Near-Optimal Space Perfect Hashing Algorithm

Nivio Ziviani

Fabiano C. Botelho

Department of Computer Science
Federal University of Minas Gerais, Brazil

University of Granada
Granada, Spain, April 14th, 2009

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 **R**andom **A**ccess **M**emory algorithm (**RAM**)
- Key set larger than the internal memory
  - **E**xternal **M**emory (cache-aware) algorithm (**EM**)

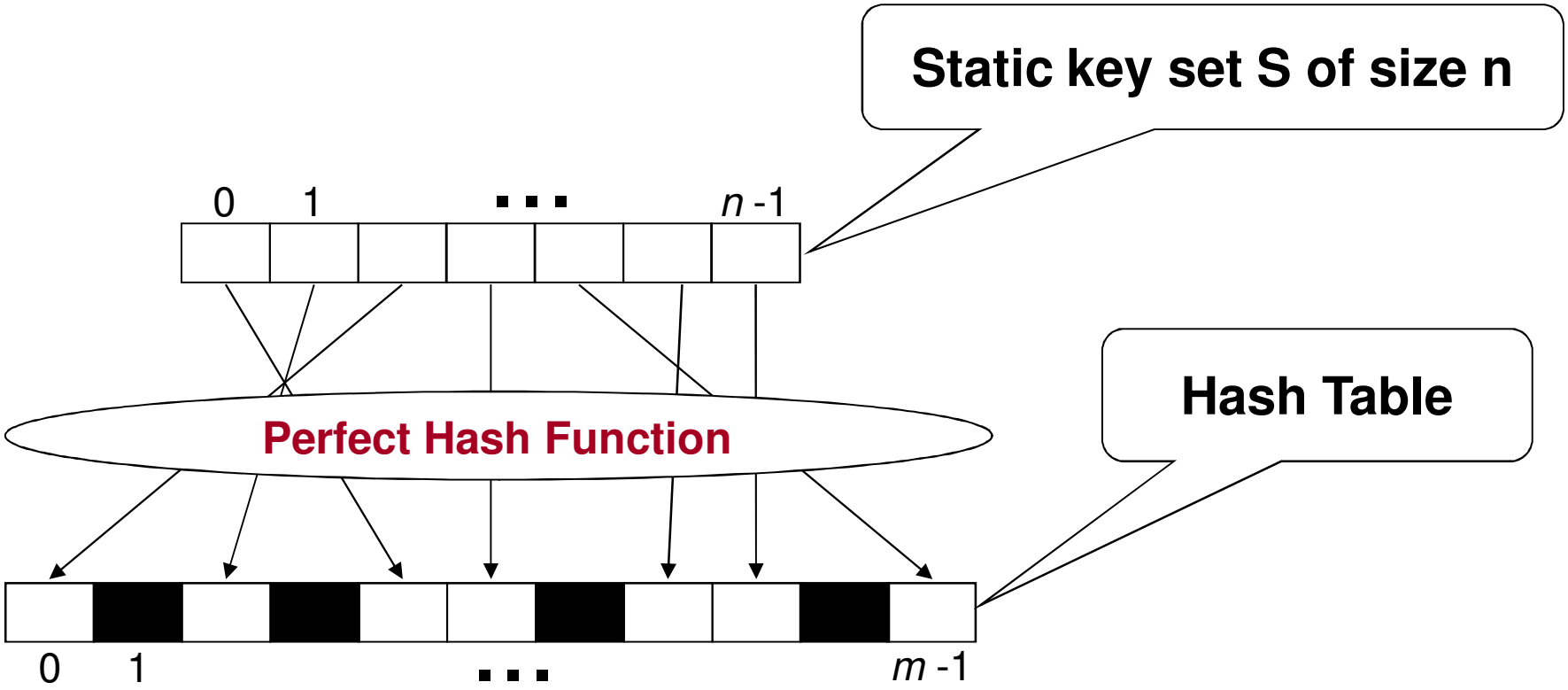
Objective of the Presentation

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 **R**andom **A**ccess **M**emory algorithm (**RAM**)
- Key set larger than the internal memory
  - **E**xternal **M**emory (cache-aware) algorithm (**EM**)

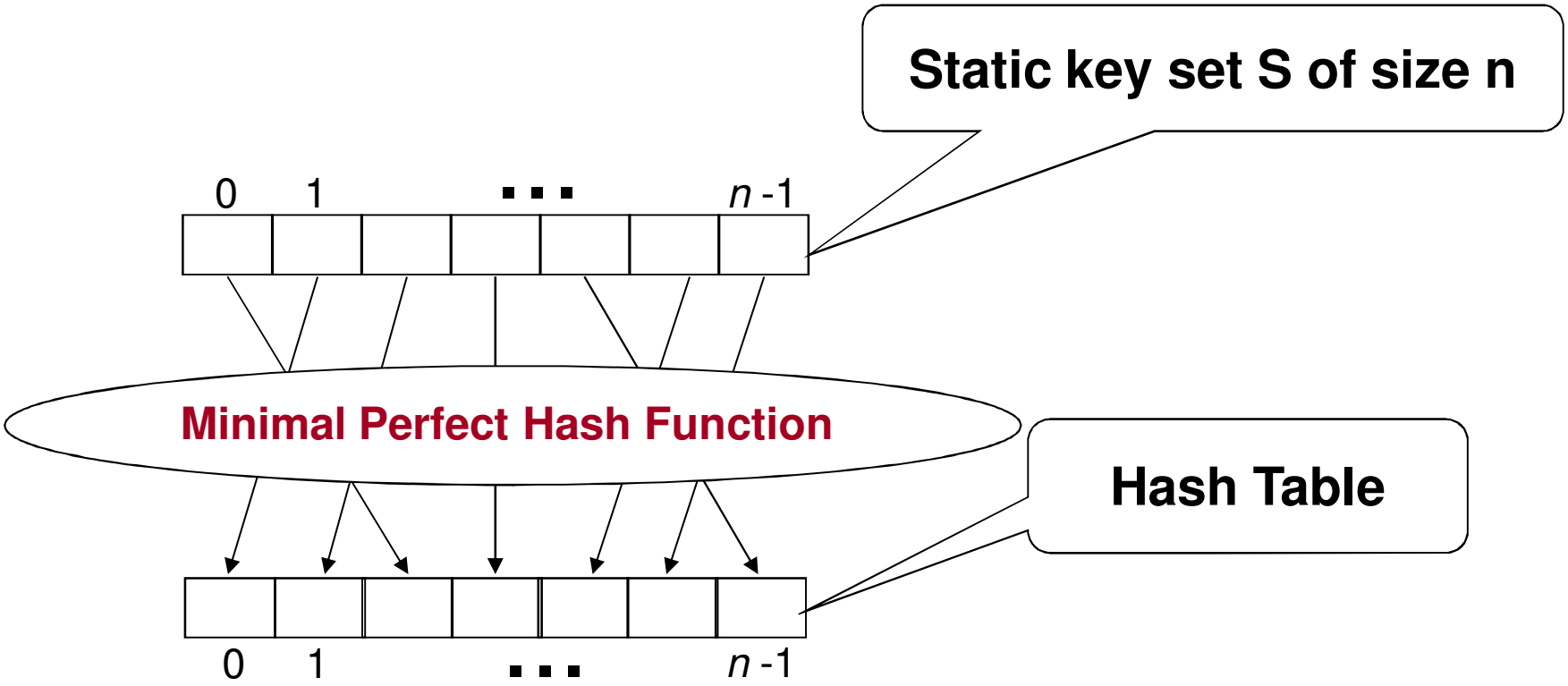
Theoretically well-founded, time efficient, higly scalable and near space-optimal

Perfect Hash Function



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

Minimal Perfect Hash Function

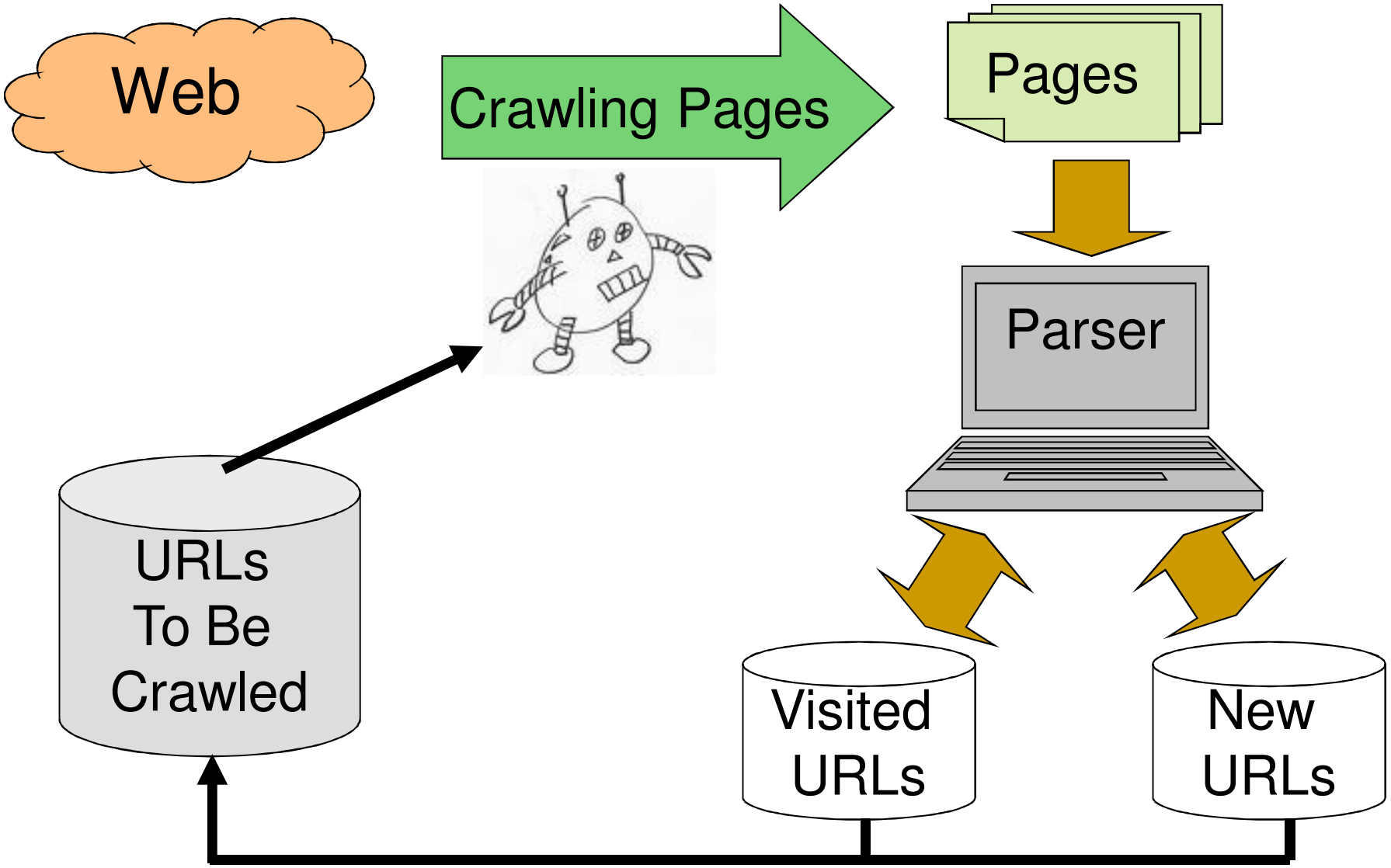


$$S \subseteq U, \text{where} |U| = u$$

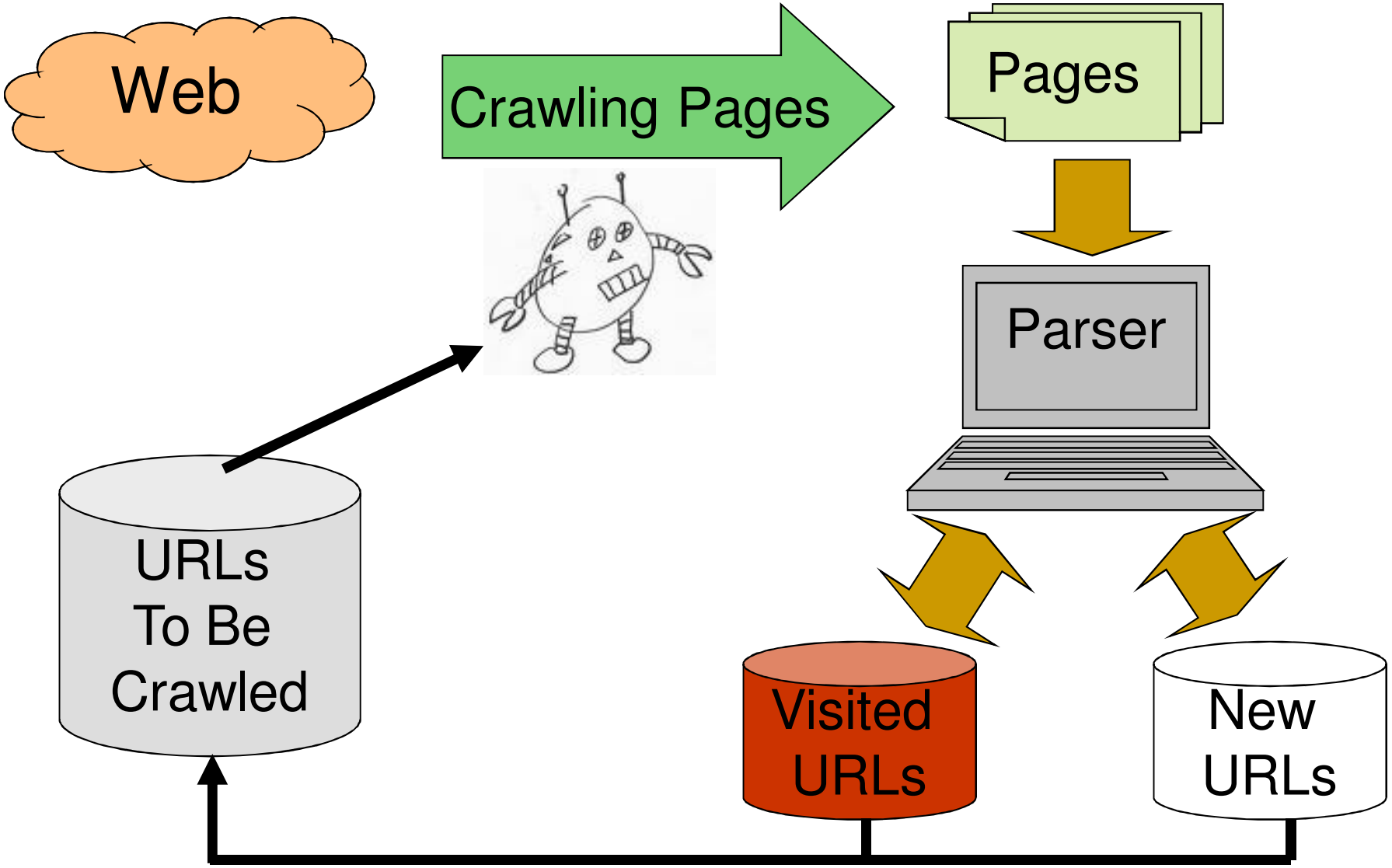
Where to use a PHF or a MPHF?

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

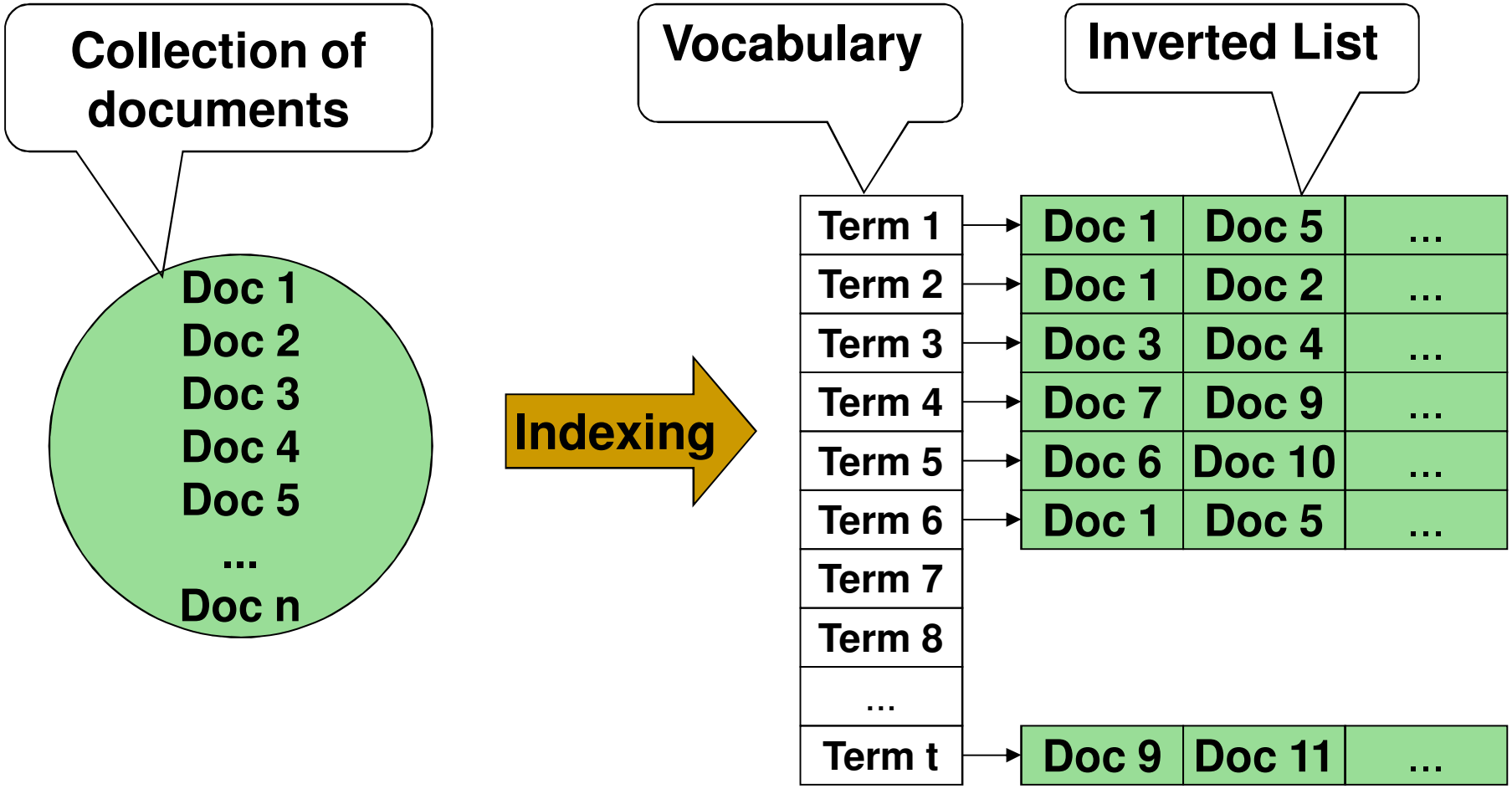
Crawling


Web
Crawling Pages
Pages
Parser
URLs
To Be
Crawled
Visited
URLs
New
URLs

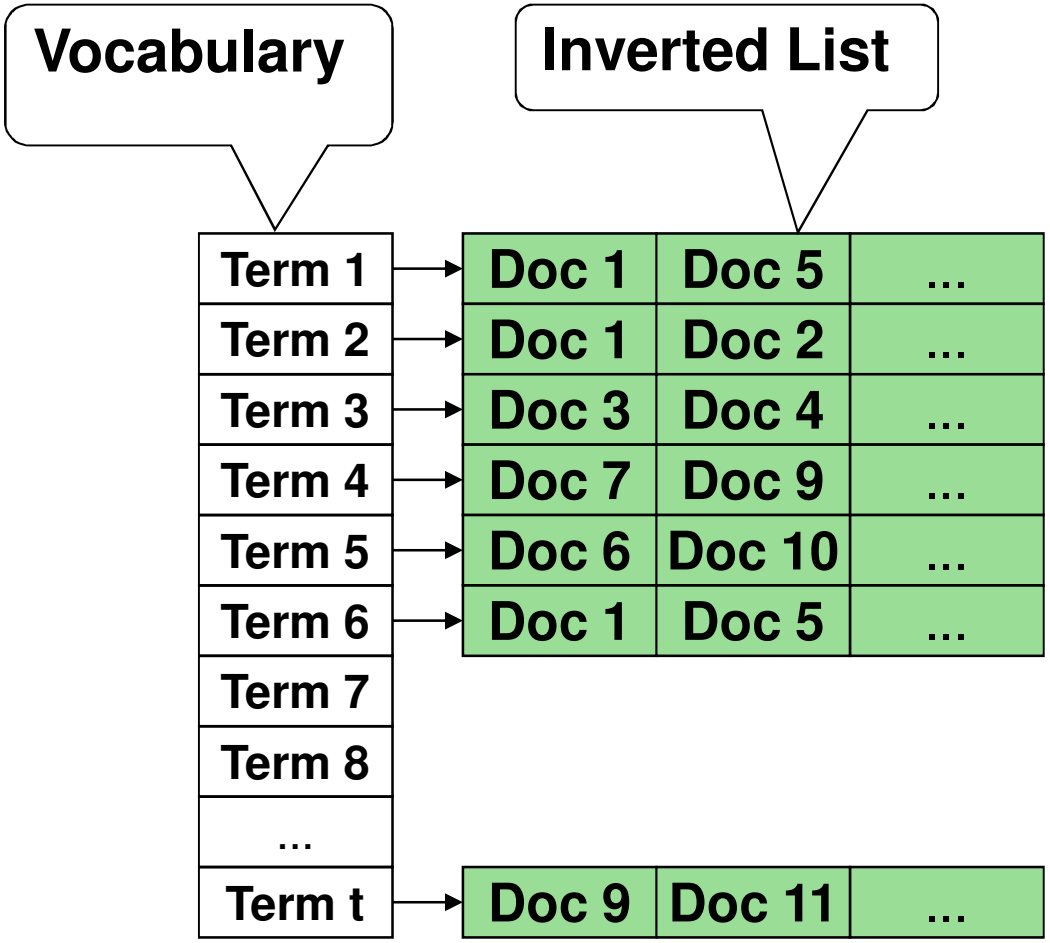
Representing Visited URLS

- 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

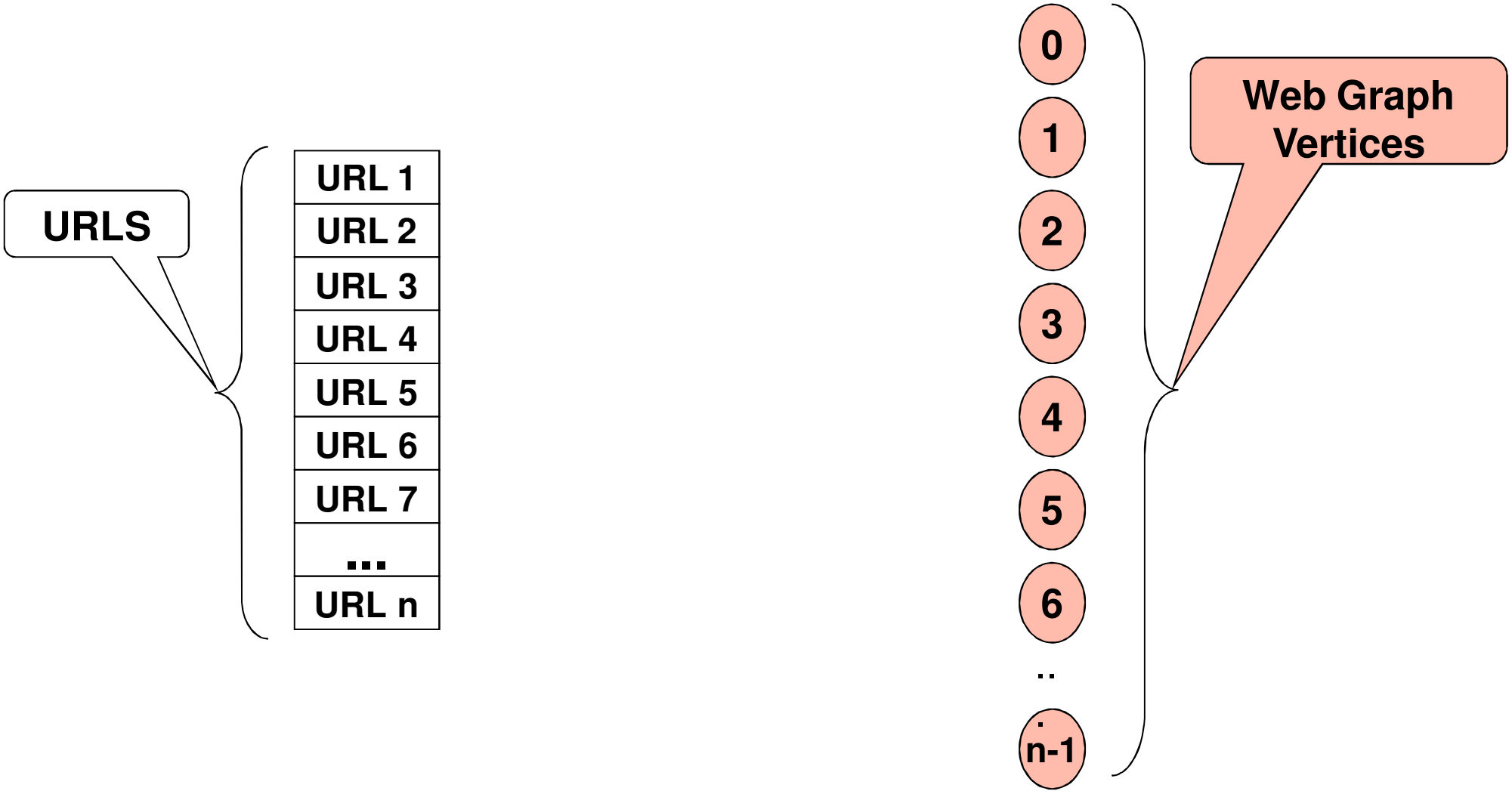
Indexing


Collection of documents
Doc 1
Doc 2
Doc 3
Doc 4
Doc 5
...
Doc n
Indexing
Vocabulary
Inverted List
Term 1
Term 2
Term 3
Term 4
Term 5
Term 6
Term 7
Term 8
...
Term t
Doc 1
Doc 5
...
Doc 1
Doc 2
...
Doc 3
Doc 4
...
Doc 7
Doc 9
...
Doc 6
Doc 10
...
Doc 1
Doc 5
...
Doc 9
Doc 11
...

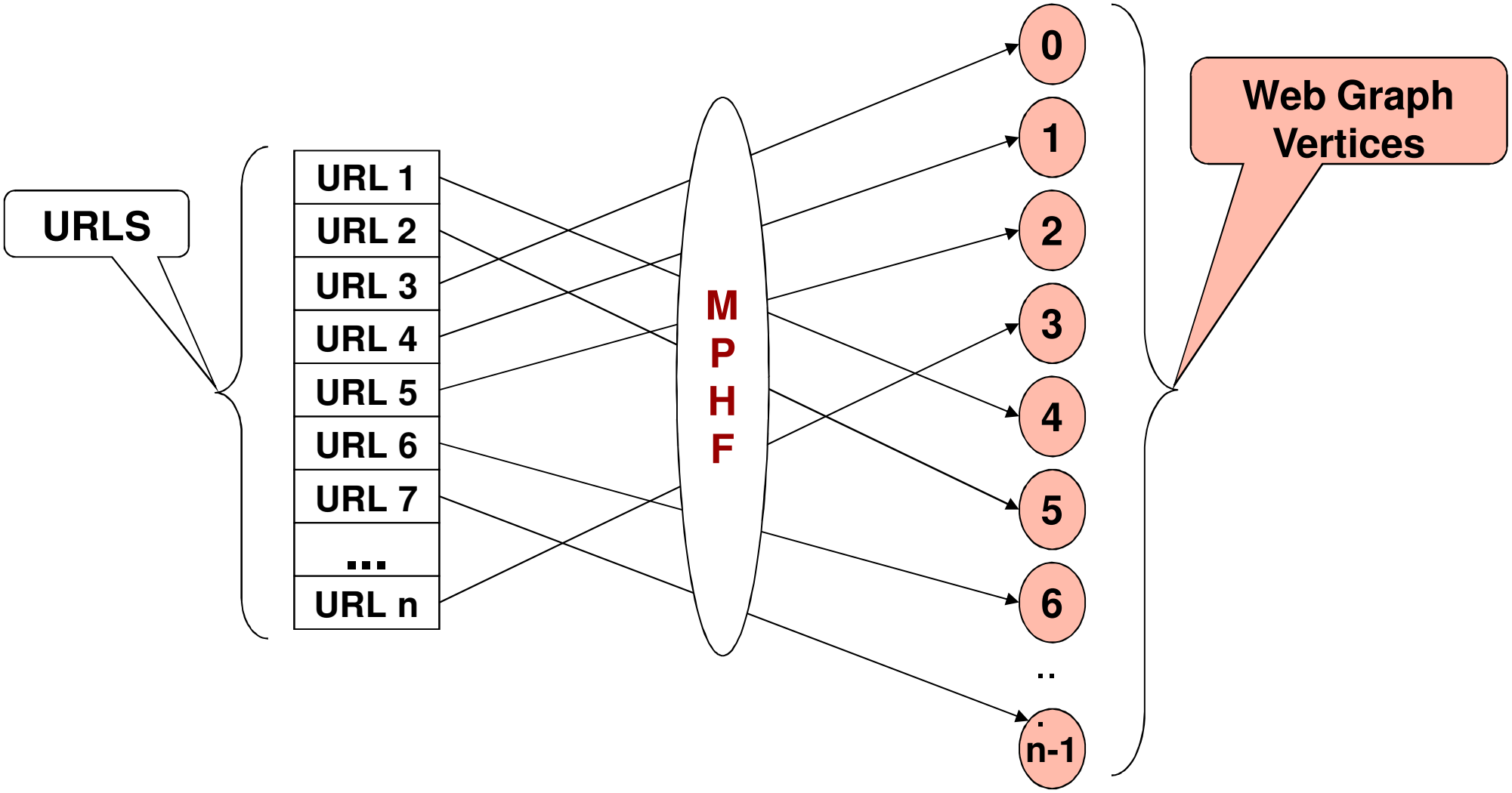
Representing the Vocabulary



Mapping URLs to Web Graph Vertices



Mapping URLs to Web Graph Vertices



Information Theoretical Lower Bounds for Storage Space

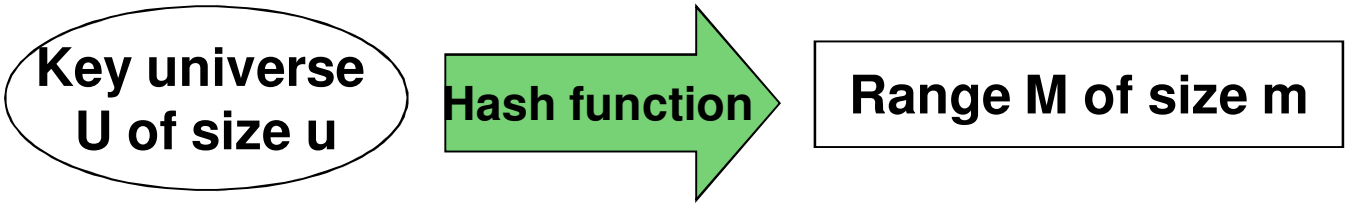
- PHFs (m ≈ n): $\text{Storage Space} \geq \frac{n^2}{m} \log e$

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

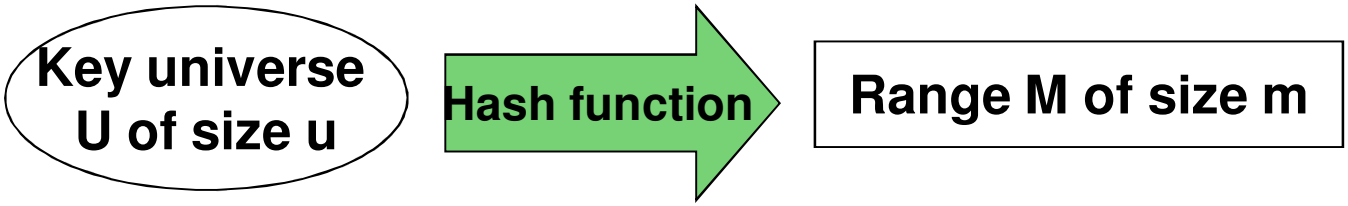
m < 3n

$\log e \approx 1.4427$

Uniform Hashing Versus Universal Hashing



Uniform Hashing Versus Universal Hashing



Uniform hashing

- # of functions from U to M?

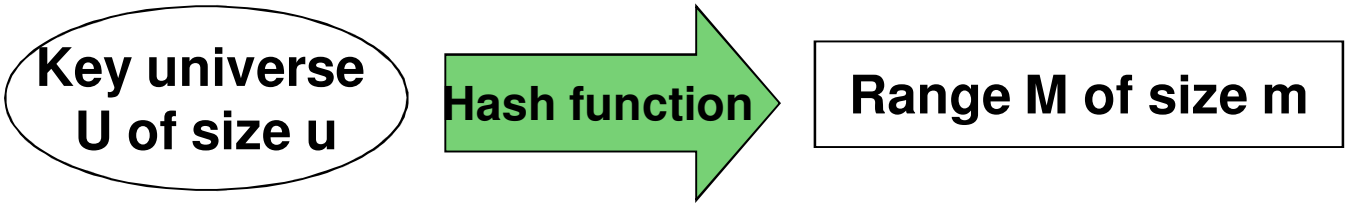
$$m^u$$

- # 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 EM (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)**
RAM (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)**
RAM (WADS 2007)	**O(n)**	**O(1)**	**O(n)**
Heuristic EM (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**

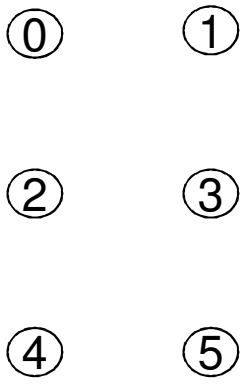
Internal Random Access and External Memory Algorithms

- 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 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)

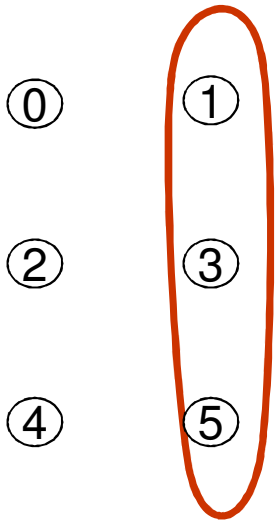
- 3-graph:



- 3-graph is induced by three uniform hash functions

Random Hypergraphs (r-graphs)

- 3-graph:

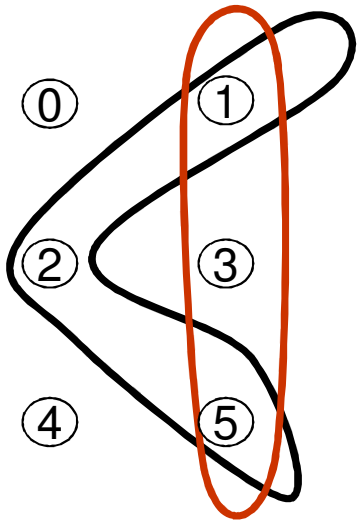


$h_0(jan) = 1 \quad h_1(jan) = 3 \quad h_2(jan) = 5$

- 3-graph is induced by three uniform hash functions

Random Hypergraphs (r-graphs)

- 3-graph:



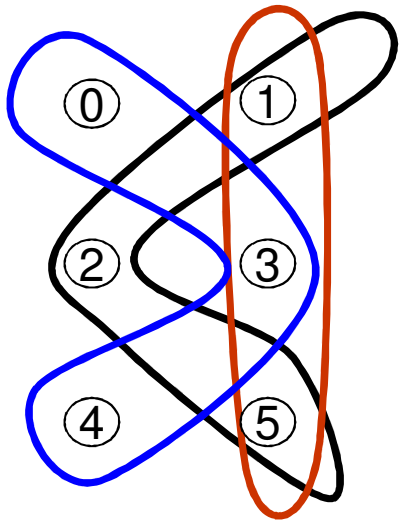
$h_0(jan) = 1 \quad h_1(jan) = 3 \quad h_2(jan) = 5$

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

- 3-graph is induced by three uniform hash functions

Random Hypergraphs (r-graphs)

- 3-graph:



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

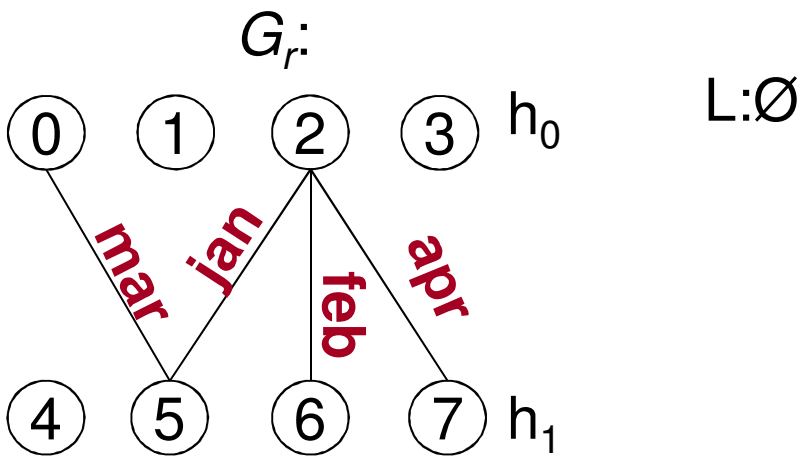
$h_0(feb) = 1 \quad h_1(feb) = 2 \quad h_2(feb) = 5$

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

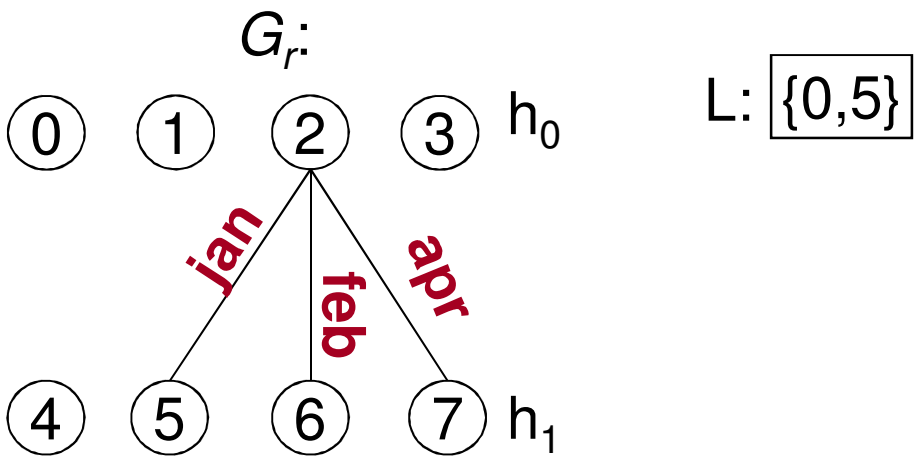
- 3-graph is induced by three uniform hash functions
- Our best result uses 3-graphs

The Internal Random Access Memory Algorithm ...

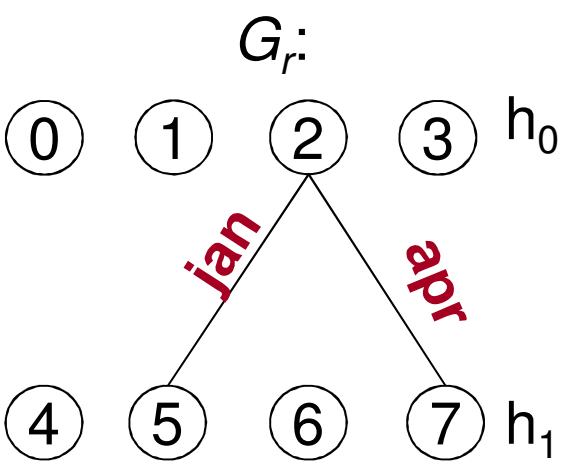
Acyclic 2-graph



Acyclic 2-graph

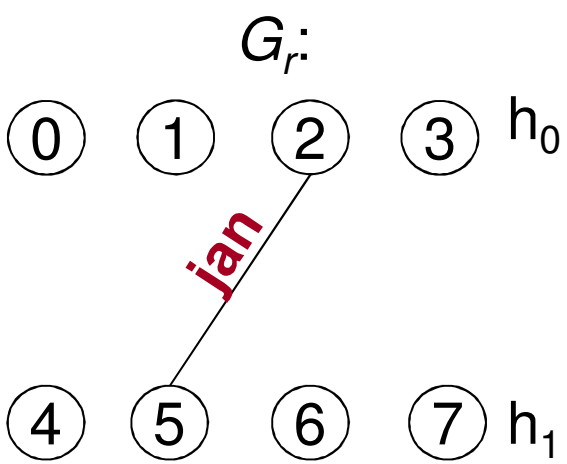


Acyclic 2-graph



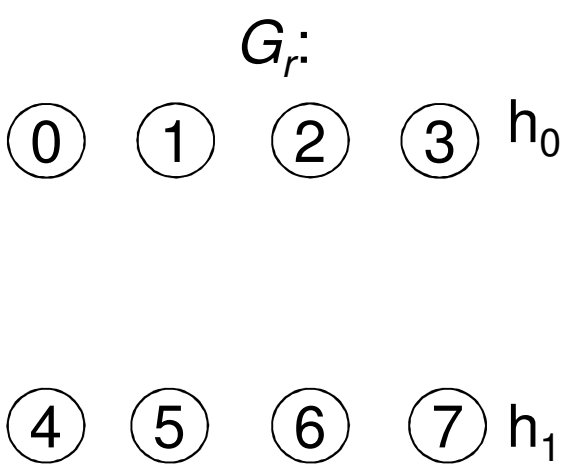
	0	1
L:	{0,5}	{2,6}

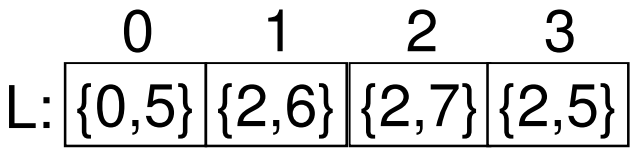
Acyclic 2-graph



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

Acyclic 2-graph





G_r is acyclic

Internal Random Access Memory Algorithm (r=2)

S

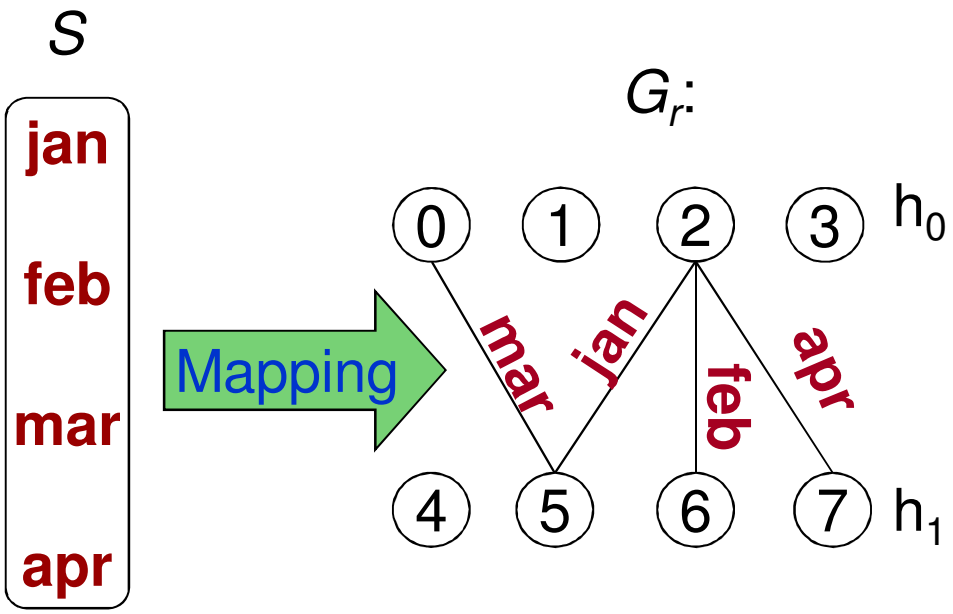
jan

feb

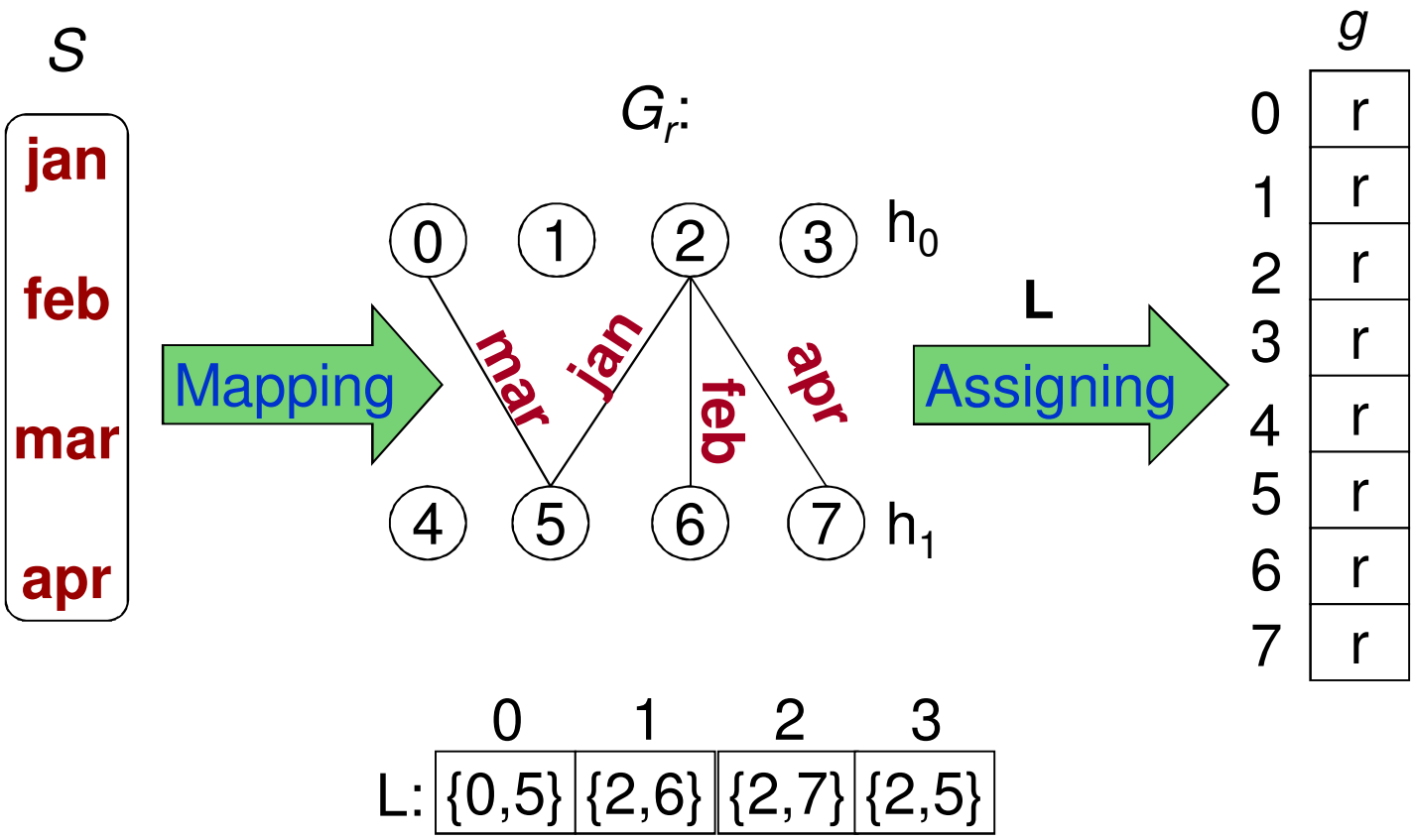
mar

apr

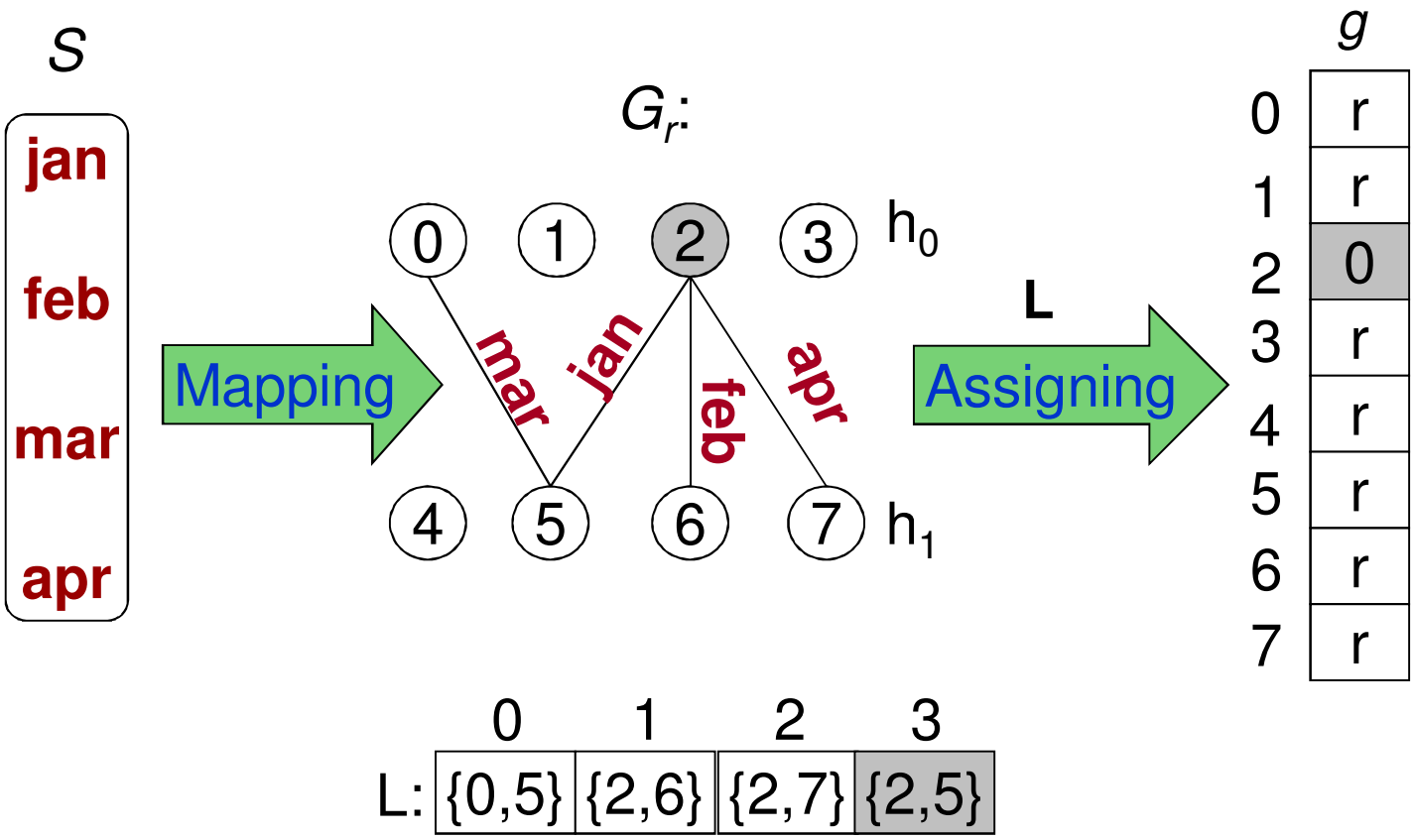
Internal Random Access Memory Algorithm (r=2)



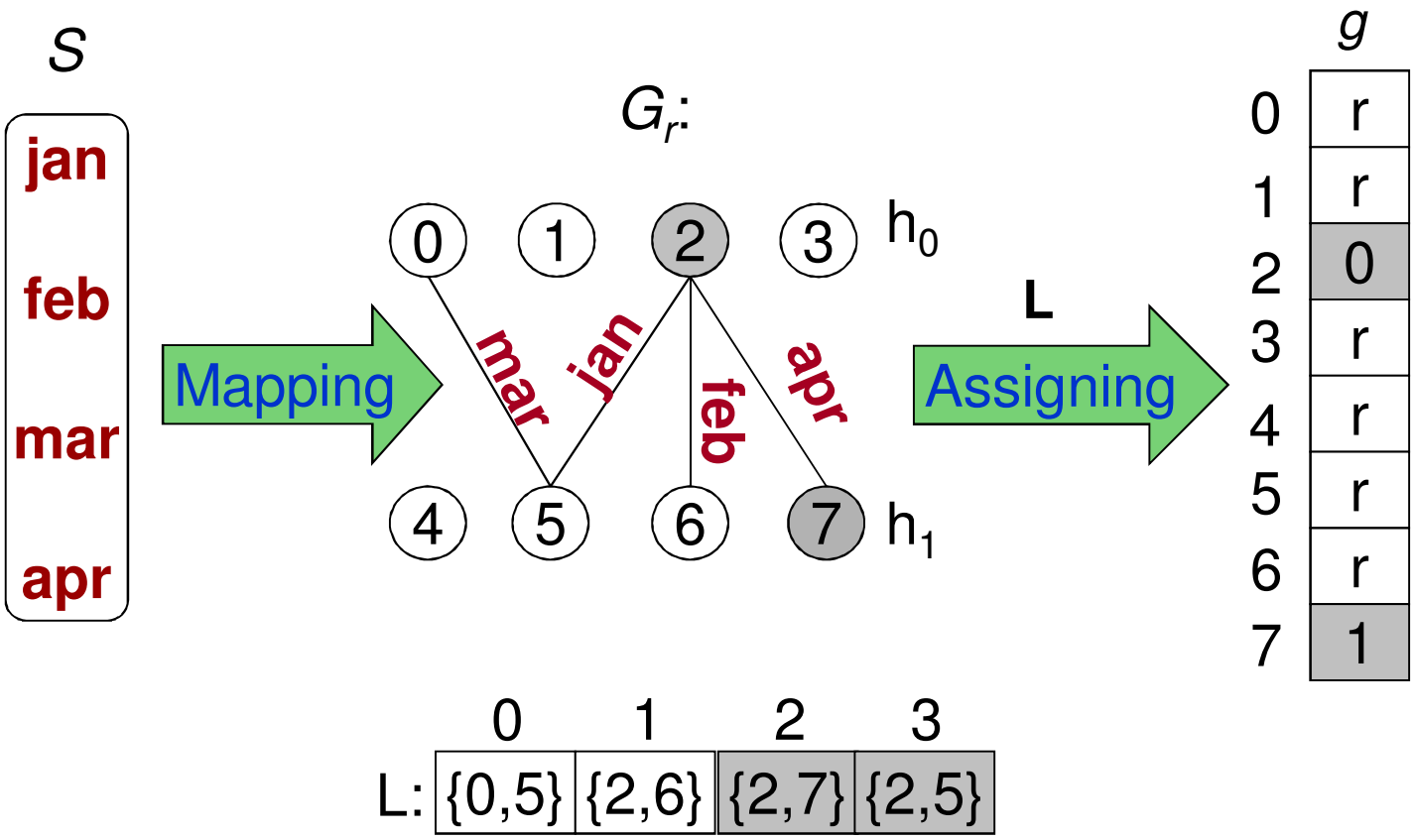
Internal Random Access Memory Algorithm (r=2)



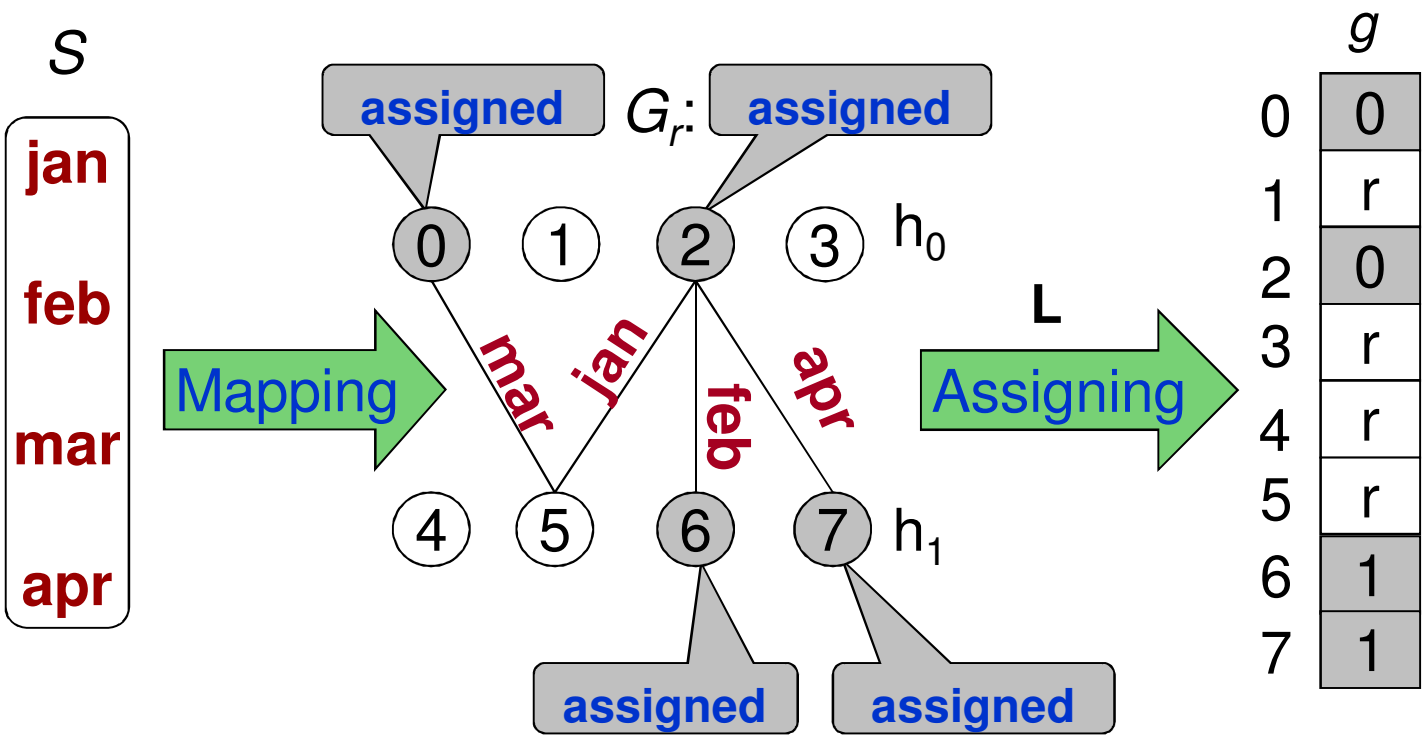
Internal Random Access Memory Algorithm (r=2)



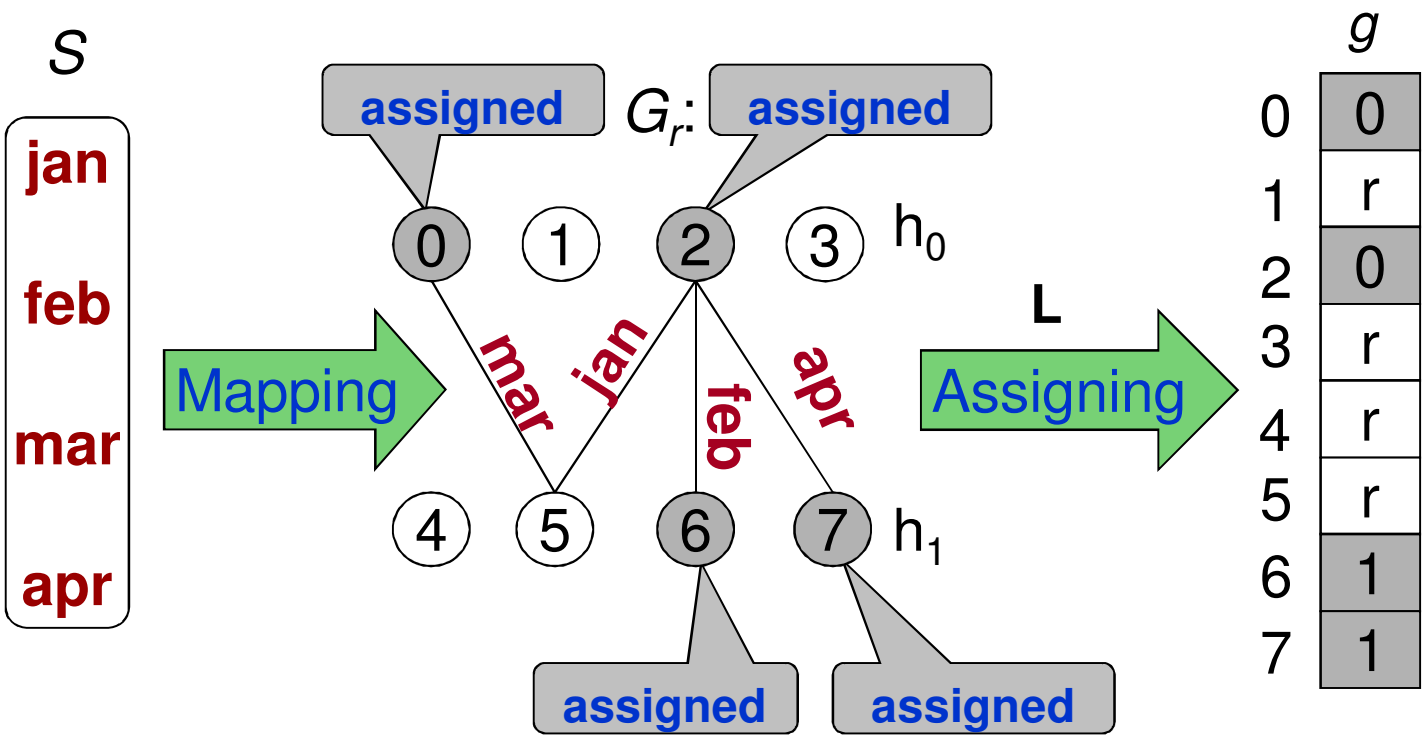
Internal Random Access Memory Algorithm (r=2)



Internal Random Access Memory Algorithm (r=2)

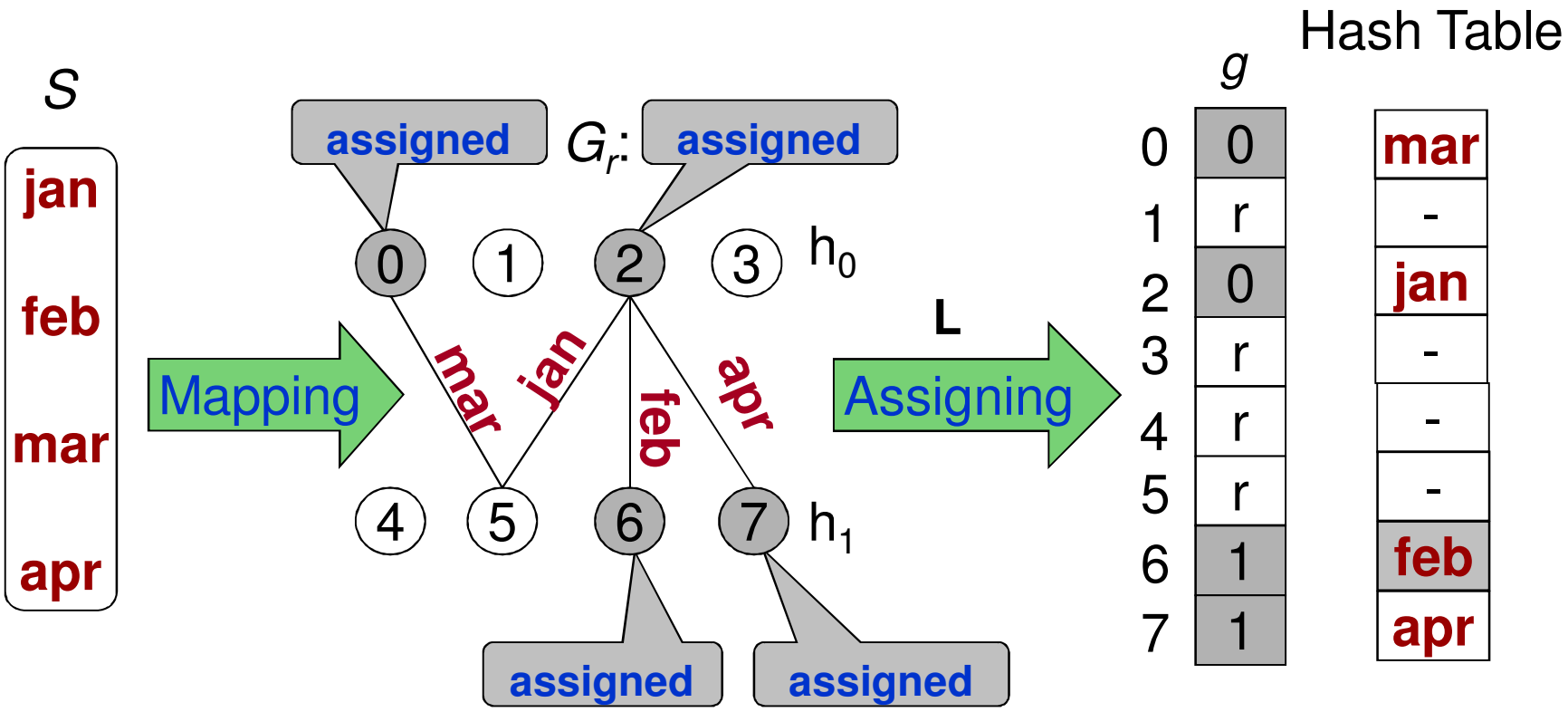


Internal Random Access Memory Algorithm (r=2)



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

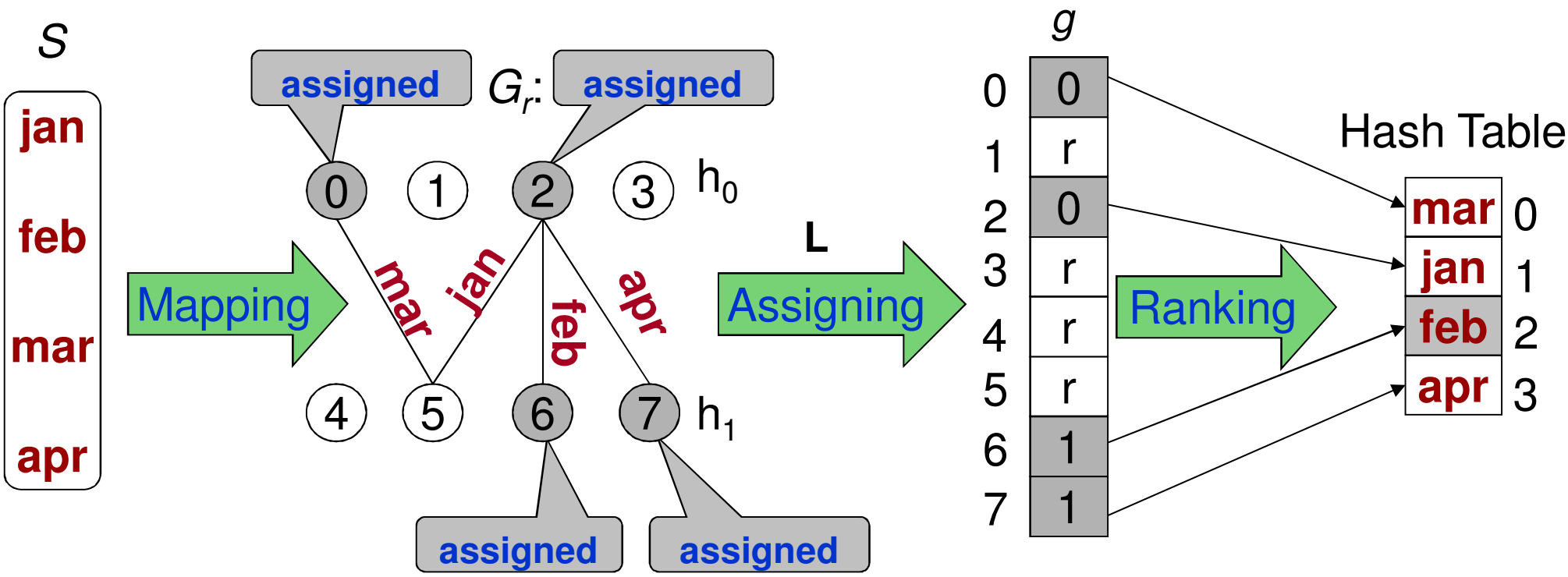
Internal Random Access Memory Algorithm: PHF



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

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

Internal Random Access Memory Algorithm: MPHF

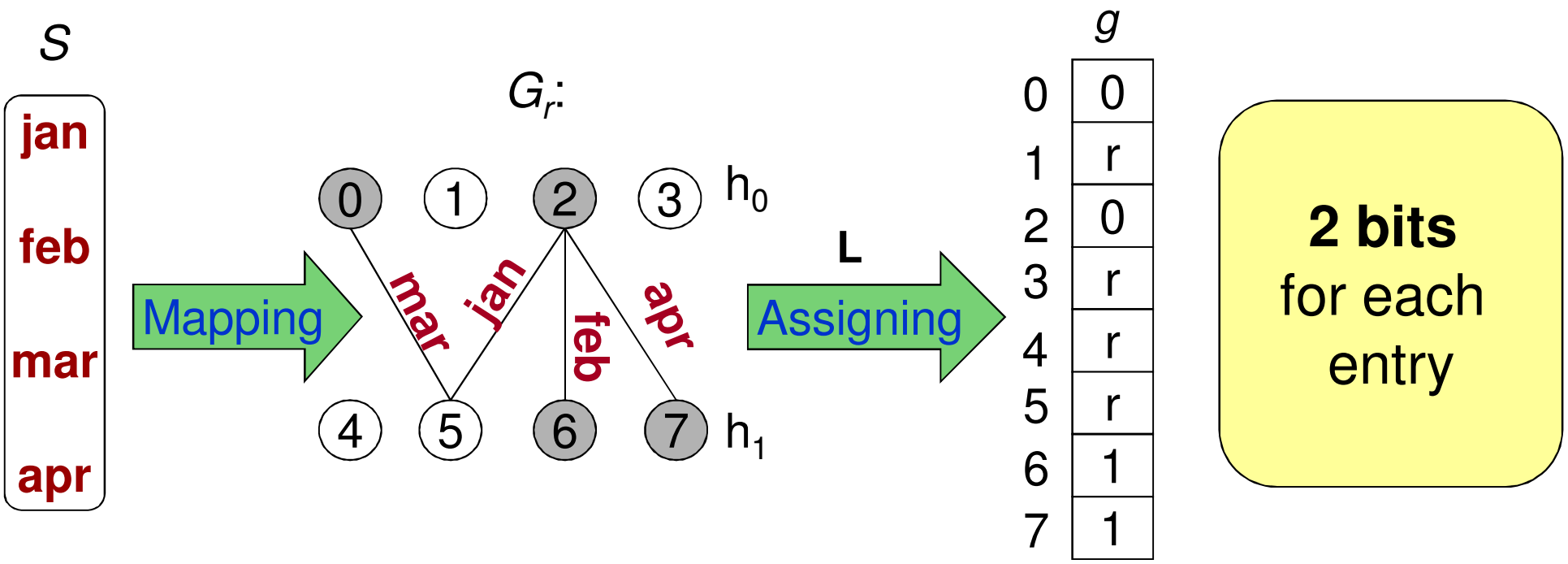


$i = (g[h_0(feb)] + g[h_1(feb)]) \bmod \mathbf{r} = (g[2] + g[6]) \bmod \mathbf{2} = 1$

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

$mphf(feb) = rank(phf(feb)) = rank(6) = 2$

Space to Represent the Function



Space to Represent the Functions (r = 3)

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

Use of Acyclic Random Hypergraphs

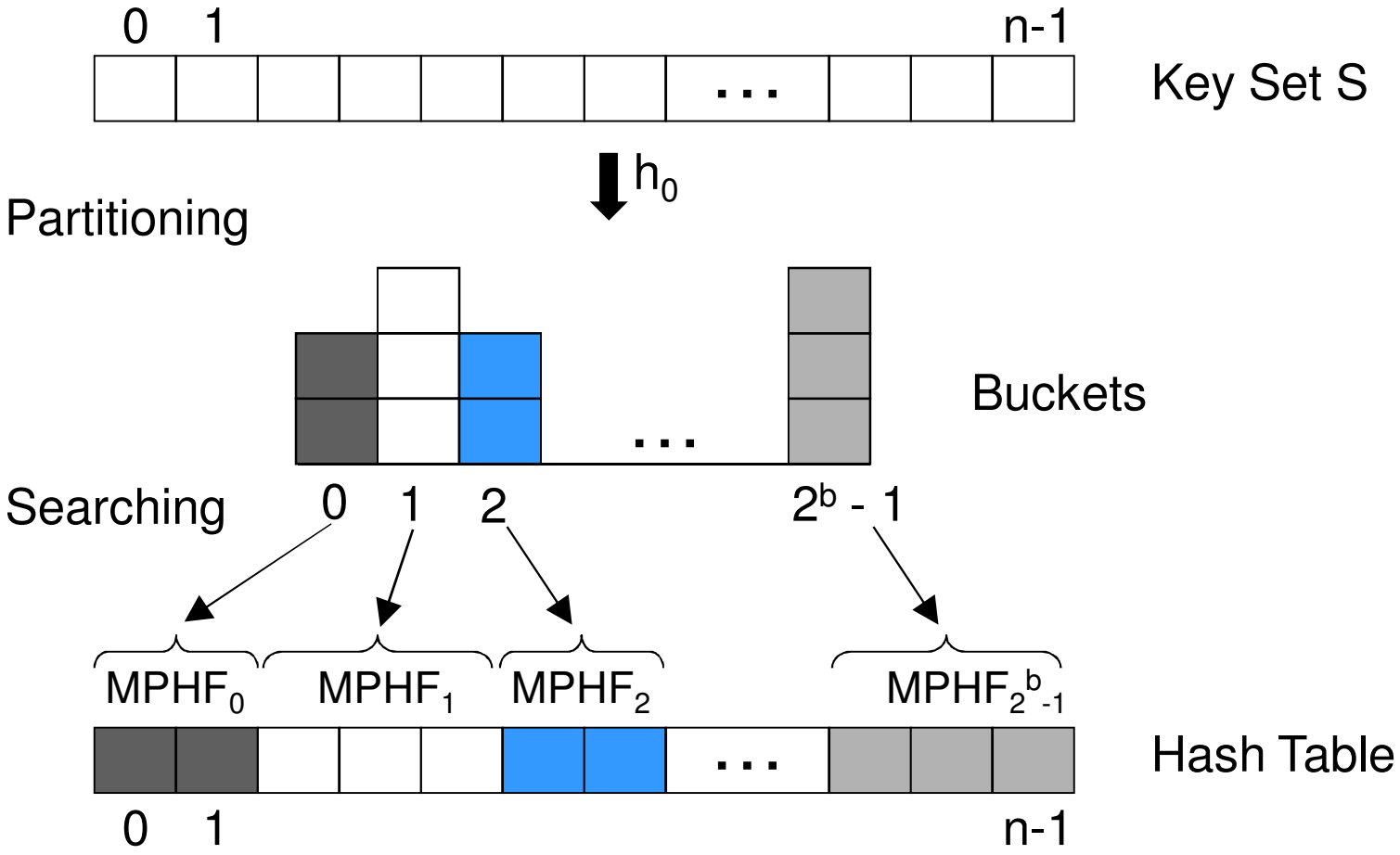
- Sufficient condition to work
- Repeatedly selects $h_0, h_1..., h_{r-1}$
- For r = 2, m = cn and c ≥ 2.09: $Pr_a = 0.29$
- For r = 3, m = cn and c≥1.23: Pr_a tends to 1
- Number of iterations is $1/Pr_a$
  - r = 2: 3.5 iterations
  - r = 3: 1.0 iteration

The External Memory Cache-Aware Algorithm ...

External Memory Algorithm (EM)

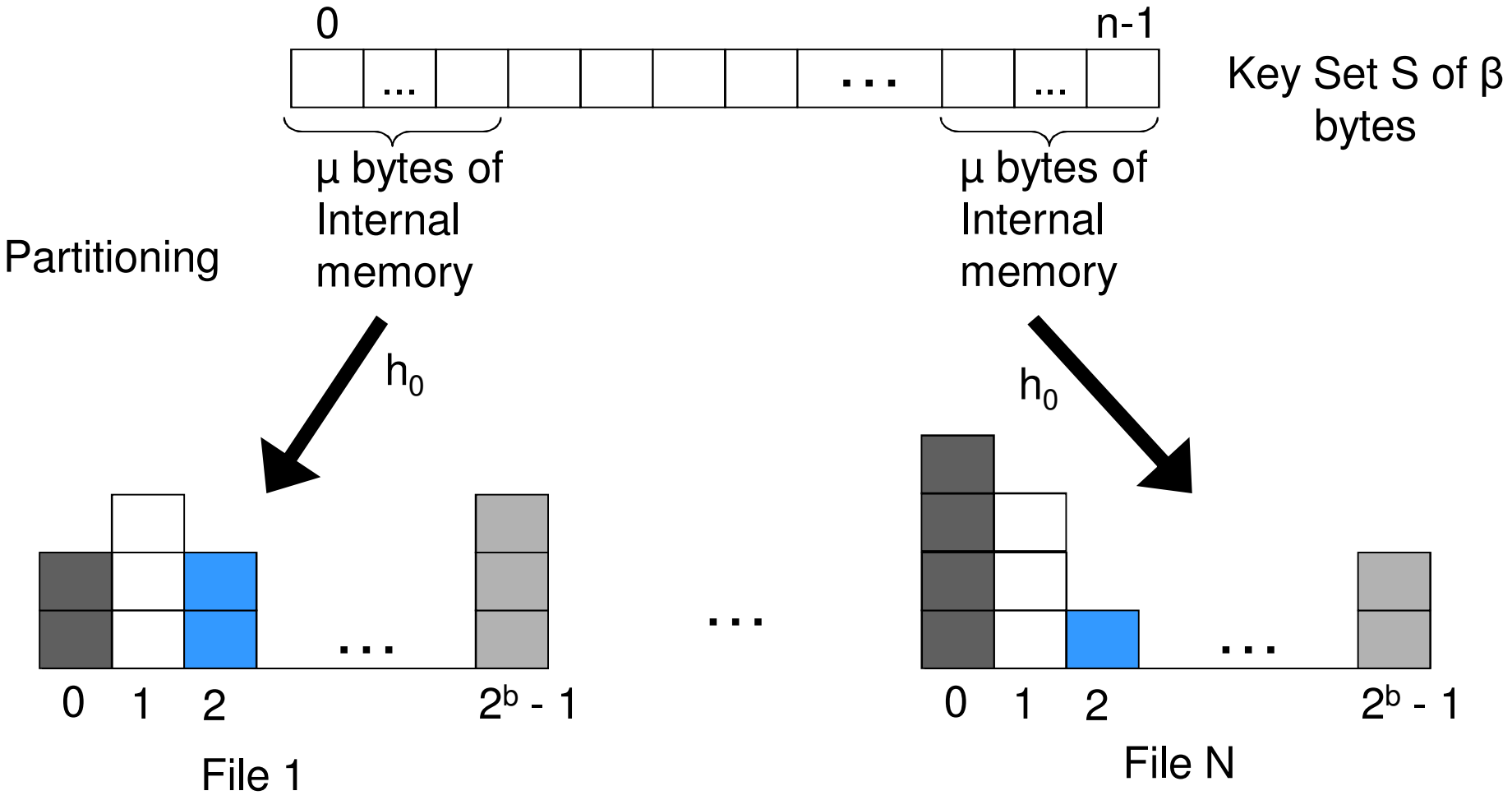
- First MPHF algorithm for very large key sets (in the order of billions of keys)
- This is possible because
  - Deals with external memory efficiently (cache-aware)
  - Generates compact functions (near space-optimal)
  - Uses a little amount of internal memory to scale
  - Works in linear time
- Two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

External Memory Algorithm (EM)



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

Key Set Does Not Fit In Internal Memory

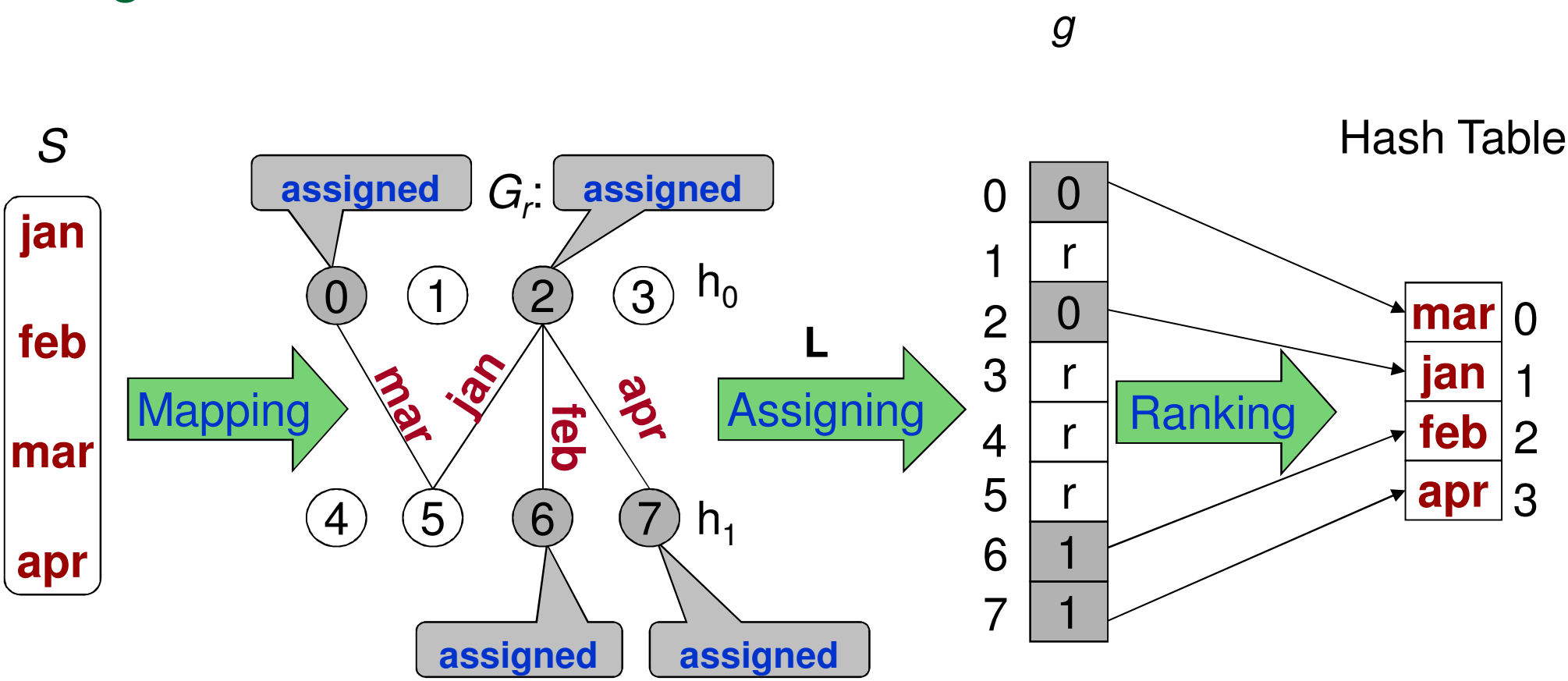


N = β/μ b = Number of bits of each bucket address Each bucket ≤ 256

Important Design Decisions

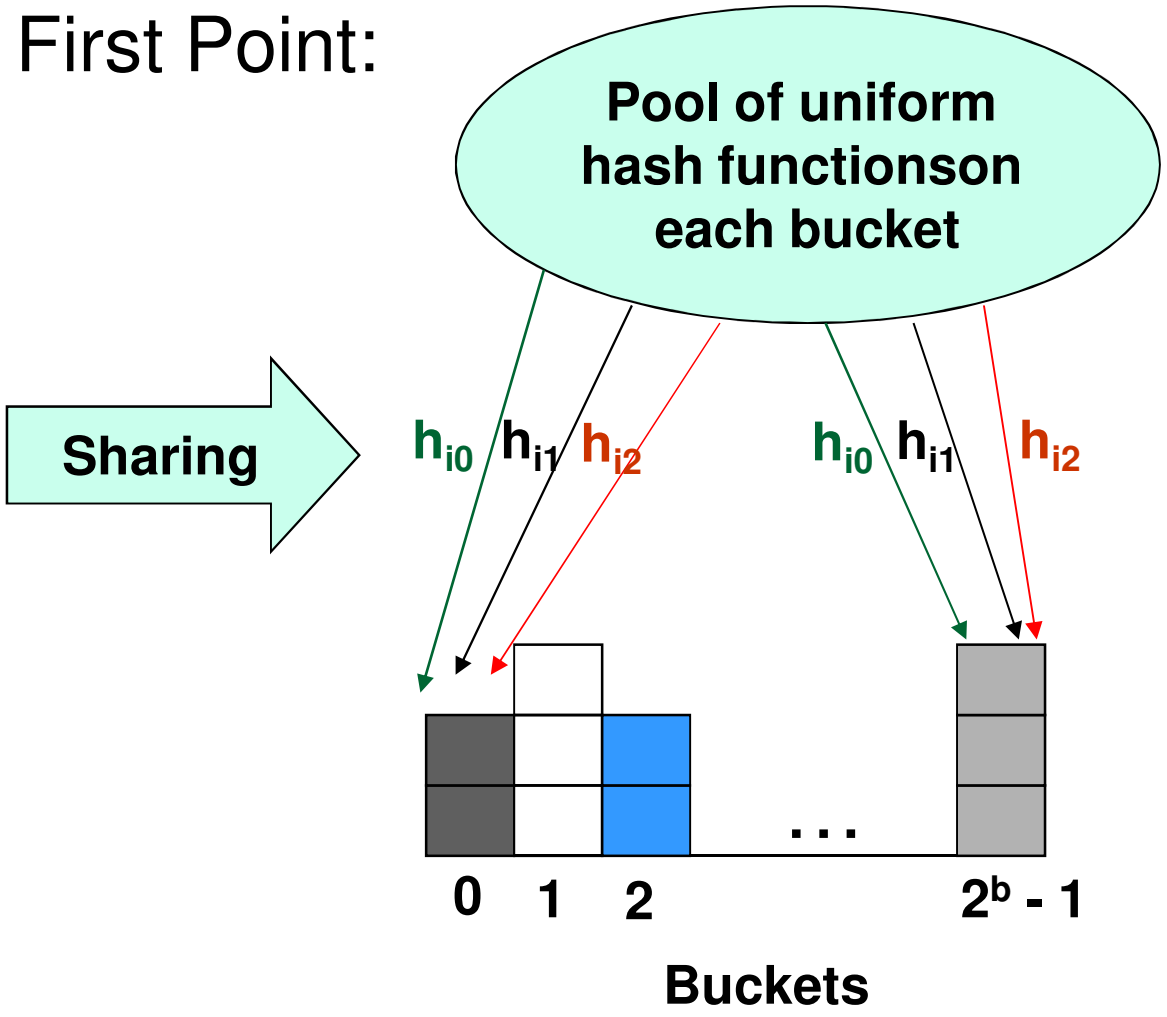
- We map long URLs to a fingerprint of fixed size using a hash function
- Use our RAM linear time and near-optimal space algorithm to generate the MPHF of each bucket
- How do we obtain a linear time complexity?
  - Using internal radix sorting to form the buckets
  - Using a heap of N entries to drive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

Use the Internal Random Access Memory Algorithm for Each Bucket



Why the EM Algorithm is Well-Founded?

First Point:



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)

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

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

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

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

Why the 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)

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

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

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

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

Pool

Why the 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)

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

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

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

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

Pool

Random numbers

Why the 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)

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 EM 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 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 architeture

Generation Time of MPHFs (in Minutes)

n (millions)	32	128	512	1024
Theoretic EM	1.3 ± 0.002	6.2 ± 0.02	27.6 ± 0.09	57.4 ± 0.06
Heuristic EM	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)
RAM (r = 3)	6.7 ± 0
Theoretic EM	9.0 ± 0.3
Heuristic EM	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)
RAM (r = 3)	6.7 ± 0	2.6
Theoretic EM	9.0 ± 0.3	3.3
Heuristic EM	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)
RAM (r = 3)	6.7 ± 0	2.6	2.1
Theoretic EM	9.0 ± 0.3	3.3	4.9
Heuristic EM	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:
  - 3,069 downloads (until March 16th, 2009)
  - Incorporated by Debian
- Library address: http://cmph.sourceforge.net

Comparing Our MPHF With Other Methods

- Open Addressing
- Chaining

Open Addressing

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

Chaining

- Use linked lists to resolve collisions
- New items can aways be added
- Methods used in comparisson
  - Chaining with move to front heuristics
  - Exact fit
  - Exact fit with move to front heuristics

Vocabularies

Collection	*n*	Shortest Key	Largest Key	Avg Key Size
AllTheWeb	5,424,923	2	31	17.46
URLs-37	37,294,116	8	496	58.77

Successful and Unsuccessful Searches

Collection	*n*	Average Key Size
AllTheWeb	10,000,000	17.46
URLs-37	250,000,000	58.77

- Successful searches
  - Follow real access patterns (power law)
  - Represents the best case
- Unsuccessful searches
  - Randomly generated keys
  - Uniform distribution

MPHF vs Linear, Quadratic and Double Hashing

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.89	2.62
LH	20	3.38	320	20	262.39	320
QH	20	3.41	320	20	262.72	320
DH	20	3.46	320	20	262.50	320

MPHF vs Linear, Quadratic and Double Hashing

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.89	2.62
LH	20	3.38	320	20	262.39	320
QH	20	3.41	320	20	262.72	320
DH	20	3.46	320	20	262.50	320

MPHF vs Linear, Quadratic and Double Hashing

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.89	2.62
LH	20	3.38	320	20	262.39	320
QH	20	3.41	320	20	262.72	320
DH	20	3.46	320	20	262.50	320

MPHF vs Linear, Quadratic and Double Hashing

Unsucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.23	2.62	100	271.42	2.62
LH	20	3.92	320	20	271.61	320
QH	20	3.79	320	20	271.92	320
DH	20	3.85	320	20	271.78	320

MPHF vs Cuckoo Hashing

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.50	2.62
CH	50	4.06	128	50	264.54	128
CH	40	3.92	160	40	263.86	160
CH	30	3.76	213	30	263.30	213
CH	20	3.67	320	20	263.20	320

MPHF vs Cuckoo Hashing

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.50	2.62
CH	50	4.06	128	50	264.54	128
CH	40	3.92	160	40	263.86	160
CH	30	3.76	213	30	263.30	213
CH	20	3.67	320	20	263.20	320

MPHF vs Cuckoo Hashing

Unsucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.23	2.62	100	271.42	2.62
CH	50	5.13	128	50	281.23	128
CH	40	4.99	160	40	275.80	160
CH	30	4.80	213	30	273.02	213
CH	20	4.70	320	20	272.52	320

MPHF vs Hopscotch, Sparse and Dense Hashing

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.89	2.62
HoH	65	3.32	83.69	56	262.20	107.43
SH	65	3.95	4.10	56	262.84	4.76
DeH	65	2.88	98.46	56	262.13	114.29

MPHF vs Hopscotch, Sparse and Dense Hashing

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.89	2.62
HoH	65	3.32	83.69	56	262.20	107.43
SH	65	3.95	4.10	56	262.84	4.76
DeH	65	2.88	98.46	56	262.13	114.29

MPHF vs Hopscotch, Sparse and Dense Hashing

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.89	2.62
HoH	65	3.32	83.69	56	262.20	107.43
SH	65	3.95	4.10	56	262.84	4.76
DeH	65	2.88	98.46	56	262.13	114.29

MPHF vs Hopscotch, Sparse and Dense Hashing

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.89	2.62
HoH	65	3.32	83.69	56	262.20	107.43
SH	65	3.95	4.10	56	262.84	4.76
DeH	65	2.88	98.46	56	262.13	114.29

MPHF vs Hopscotch, Sparse and Dense Hashing

Unsucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.23	2.62	100	271.42	2.62
HoH	65	3.88	83.69	56	271.51	107.43
SH	65	4.60	4.10	56	271.76	4.76
DeH	65	3.84	98.46	56	271.75	114.29

MPHF vs Chaining Methods

Sucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.07	2.62	100	261.89	2.62
CHMTF	40	3.56	224.00	40	262.70	224.00
EFH	100	3.71	80.00	200	262.75	40.00
EFHMTF	200	3.64	40.00	200	262.66	40.00

MPHF vs Chaining Methods

Unsucessful searches

Data Struc.	AllTheWeb			URLs-37		
	α(%)	T(s)	S_o	α(%)	T(s)	S_o
MPH	100	2.23	2.62	100	271.42	2.62
CHMTF	40	4.02	224.00	40	271.66	224.00
EFH	40	4.09	200.00	40	271.40	200.00
EFHMTF	40	4.06	200.00	40	271.45	200.00

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

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