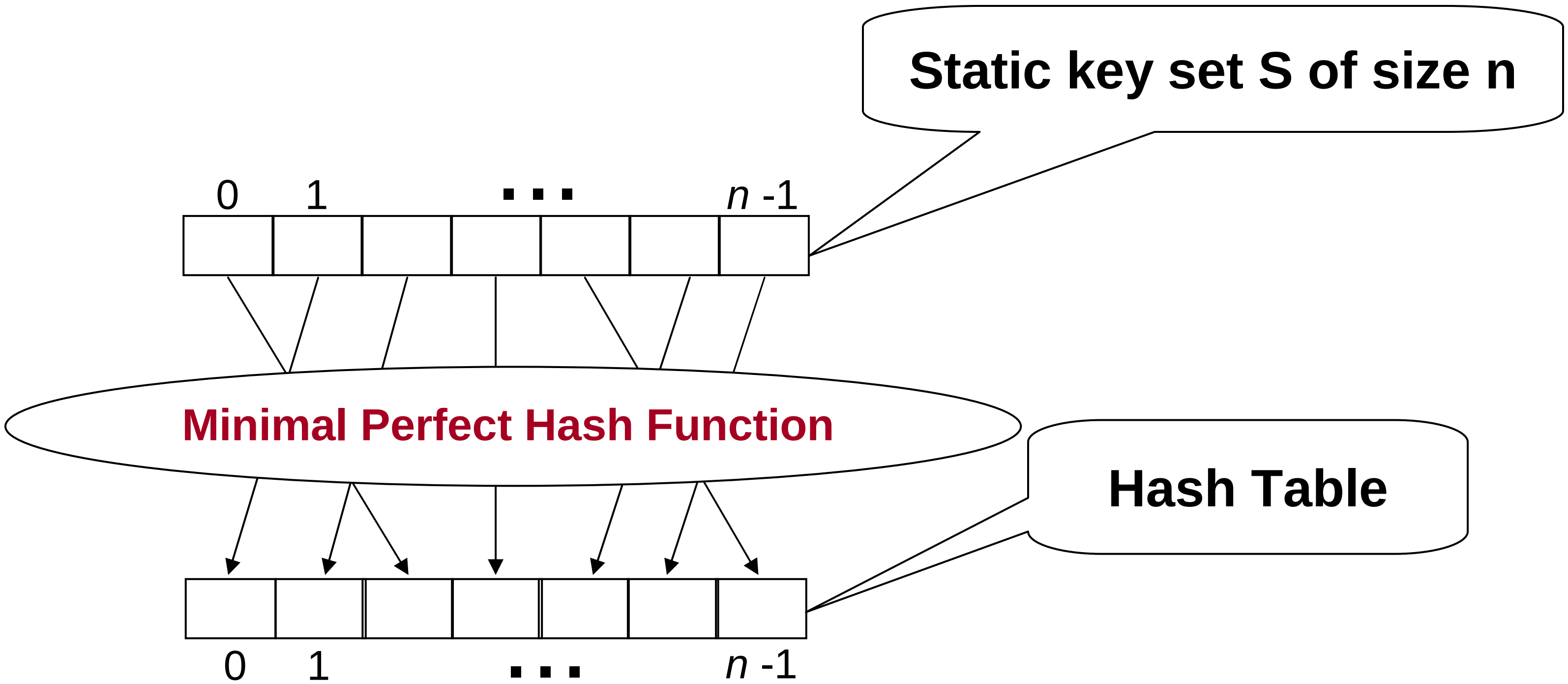
Theoretically well-founded, time efficient, higly scalable and near space-optimal

Perfect Hash Function



$$S \subseteq U, \text{ where } |U| = u$$

Minimal Perfect Hash Function

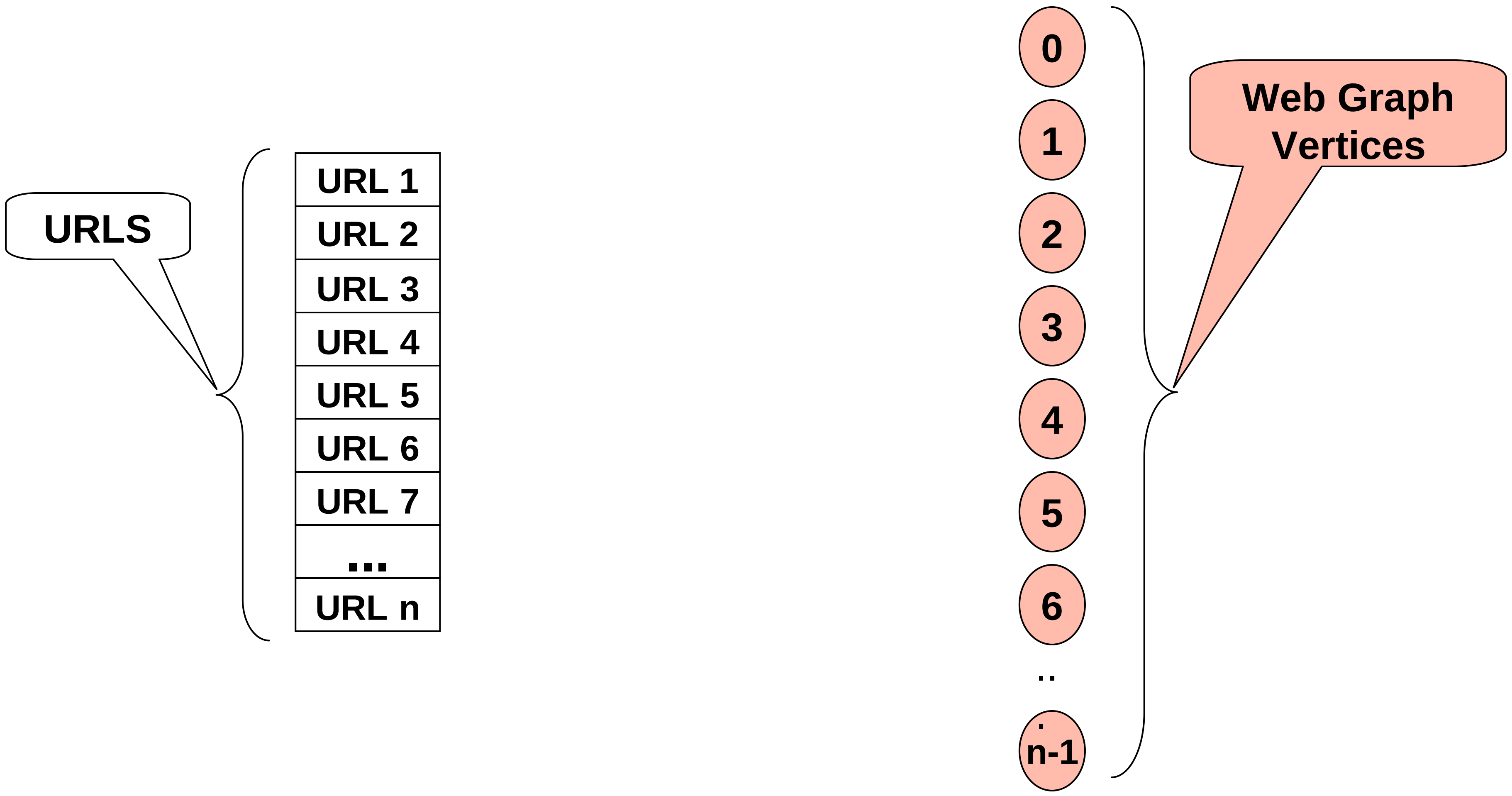


$$S \subseteq U, \text{ where } |U| = u$$

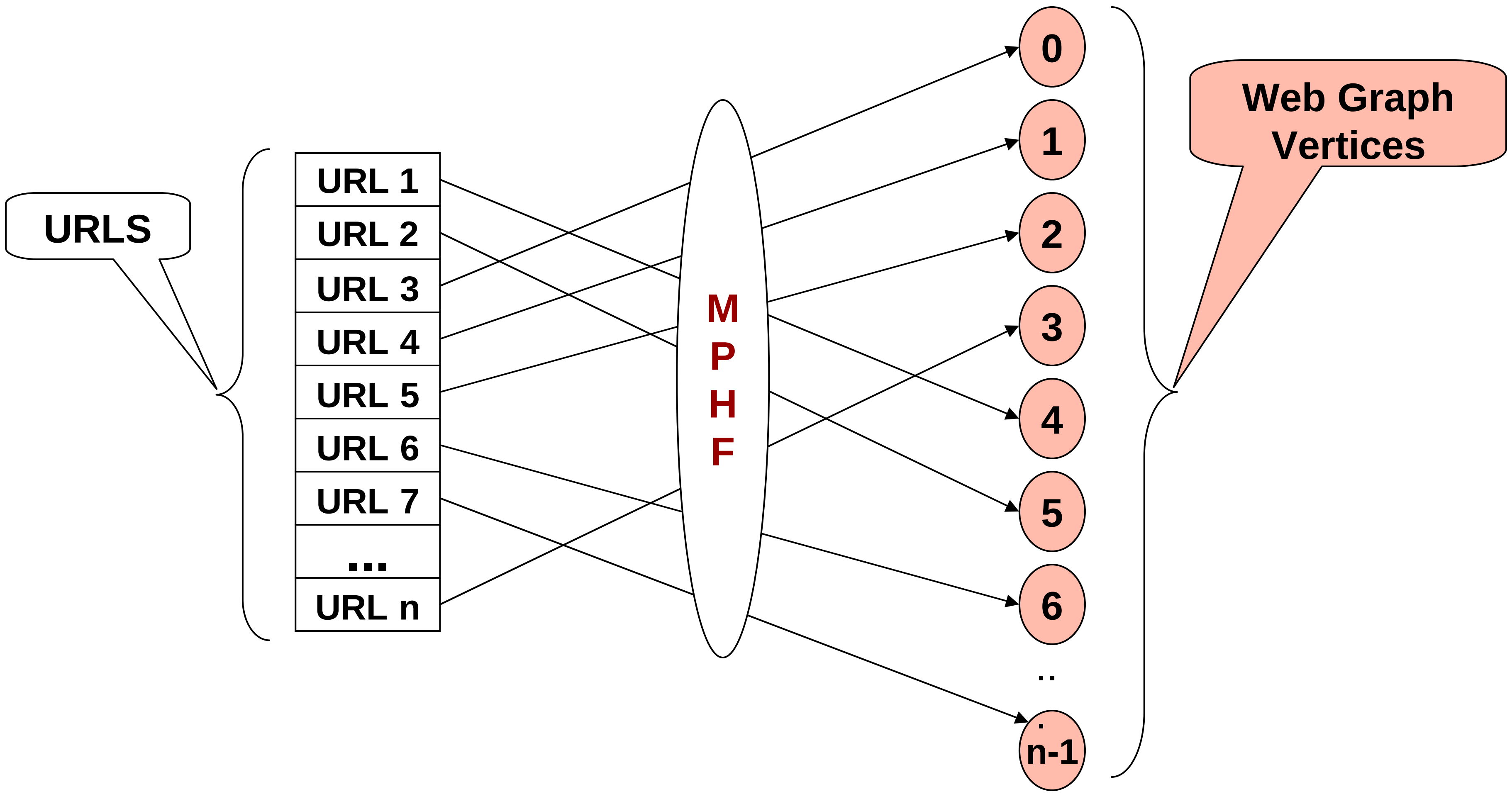
Where to use a PHF or a MPHF?

- Access items based on the value of a key is ubiquitous in Computer Science
- Work with huge static item 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

Mapping URLs to Web Graph Vertices



Mapping URLs to Web Graph Vertices



Information Theoretical Lower Bounds for Storage Space

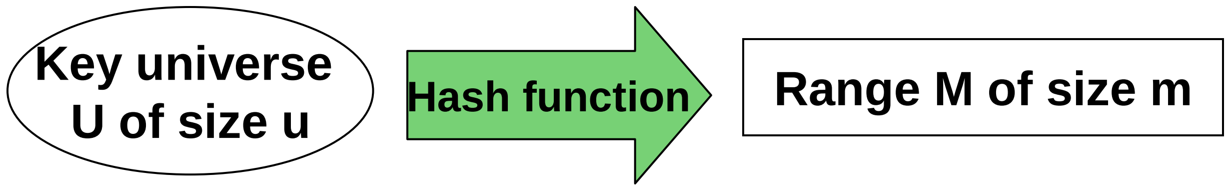
- PHFs (m ≈ n): $\text{Storage Space} \geq \frac{n^2}{m} \log e$

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

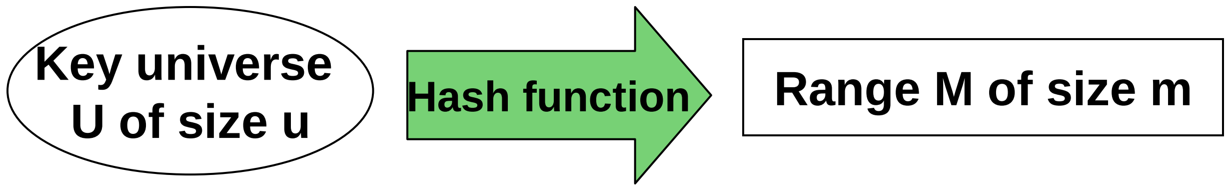
m < 3n

$\log e \approx 1.4427$

Uniform Hashing Versus Universal Hashing



Uniform Hashing Versus Universal Hashing



Uniform hashing

- # of functions from U to M?

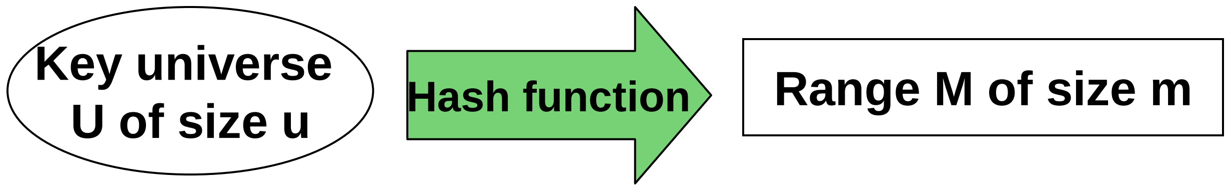
$$m^u$$

- # of bits to encode each fucntion

$$u \log m$$

- Independent functions with values uniformly distributed

Uniform Hashing Versus Universal Hashing



Uniform hashing

- # of functions from U to M?

$$m^u$$

- # of bits to encode each fucntion

$$u \log m$$

- 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 ½, 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

Work	Gen. Time	Eval. Time	Size (bits)
Mehlhorn (1984)	Expon.	Expon.	O(n)
Schmidt and Siegel (1990)	Not analyzed	O(1)	O(n)
Hagerup and Tholey (2001)	O(n+log log u)	O(1)	O(n)

Theoretical Results – Use Uniform Hashing

Work	Gen. Time	Eval. Time	Size (bits)
Mehlhorn (1984)	Expon.	Expon.	O(n)
Schmidt & Siegel (1990)	Not analyzed	O(1)	O(n)
Hagerup & Tholey (2001)	O(n+log log u)	O(1)	O(n)
Theoretic ECA (CIKM 2007)	**O(n)**	**O(1)**	**O(n)**

Practical Results – Assume Uniform Hashing For Free

Work	Gen. Time	Eval. Time	Size (bits)
Czech, Havas & Majewski (1992)	**O(n)**	**O(1)**	**O(n log n)**
Majewski, Wormald, Havas & Czech (1996)	**O(n)**	**O(1)**	**O(n log n)**
Pagh (1999)	**O(n)**	**O(1)**	**O(n log n)**
Botelho, Kohayakawa, & Ziviani (2005)	**O(n)**	**O(1)**	**O(n log n)**

Practical Results – Assume Uniform Hashing For Free

Work	Gen. Time	Eval. Time	Size (bits)
Czech, Havas & Majewski (1992)	**O(n)**	**O(1)**	**O(n log n)**
Majewski, Wormald, Havas & Czech (1996)	**O(n)**	**O(1)**	**O(n log n)**
Pagh (1999)	**O(n)**	**O(1)**	**O(n log n)**
Botelho, Kohayakawa, & Ziviani (2005)	**O(n)**	**O(1)**	**O(n log n)**
IRA (WADS 2007)	**O(n)**	**O(1)**	**O(n)**

Practical Results – Assume Uniform Hashing For Free

Work	Gen. Time	Eval. Time	Size (bits)
Czech, Havas & Majewski (1992)	**O(n)**	**O(1)**	**O(n log n)**
Majewski, Wormald, Havas & Czech (1996)	**O(n)**	**O(1)**	**O(n log n)**
Pagh (1999)	**O(n)**	**O(1)**	**O(n log n)**
Botelho, Kohayakawa, & Ziviani (2005)	**O(n)**	**O(1)**	**O(n log n)**
IRA (WADS 2007)	**O(n)**	**O(1)**	**O(n)**
Heuristic ECA (CIKM 2007)	**O(n)**	**O(1)**	**O(n)**

Empirical Results

Work	Application	Gen. Time	Eval. Time	Size (bits)
Fox, Chen & Heath (1992)	**Index data in CD-ROM**	**Exp.**	**O(1)**	**O(n)**
Lefebvre & Hoppe (2006)	**Sparse spatial data**	**O(n)**	**O(1)**	**O(n)**
Chang, Lin & Chou (2005, 2006)	**Data mining**	**O(n)**	**O(1)**	**Not analyzed**

Internal Random Access and External Cache-Aware Memory Algorithms

- Near space optimal
- 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 all (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

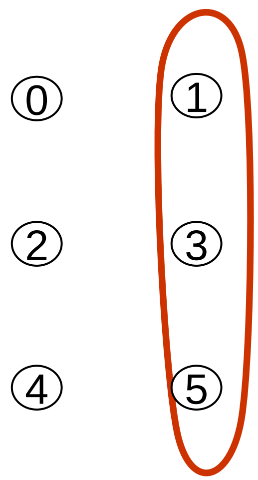
2 3

4 5

- 3-graph is induced by three uniform hash functions

Random Hypergraphs (r-graphs)

- 3-graph:

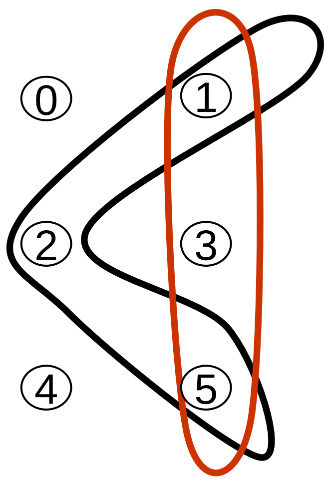


$h_0(jan) = 1 \quad h_1(jan) = 3 \quad h_2(jan) = 5$

- 3-graph is induced by three uniform hash functions

Random Hypergraphs (r-graphs)

- 3-graph:



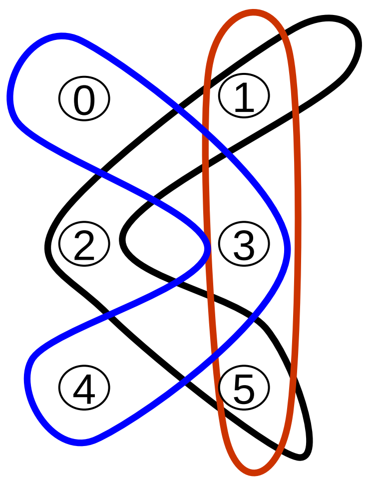
$h_0(jan) = 1 \quad h_1(jan) = 3 \quad h_2(jan) = 5$

$h_0(feb) = 1 \quad h_1(feb) = 2 \quad h_2(feb) = 5$

- 3-graph is induced by three uniform hash functions

Random Hypergraphs (r-graphs)

- 3-graph:



$h_0(jan) = 1 \quad h_1(jan) = 3 \quad h_2(jan) = 5$

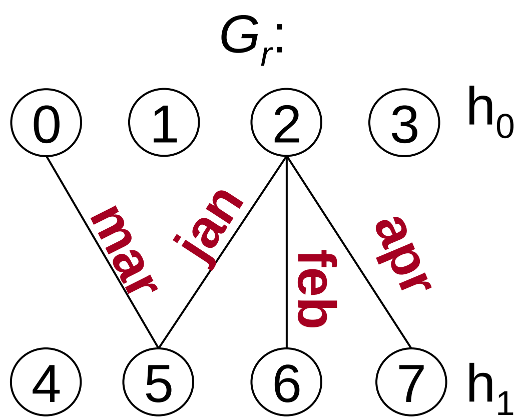
$h_0(feb) = 1 \quad h_1(feb) = 2 \quad h_2(feb) = 5$

$h_0(mar) = 0 \quad h_1(mar) = 3 \quad h_2(mar) = 4$

- 3-graph is induced by three uniform hash functions
- Our best result uses 3-graphs

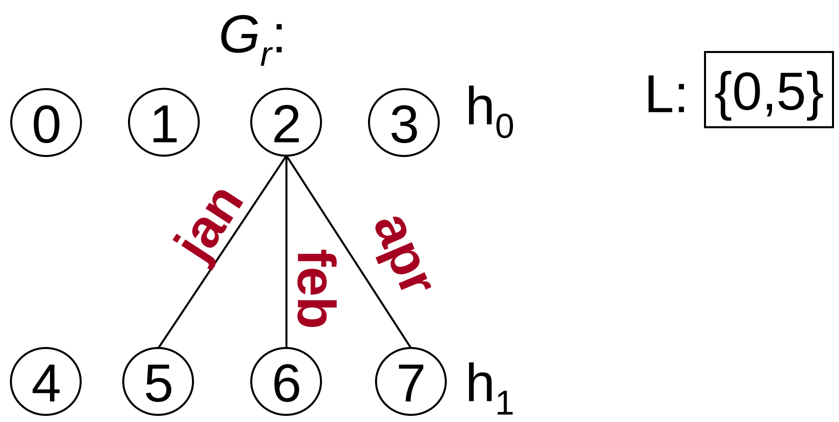
The Internal Random Access memory algorithm ...

Acyclic 2-graph

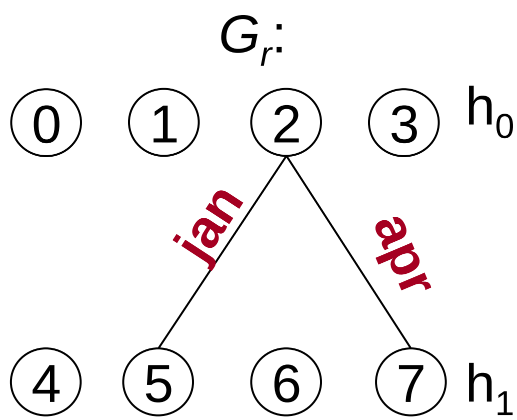


L:Ø

Acyclic 2-graph

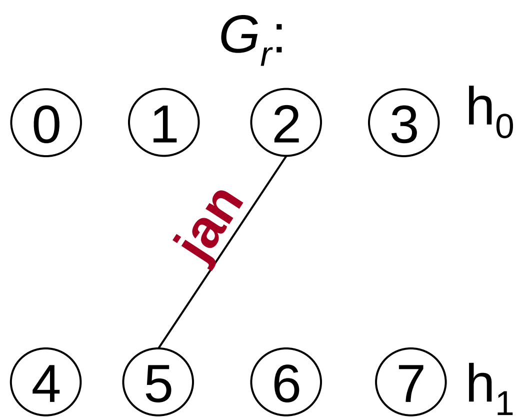


Acyclic 2-graph



	0	1
L:	{0,5}	{2,6}

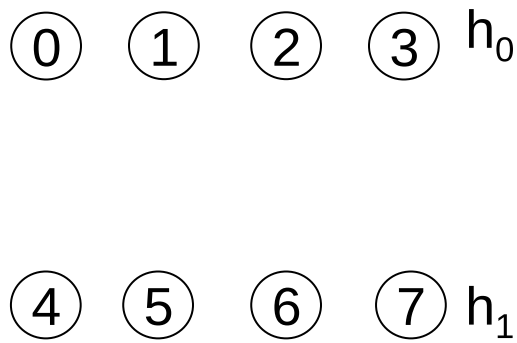
Acyclic 2-graph



	0	1	2
L:	{0,5}	{2,6}	{2,7}

Acyclic 2-graph

G_r:



	0	1	2	3
L:	{0,5}	{2,6}	{2,7}	{2,5}

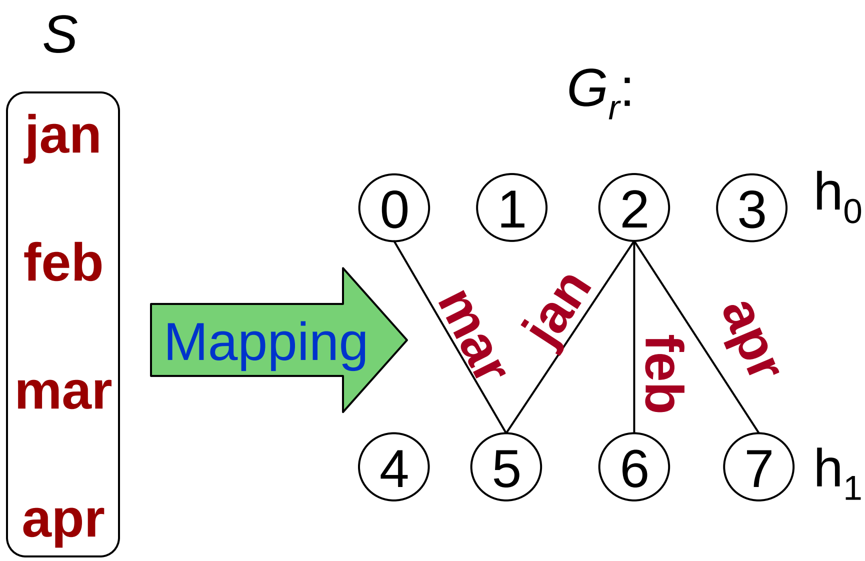
G_r is acyclic

Internal Random Access Memory Algorithm (r=2)

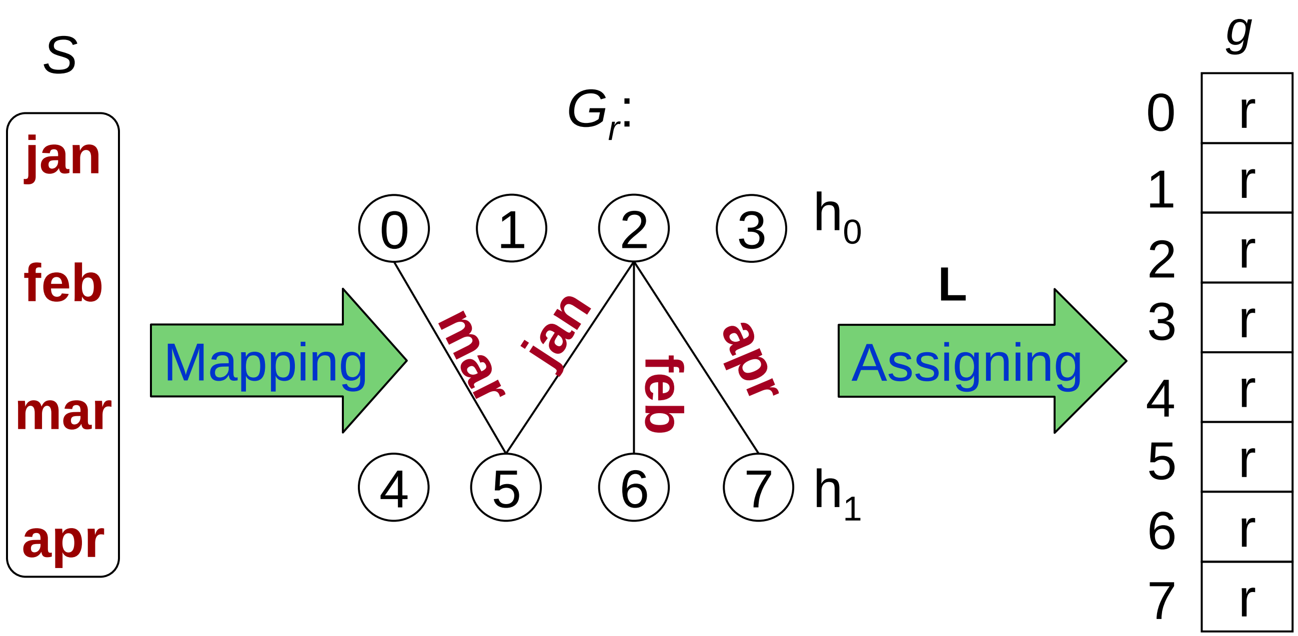
S

jan
feb
mar
apr

Internal Random Access Memory Algorithm (r=2)

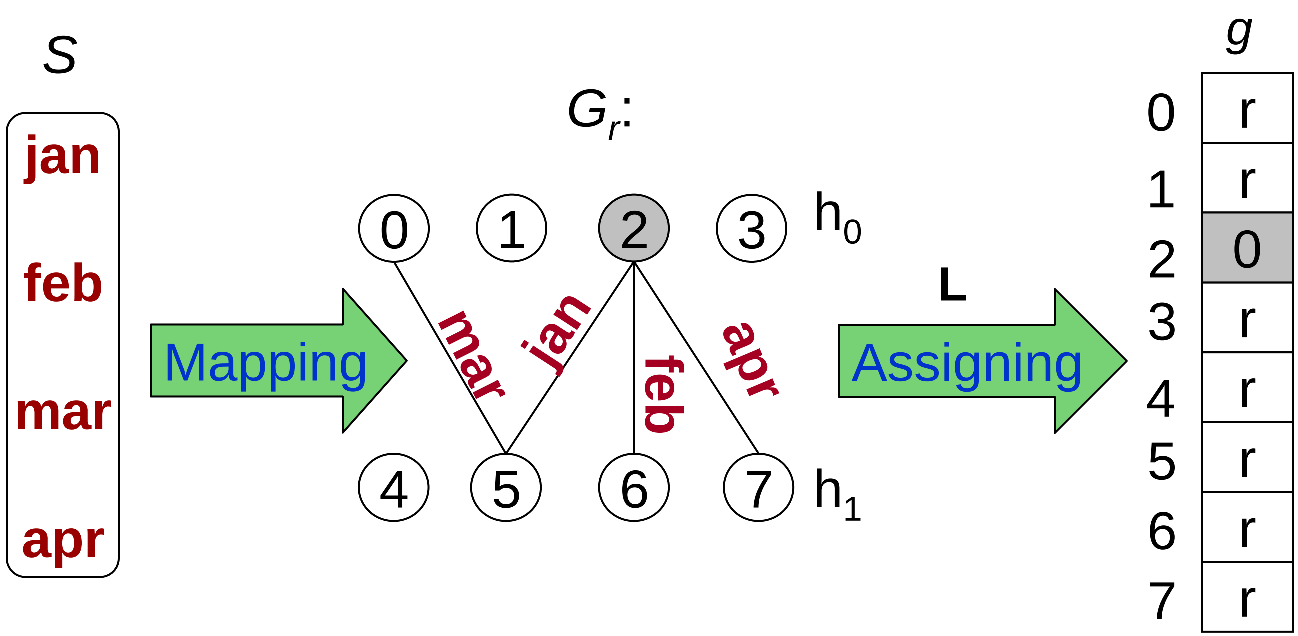


Internal Random Access Memory Algorithm (r=2)



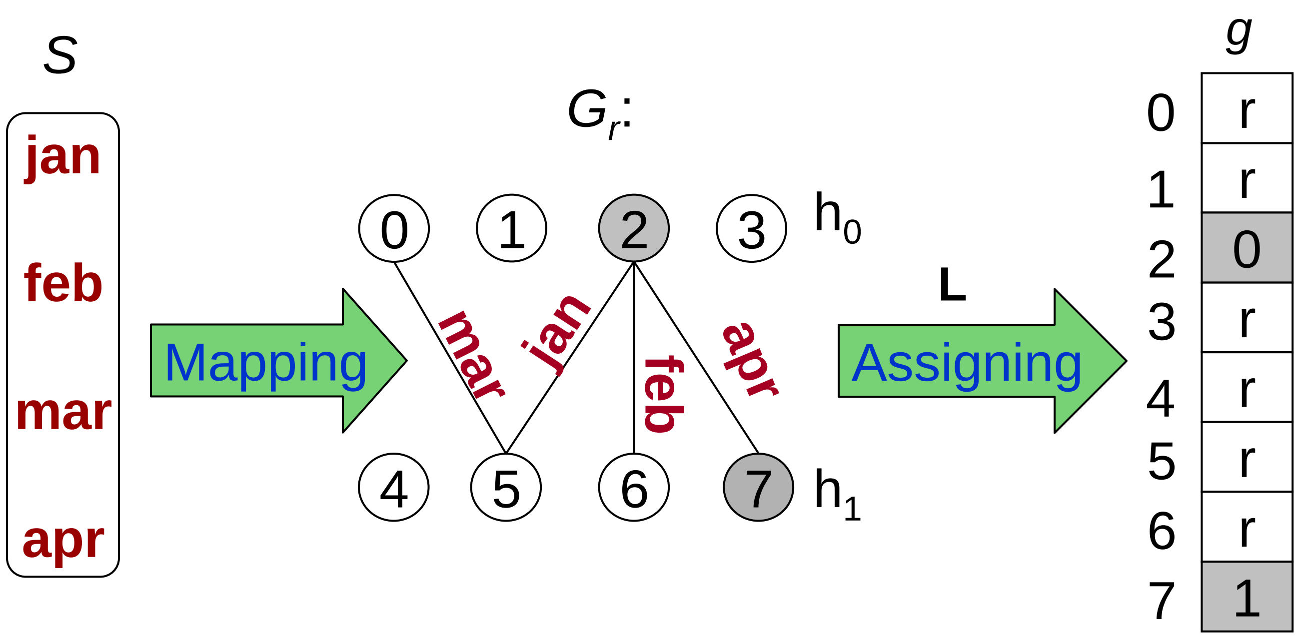
	0	1	2	3
L:	{0,5}	{2,6}	{2,7}	{2,5}

Internal Random Access Memory Algorithm (r=2)



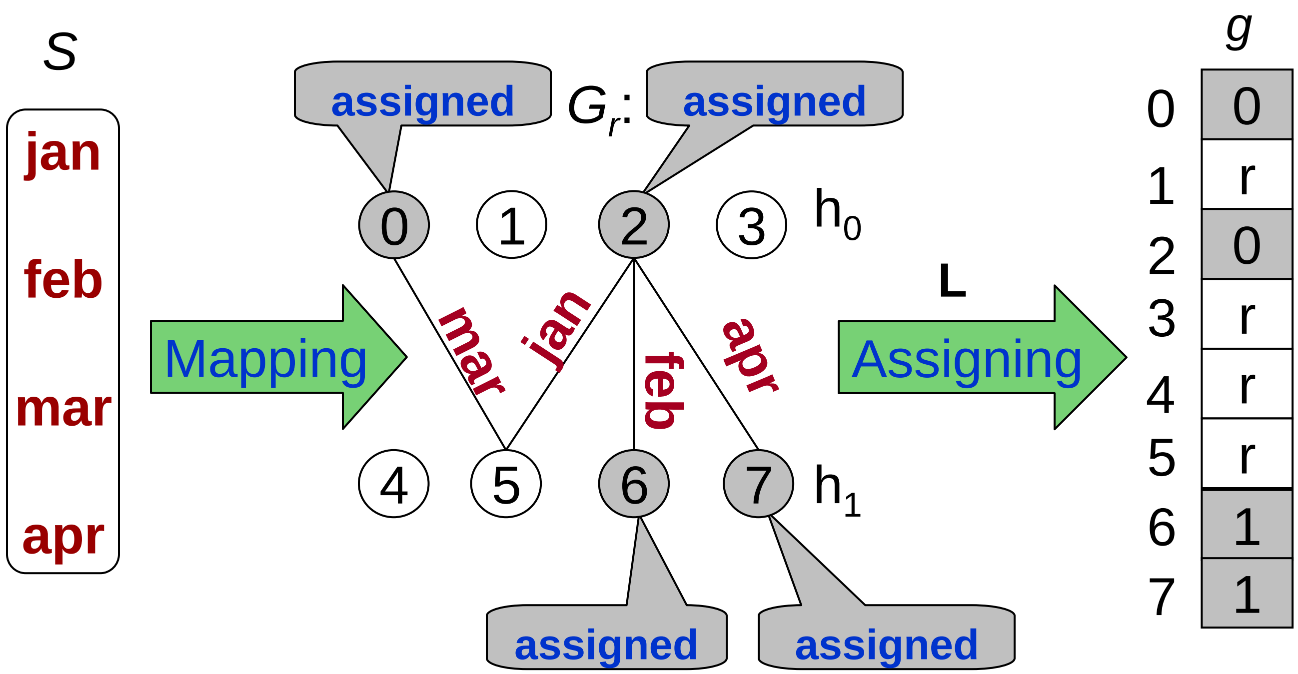
	0	1	2	3
L:	{0,5}	{2,6}	{2,7}	{2,5}

Internal Random Access Memory Algorithm (r=2)

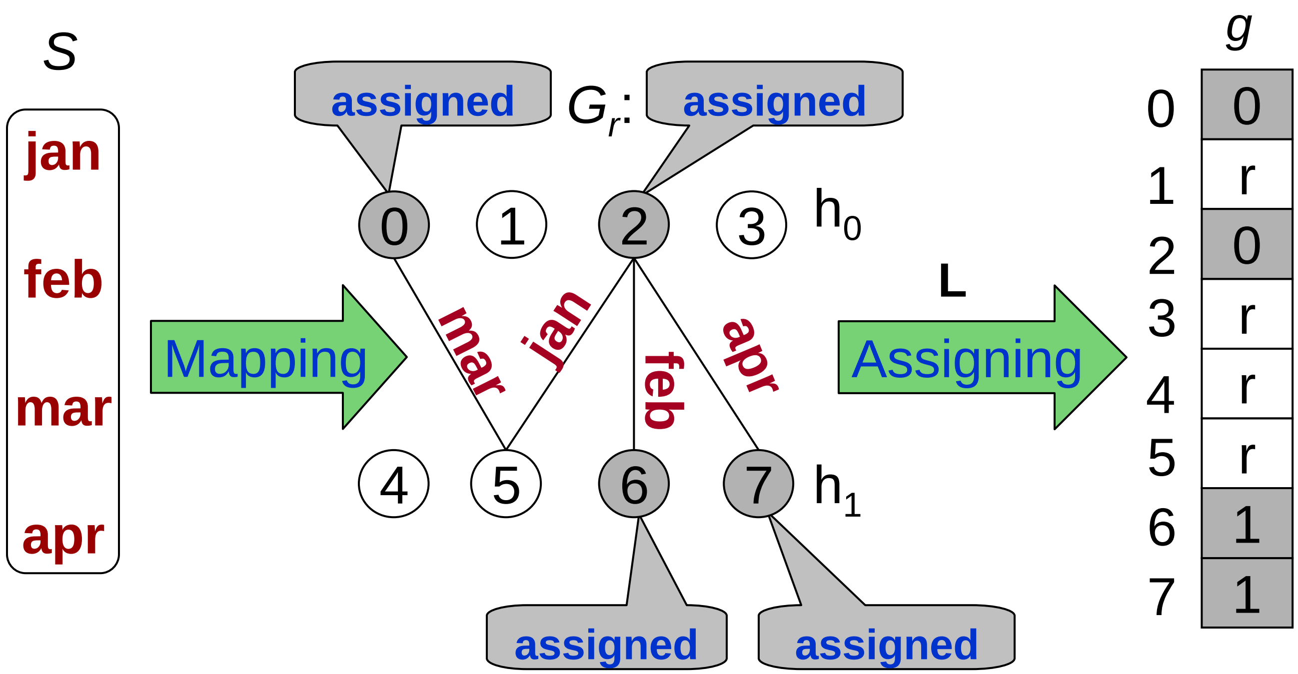


	0	1	2	3
L:	{0,5}	{2,6}	{2,7}	{2,5}

Internal Random Access Memory Algorithm (r=2)

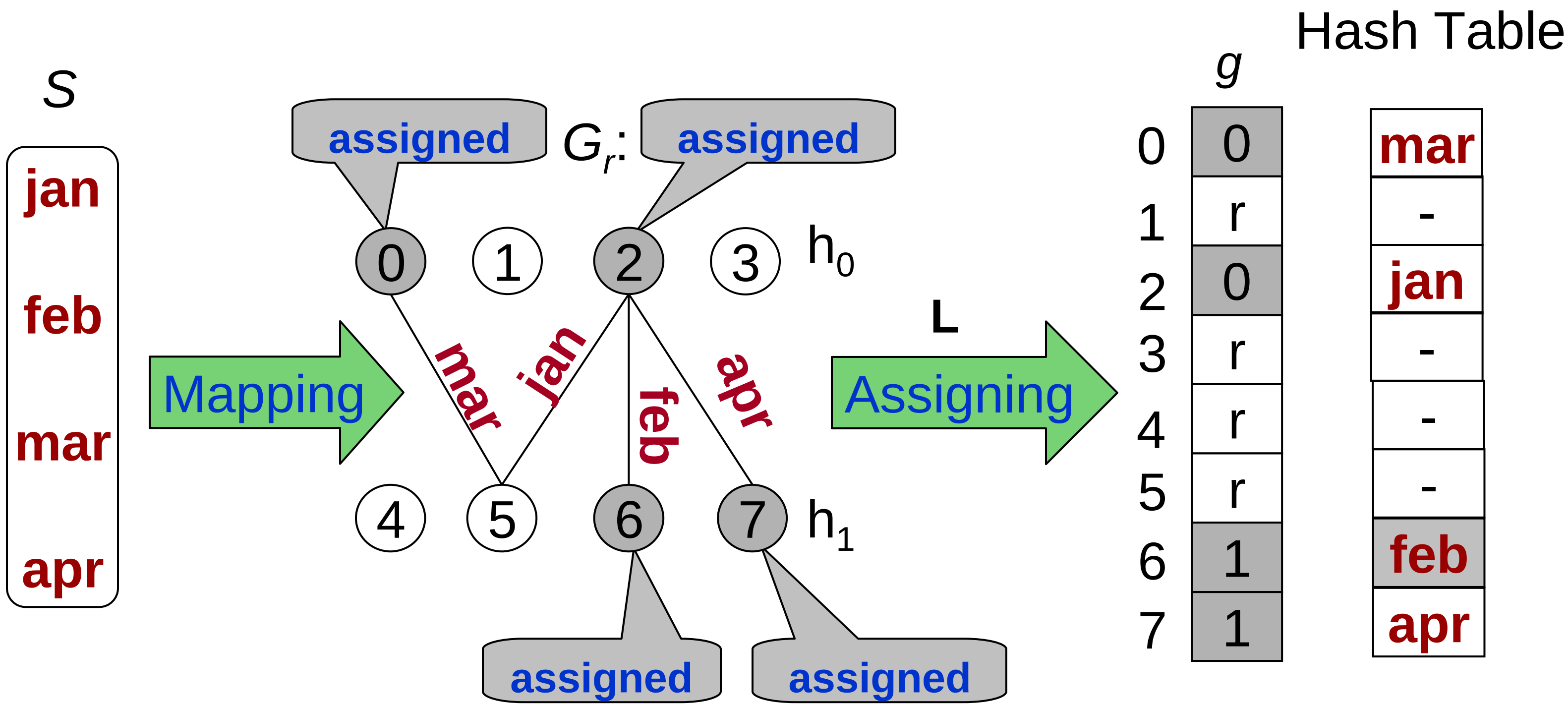


Internal Random Access Memory Algorithm (r=2)



$$i = (g[h_0(\text{feb})] + g[h_1(\text{feb})]) \bmod \mathbf{r} = (g[2] + g[6]) \bmod \mathbf{2} = 1$$

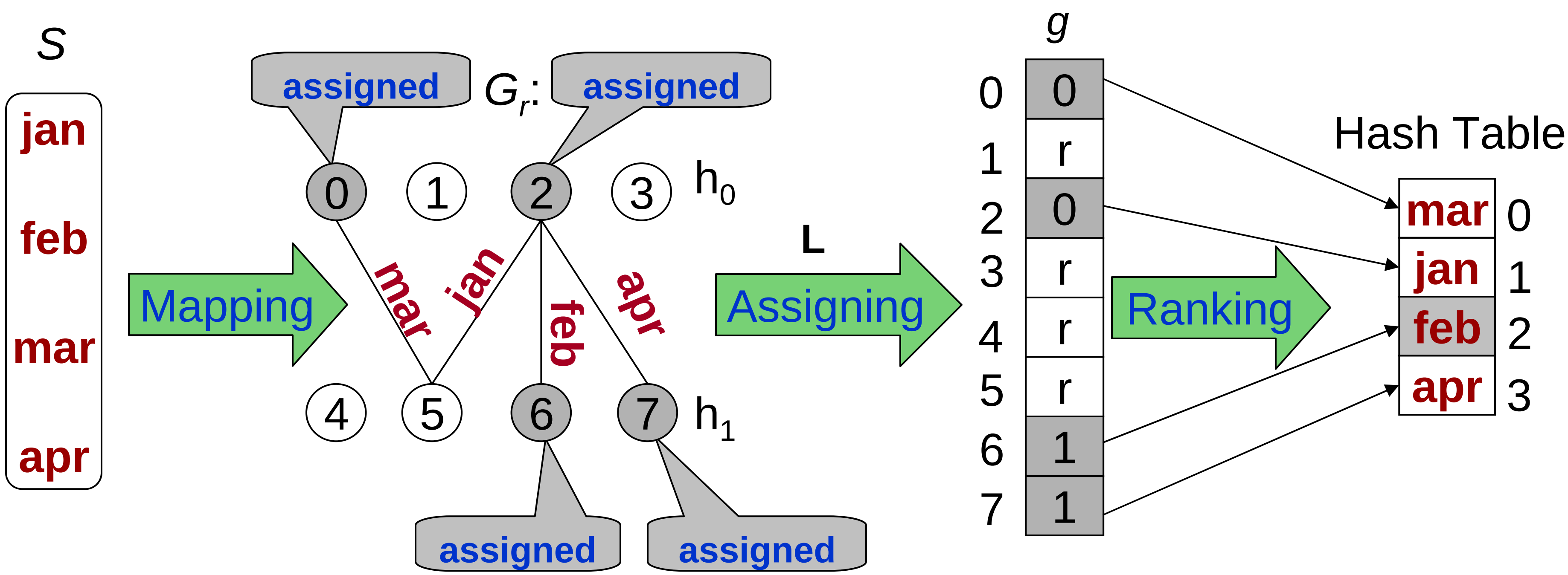
Internal Random Access Memory Algorithm: PHF



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

phf(feb) = $h_{i=1}$ (feb) = 6

Internal Random Access Memory Algorithm: MPHF

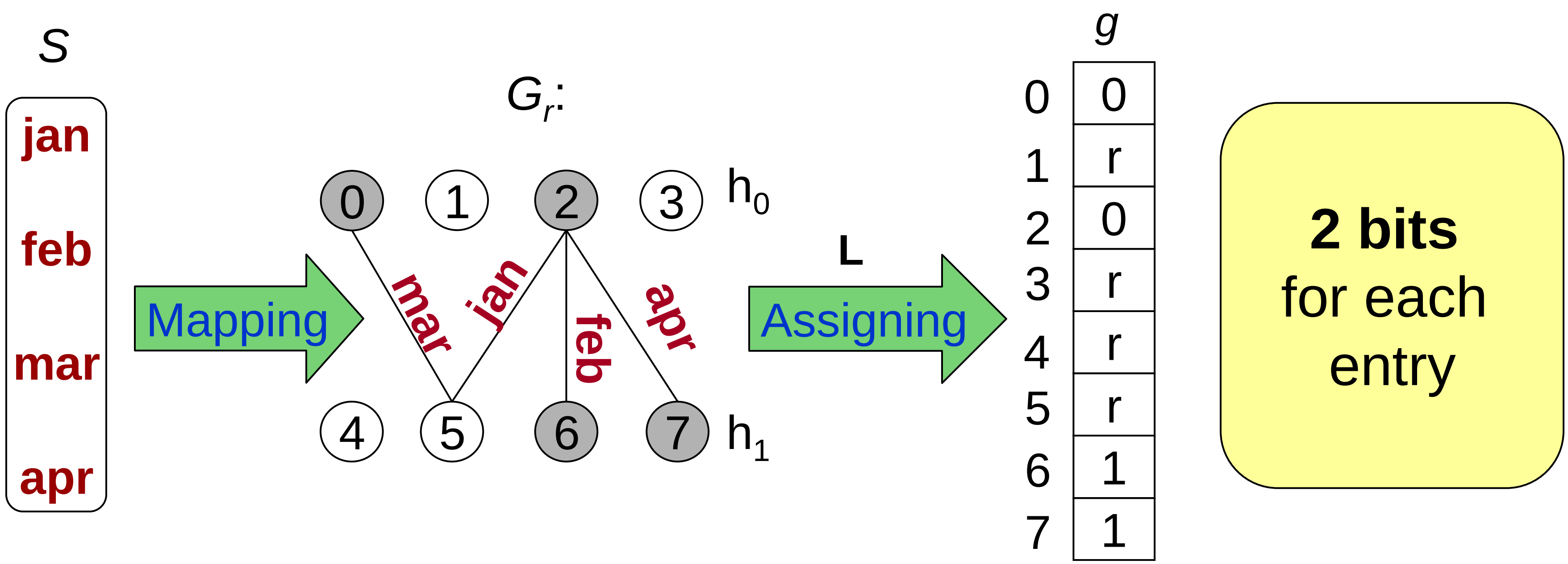


$i = (g[h_0(\text{feb})] + g[h_1(\text{feb})]) \bmod \mathbf{r} = (g[2] + g[6]) \bmod \mathbf{2} = 1$

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

$\text{mphf}(\text{feb}) = \text{rank}(\text{phf}(\text{feb})) = \text{rank}(6) = 2$

Space to Represent the Function



Space to Represent the Functions (r = 3)

- PHF g: [0,m-1] $\rightarrow$ {0,1,2}
  - m = cn bits, c = 1.23 $\rightarrow$ **2.46** n bits
  - (log 3) cn bits, c = 1.23 $\rightarrow$ **1.95 n** bits (arith. coding)
  - Optimal: **0.89n** bits
- MPHF g: [0,m-1] $\rightarrow$ {0,1,2,3} (ranking info required)
  - 2m + εm = (2+ ε)cn bits
  - For c = 1.23 and ε = 0.125 $\rightarrow$ **2.62 n** bits
  - Optimal: **1.44n** bits.

Use of Acyclic Random Hypergraphs

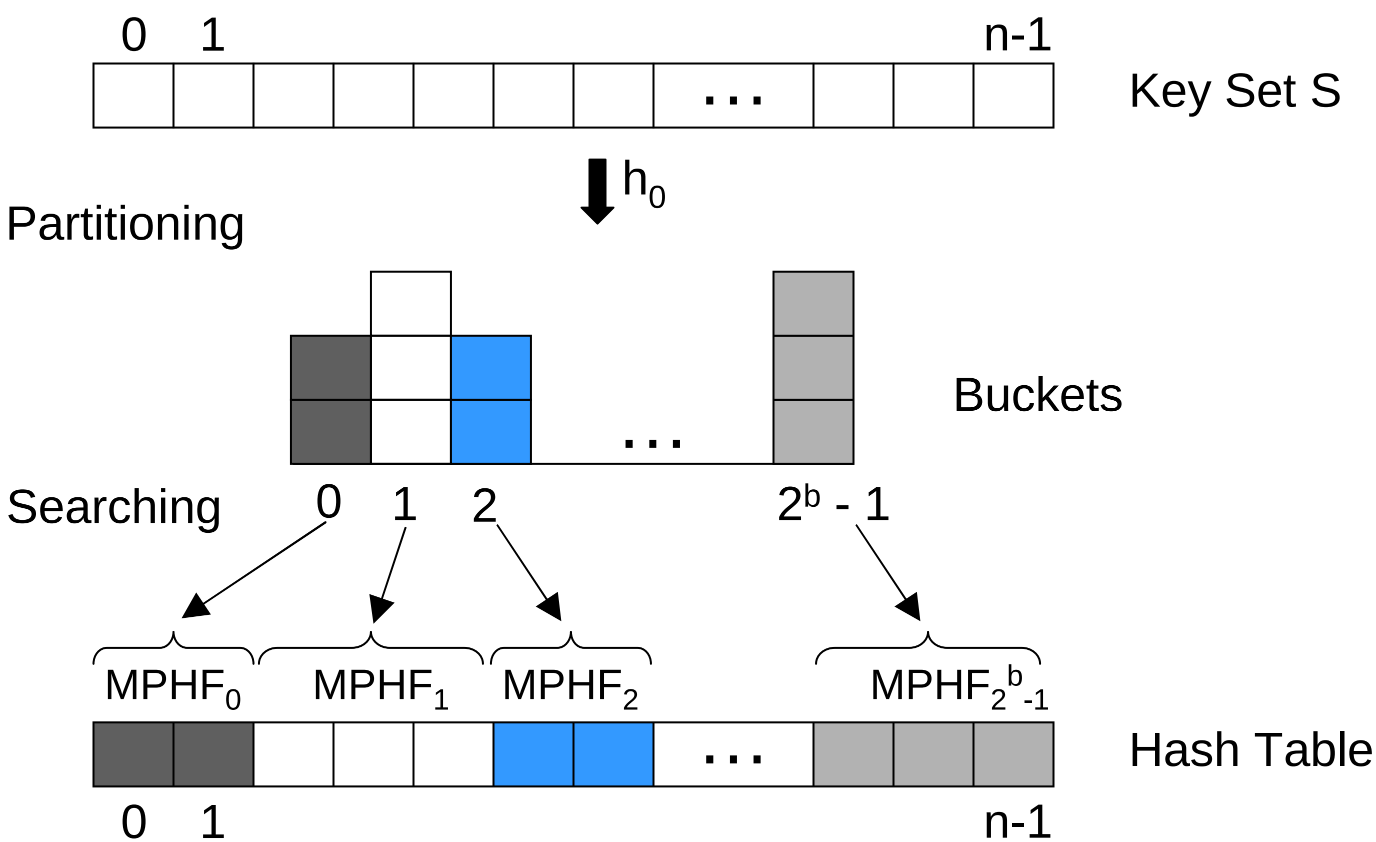
- Sufficient condition to work
- Repeatedly selects h_0, h_1..., h_{r-1}
- For r = 2, m = cn and c ≥ 2.09: Pr_a = 0.29
- For r = 3, m = cn and c≥1.23: Pr_a tends to 1
- Number of iterations is $1/Pr_a$
  - r = 2: 3.5 iterations
  - r = 3: 1.0 iteration

The External Cache-Aware memory algorithm ...

External Cache-Aware Memory Algorithm (ECA)

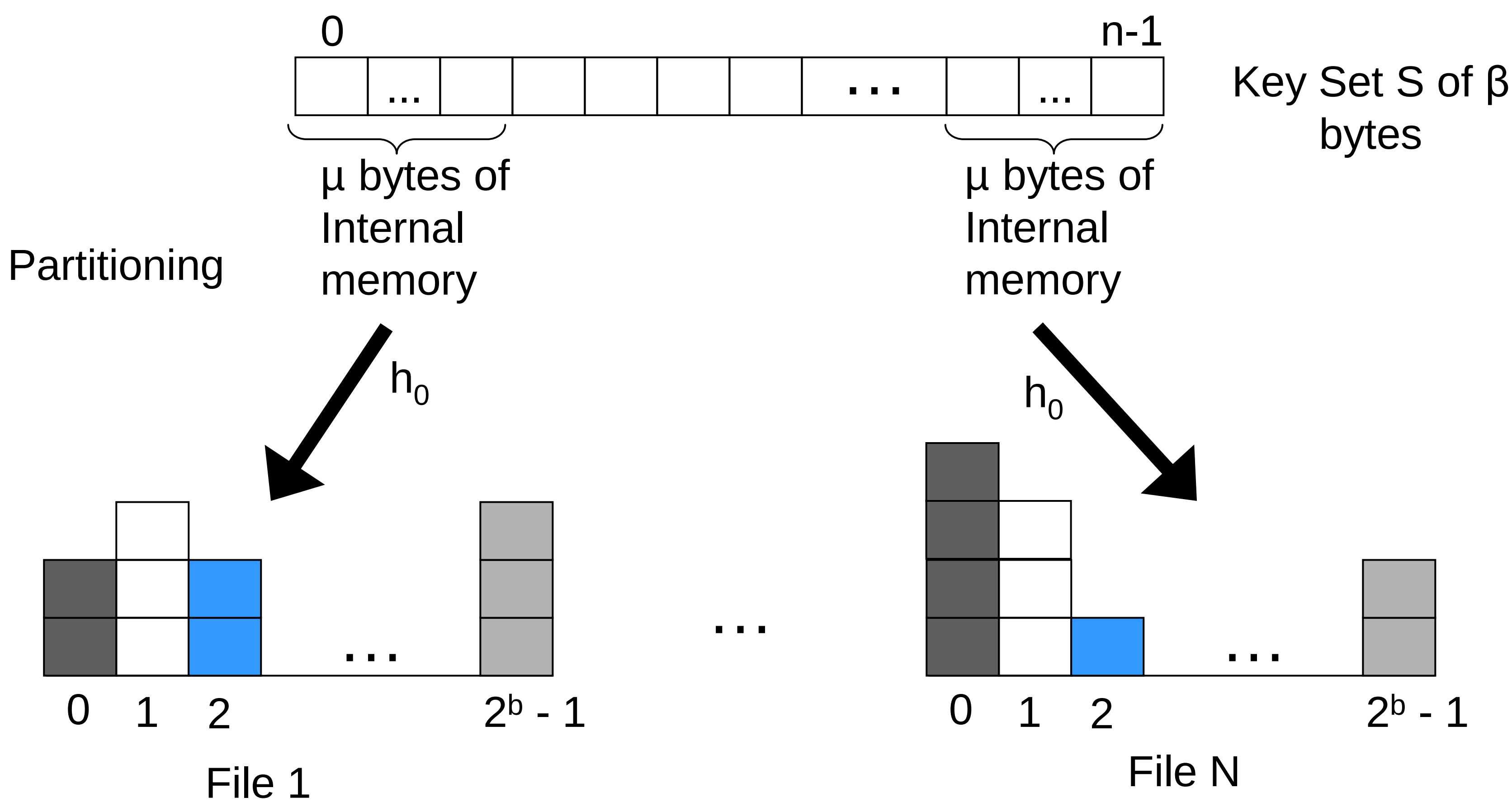
- First MPHF algorithm for very large key sets (in the order of billions of keys)
- This is possible because
  - Deals with external memory efficiently
  - Generates compact functions (near space-optimal)
  - Uses a little amount of internal memory to scale
  - Works in linear time
- Two implementations:
  - Theoretical well-founded ECA (uses uniform hashing)
  - Heuristic ECA (uses universal hashing)

External Cache-Aware Memory Algorithm (ECA)



$MPHF(x) = MPHF_i(x) + offset[i];$

Key Set Does Not Fit In Internal Memory

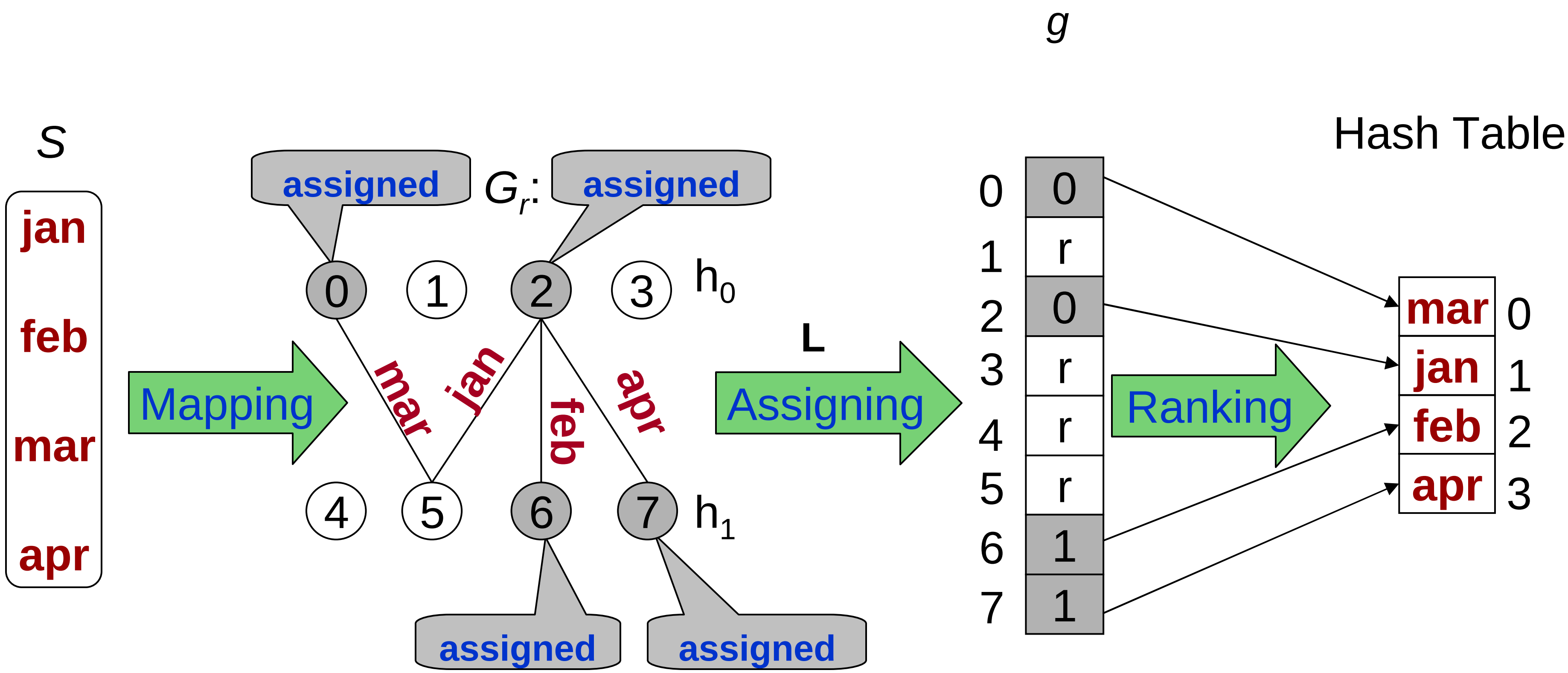


$N = \beta/\mu$ b = Number of bits of each bucket address Each bucket ≤ 256

Important Design Decisions

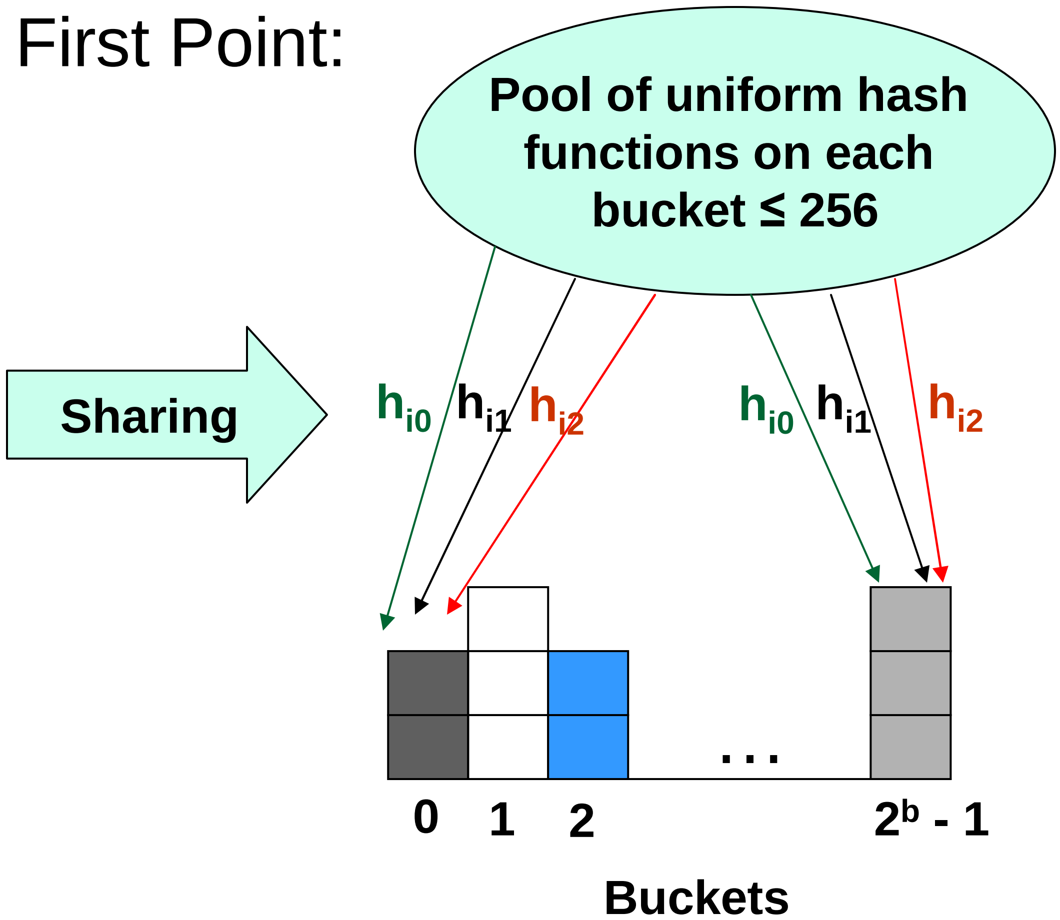
- We map long URLs to a fingerprint of fixed size using a hash function
- Use our IRA linear time and near space-optimal algorithm to generate the MPHF of each bucket
- How do we obtain a linear time complexity?
  - Using internal radix sorting to form the buckets
  - Using a heap of N entries to drive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

Use the Internal Random Access Memory Algorithm for Each Bucket



Why the ECA Algorithm is Well-Founded?

First Point:



Why the ECA 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)

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

$$h_{i0}(x) = f(x,s,0) \bmod |B_i|$$

$$h_{i1}(x) = f(x,s,1) \bmod |B_i| + |B_i|$$

$$h_{i2}(x) = f(x,s,2) \bmod |B_i| + 2|B_i|$$

Why the ECA 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)

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

$$h_{i0}(x) = f(x,s,0) \bmod |B_i|$$

$$h_{i1}(x) = f(x,s,1) \bmod |B_i| + |B_i|$$

$$h_{i2}(x) = f(x,s,2) \bmod |B_i| + 2|B_i|$$

Pool

Why the ECA 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)

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

$$h_{i0}(x) = f(x,s,0) \bmod |B_i|$$

$$h_{i1}(x) = f(x,s,1) \bmod |B_i| + |B_i|$$

$$h_{i2}(x) = f(x,s,2) \bmod |B_i| + 2|B_i|$$

Pool   Random numbers

Why the ECA 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)

Computed by a linear hash function

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

$$h_{i0}(x) = f(x,s,0) \bmod |B_i|$$

$$h_{i1}(x) = f(x,s,1) \bmod |B_i| + |B_i|$$

$$h_{i2}(x) = f(x,s,2) \bmod |B_i| + 2|B_i|$$

Pool Random numbers

Why the ECA Algorithm is Well-Founded?

Second Point:

Computing fingerprints of 128 bits with the linear hash functions

$$h'(x) = 10010101110011011010000111000110$$

$$1101110101001000$$

$$0011000111000110$$

$$0000000111010110$$

$$0011111111000110$$

$$1111111111000110$$

$$0000000000000110$$

$$h_0(x) = h'(x)[96{,}127] >> (32 - b)$$

$$y_6(x) = h'(x)[80{,}95]$$

.

.

.

$$y_1(x) = h'(x)[0{,}15]$$

Why the ECA 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 ECA 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 architeture

Generation Time of MPHFs (in Minutes)

n (millions)	32	128	512	1024
Theoretic ECA	1.3 ± 0.002	6.2 ± 0.02	27.6 ± 0.09	57.4 ± 0.06
Heuristic ECA	0.95 ± 0.02	5.1 ± 0.01	22.0 ± 0.13	46.2 ± 0.06

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

Algorithms	Generation Time (sec)
IRA (r = 3)	6.7 ± 0
Theoretic ECA	9.0 ± 0.3
Heuristic ECA	6.4 ± 0.3
BKZ	12.8 ± 1.6
CHM	17.0 ± 3.2
FCH	2,400.1 ± 711.6
PAGH	42.8 ± 2.4

3,541,615 URLs

Generation Time and Storage Space

Algorithms	Generation Time (sec)	Space (bits/key)
IRA (r = 3)	6.7 ± 0	2.6
Theoretic ECA	9.0 ± 0.3	3.3
Heuristic ECA	6.4 ± 0.3	3.1
BKZ	12.8 ± 1.6	21.8
CHM	17.0 ± 3.2	45.5
FCH	2,400.1 ± 711.6	4.2
PAGH	42.8 ± 2.4	44.2

3,541,615 URLs

Generation Time, Storage Space and Evaluation Time

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

3,541,615 URLs

Key length = 64 bytes

C Minimal Perfect Hashing Library

- Why to build a library?
  - Lack of similar libraries in the free software community
  - Test the applicability of our algorithm out there
- Feedbacks:
  - 1,883 downloads (until Apr 15th, 2008)
  - Incorporated by Debian
- Library address: http://cmph.sourceforge.net

Conclusions

- Three implementations were developed:
  - Theoretic ECA (external memory)
  - Heuristic ECA (external memory)
  - IRA (internal memory, used in ECA algorithm)
- Near space-optimal 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

Parallel Version of the ECA Algorithm

PCs	2	4	8	10	14
Speedup	1.8	3.5	7.0	8.7	12.2

1 billion URLs using 14 PCs in 5 minutes