A Highly Scalable Perfect Hashing Algorithm

Nivio Ziviani

Fabiano C. Botelho

Department of Computer Science

Federal University of Minas Gerais, Brazil

3rd Intl. Conference on Scalable Information Systems

Naples, Italy, June 4-6, 2008

INFOSCALE 2008, June 4-6, Vico Equense, Italy
Copyright © 2008 978-963-9799-28-8
DOI 10.4108/infoscale.2008.28

Where is Belo Horizonte?

Pampulha's Church
Oscar Niemeyer

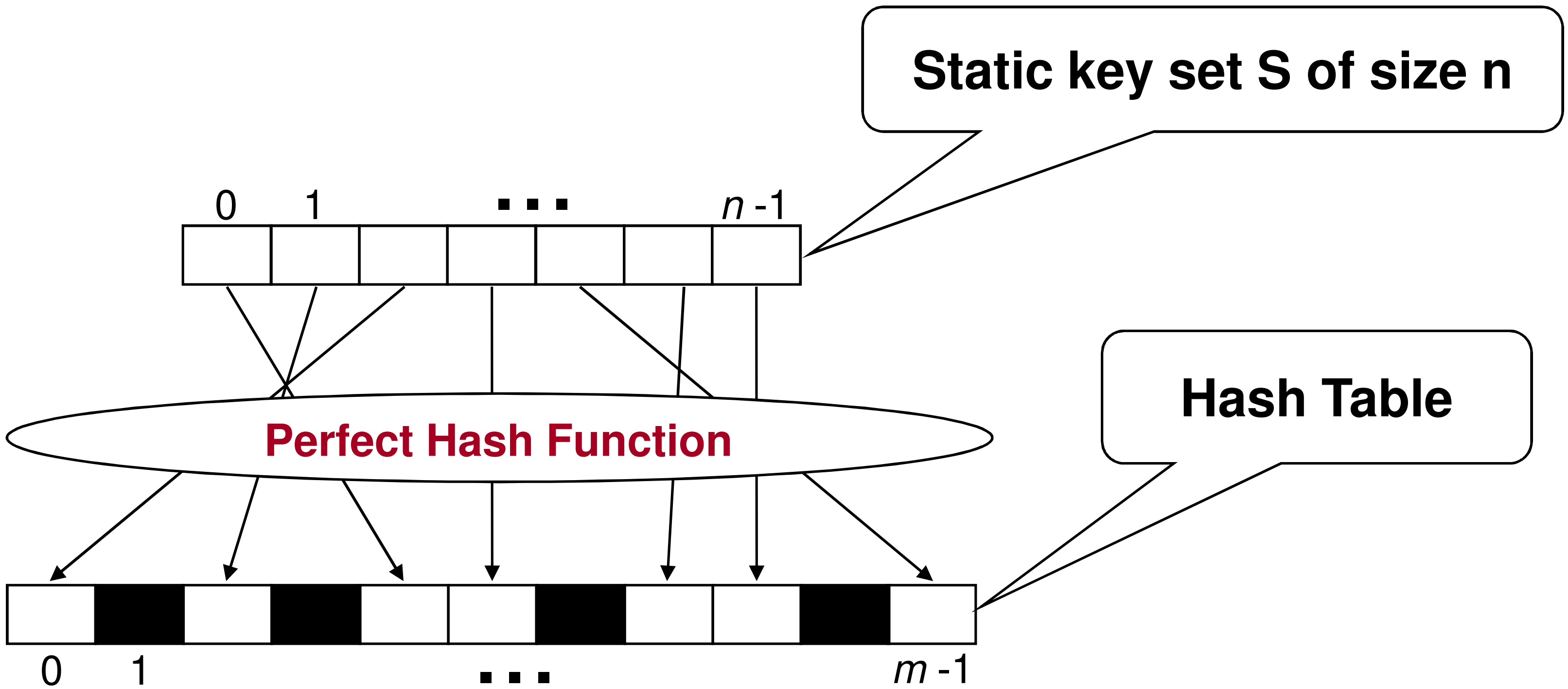
Objective of the Presentation

Present a perfect hashing algorithm:

- Sequential construction of the function
- Distributed construction of the function
- Description and evaluation of the function:
  - *Centralized* in one machine
  - *Distributed* among the participating machines

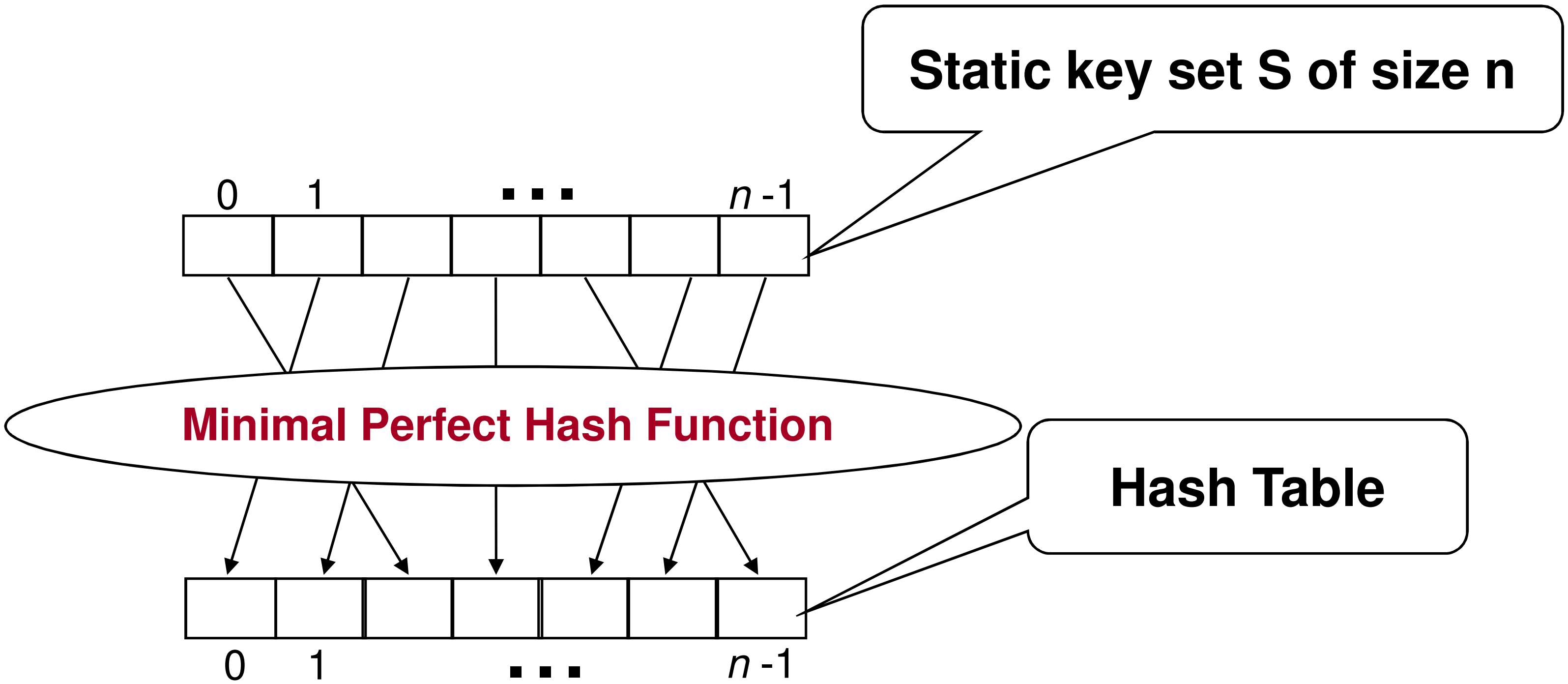
Algorithm is highly scalable, time efficient and near space-optimal

Perfect Hash Function



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

Minimal Perfect Hash Function



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

The Algorithm

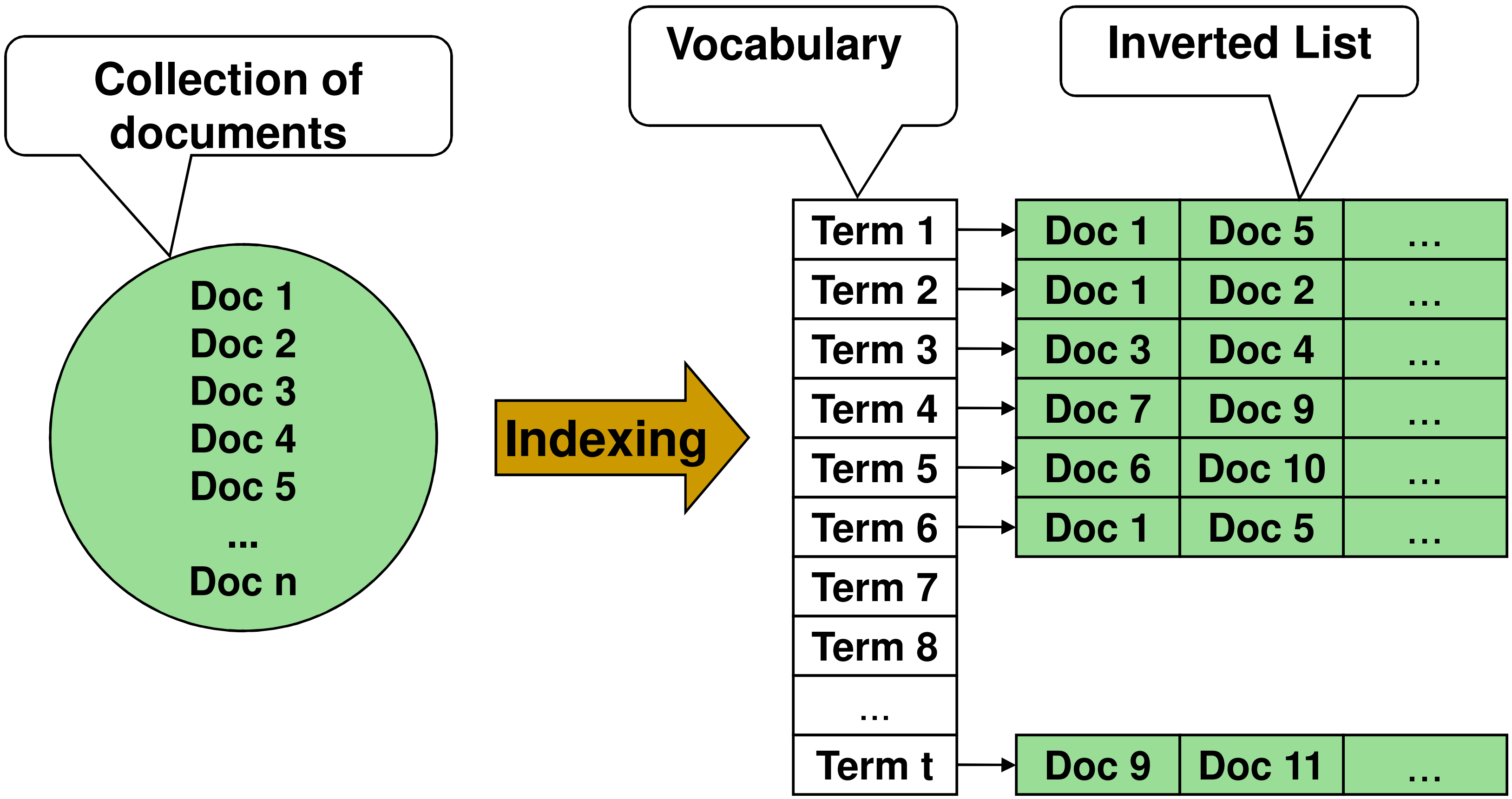
A perfect hashing algorithm that uses the idea of partitioning the input key set into small buckets:

- Key set fits in the internal memory
  - **I**nternal **R**andom **A**ccess memory algorithm
- Key set larger than the internal memory
  - **E**xternal **C**ache-**A**ware memory algorithm

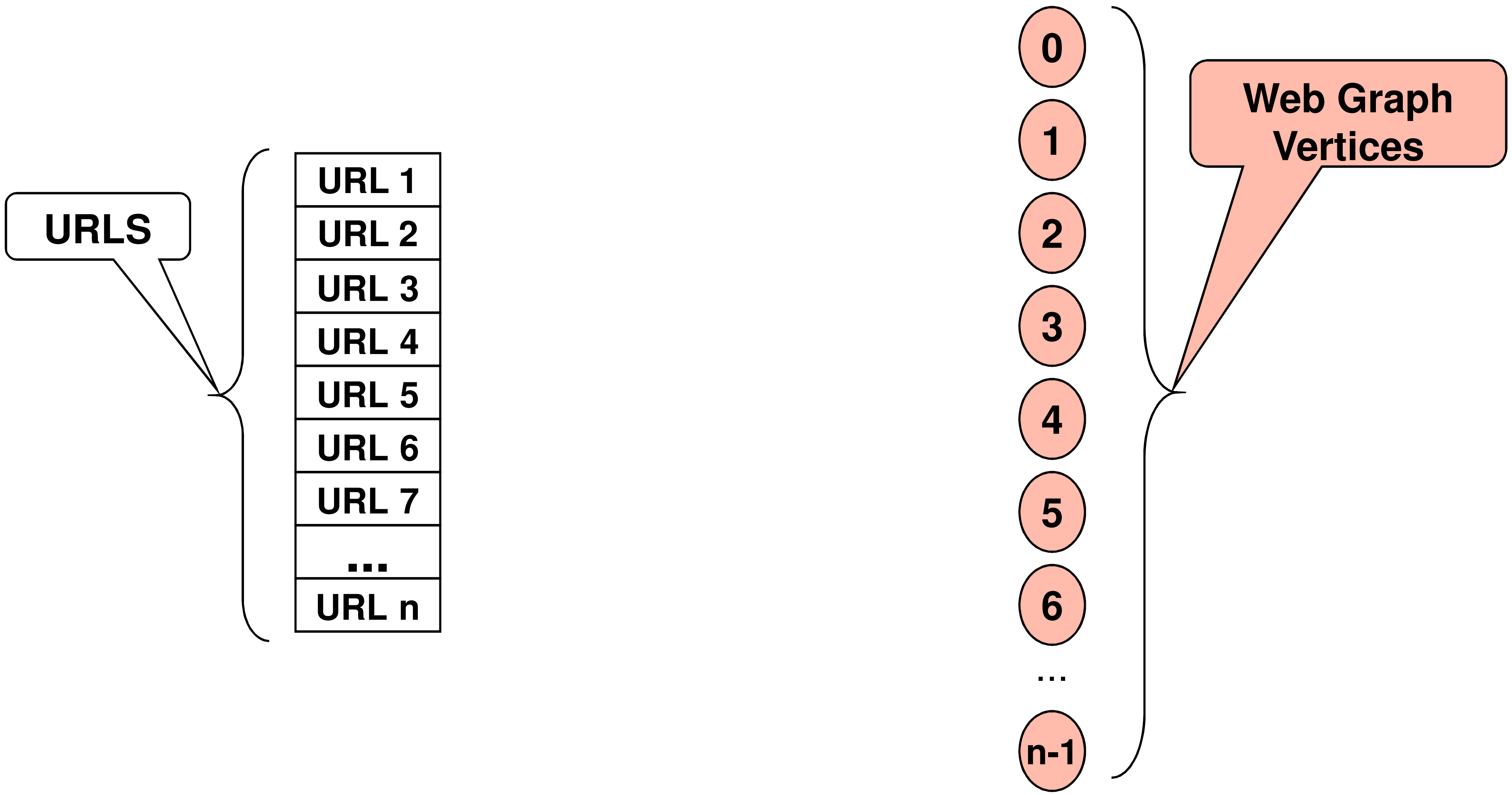
Where to use a PHF or a MPHF?

- 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

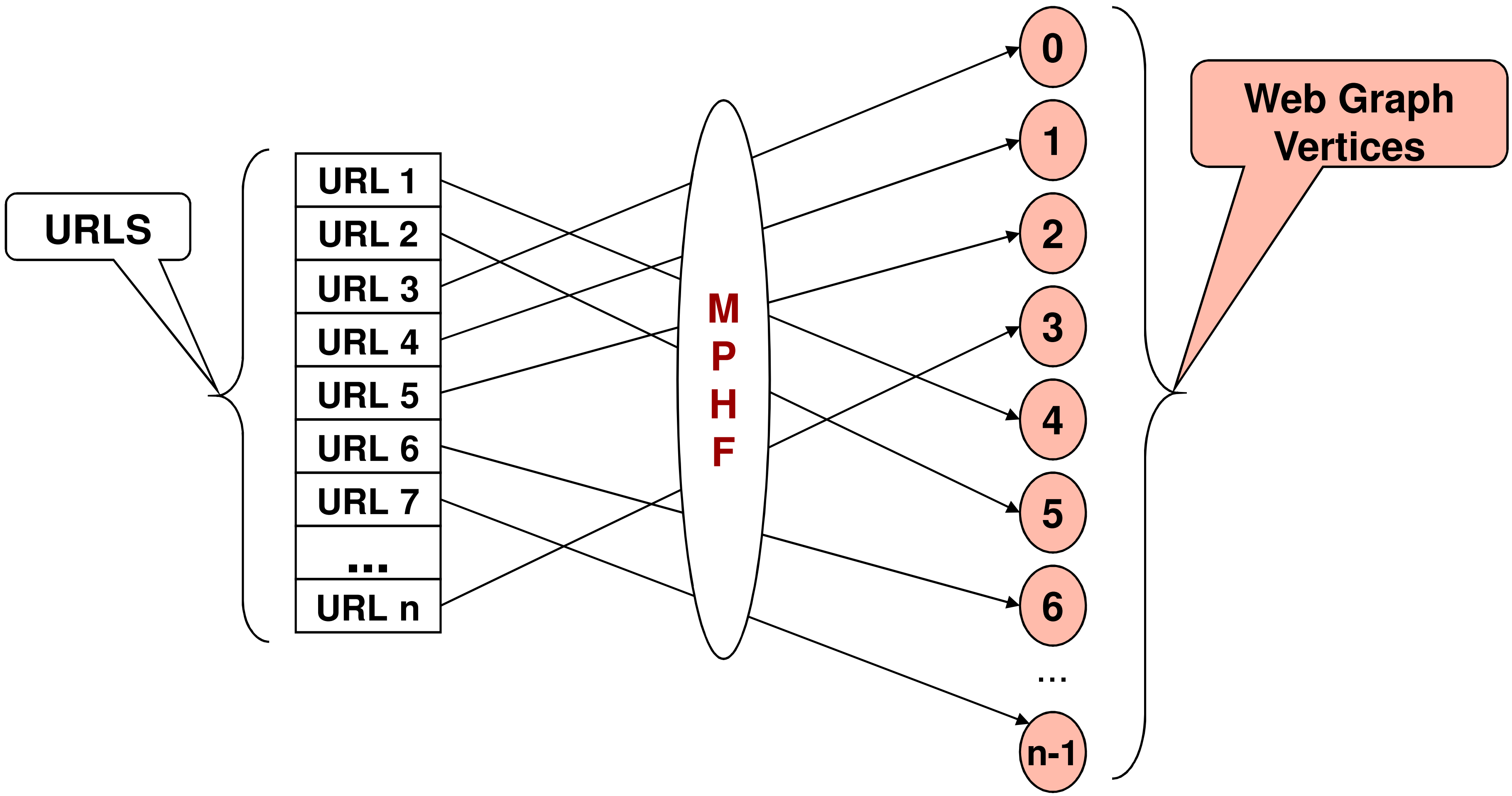
Indexing: Representing the Vocabulary



Mapping URLs to Web Graph Vertices



Mapping URLs to Web Graph Vertices



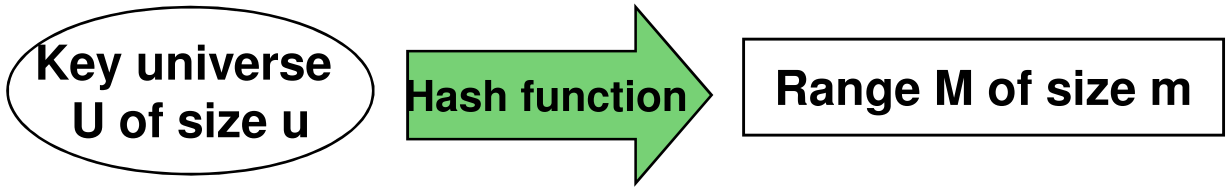
Information Theoretical Lower Bounds for Storage Space

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

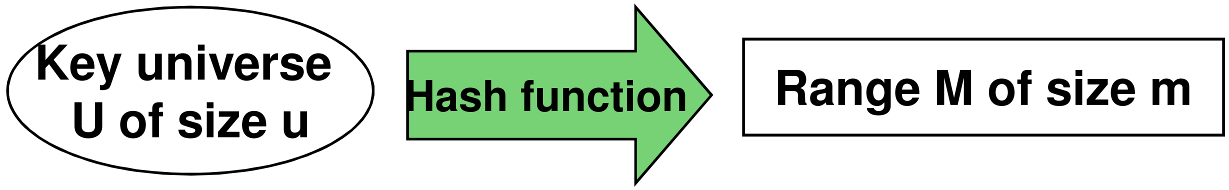
m < 3n

$\log e \approx 1.4427$

Uniform Hashing Versus Universal Hashing



Uniform Hashing Versus Universal Hashing



Uniform hashing

- # of functions from U to M?

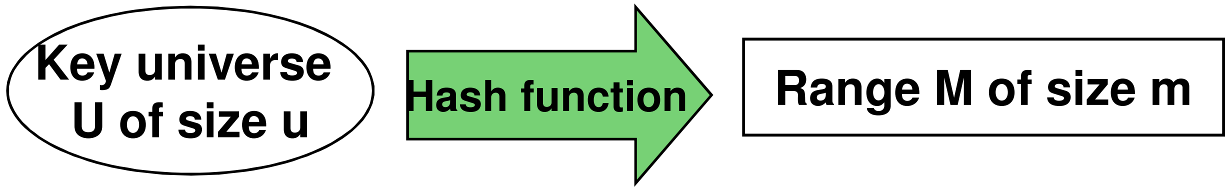
$$m^u$$

- # 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

  (use universal hashing - assume uniform hashing for free)

- Heuristics

Theoretical Results

Use Complete Randomness (Uniform Hash Functions)

Work	Gen. Time	Eval. Time	Size (bits)
Mehlhorn (1984)	Expon.	Expon.	O(n)
Hagerup and Tholey (2001)	O(n+log log u)	O(1)	O(n)

Theoretical Results

Use Complete Randomness (Uniform Hash Functions)

Work	Gen. Time	Eval. Time	Size (bits)
Mehlhorn (1984)	Expon.	Expon.	O(n)
Hagerup & Tholey (2001)	O(n+log log u)	O(1)	O(n)
Botelho & Ziviani (CIKM 2007)	**O(n)**	**O(1)**	**O(n)**

Practical Results

Assume Uniform Hashing for Free (Use Universal Hashing)

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)**

Practical Results

Assume Uniform Hashing for Free (Use Universal Hashing)

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)**
Botelho, Pagh, Ziviani (WADs 2007)	**O(n)**	**O(1)**	**O(n)**

Practical Results

Assume Uniform Hashing for Free (Use Universal Hashing)

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)**
Botelho, Pagh, Ziviani (WADs 2007)	**O(n)**	**O(1)**	**O(n)**
Botelho & Ziviani (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)**

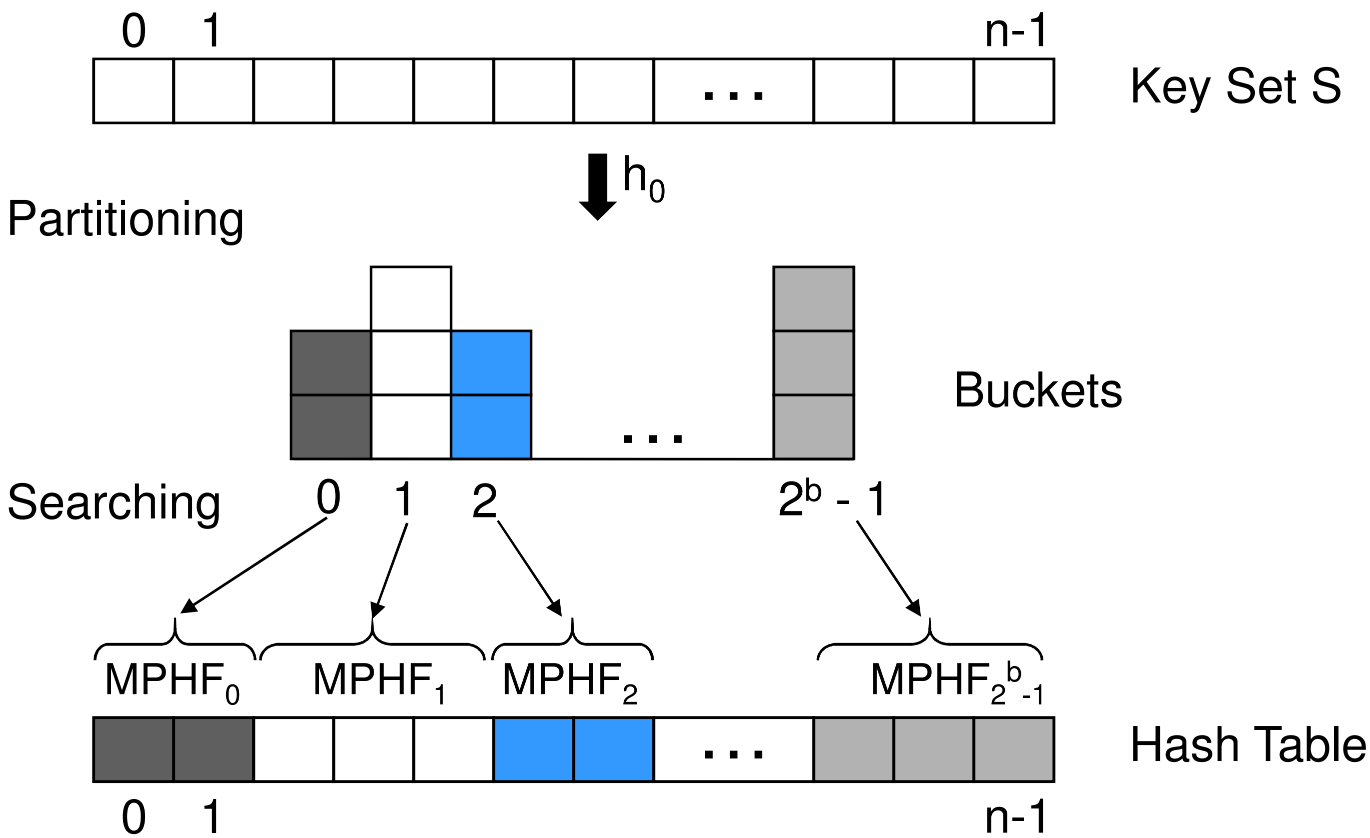
The Sequential

External Cache-Aware Algorithm...

External Cache-Aware Memory Algorithm

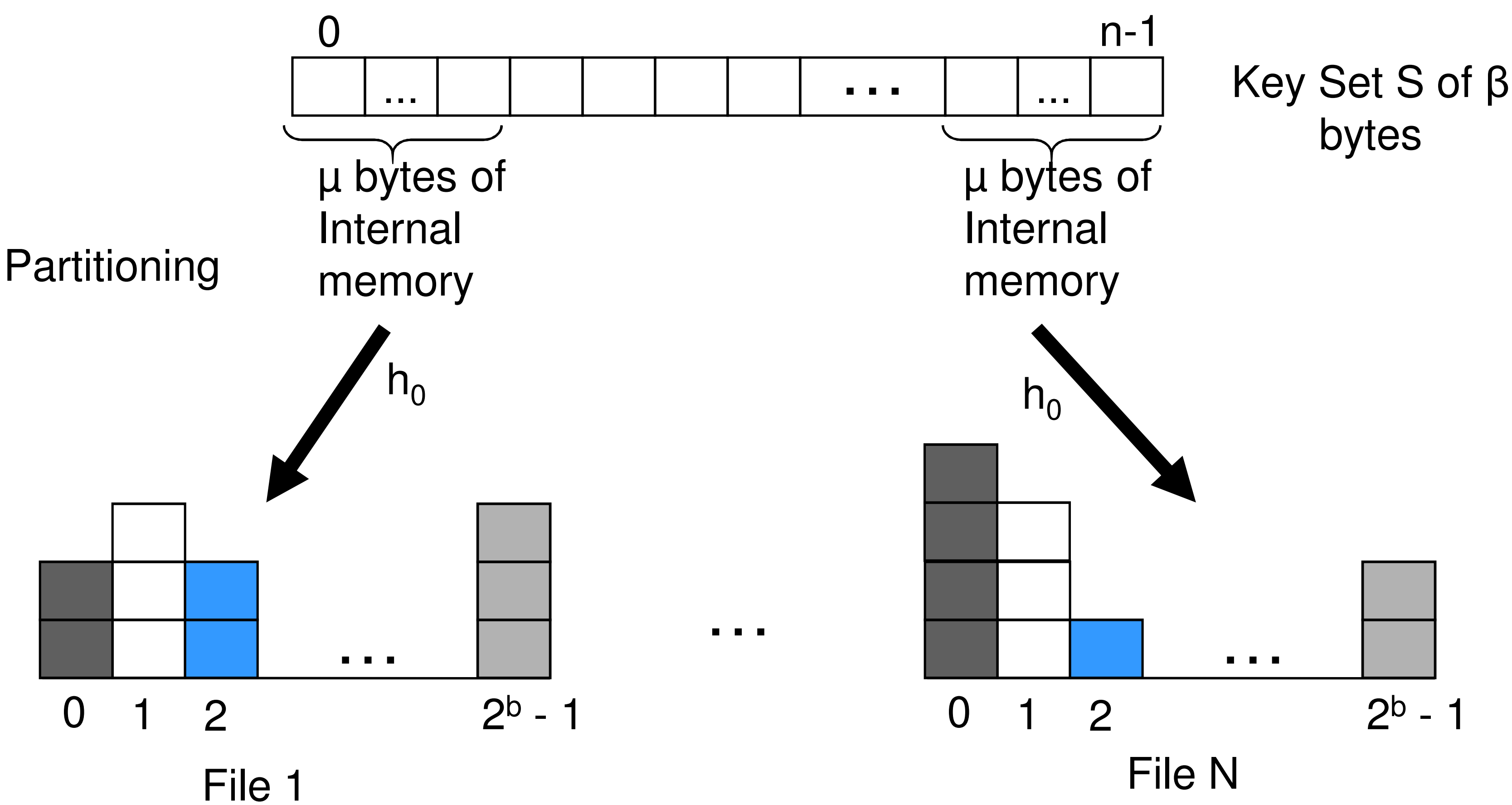
- First MPHF algorithm for very large key sets (in the order of billions of keys)
- This is possible because
  - Deals with external memory efficiently
  - Works in linear time
  - Generates compact functions (near space-optimal)
    - MPHF ($m = n$): $3.3n$ bits
    - PHF ($m = 1.23n$): $2.7n$ bits
    - Theoretical lower bound:
      - MPHF: $1.44n$ bits
      - PHF: $0.89n$ bits

Sequential External Perfect Hashing Algorithm



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

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 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

Algorithm Used for the Buckets:

Internal Random Access Memory Algorithm...

Internal Random Access Memory Algorithm

- 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

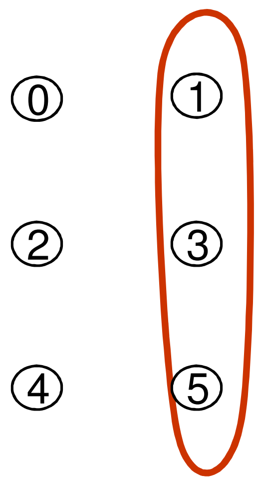
2 3

4 5

- 3-graph is induced by three uniform hash functions

Random Hypergraphs (r-graphs)

- 3-graph:

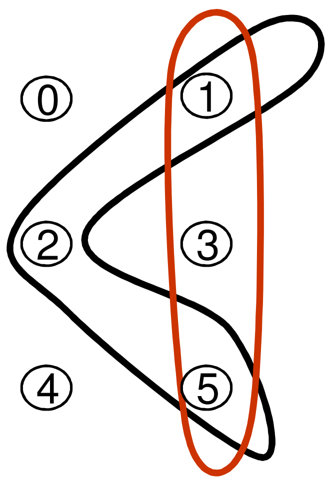


$h_0(jan) = 1 \quad h_1(jan) = 3 \quad h_2(jan) = 5$

- 3-graph is induced by three uniform hash functions

Random Hypergraphs (r-graphs)

- 3-graph:



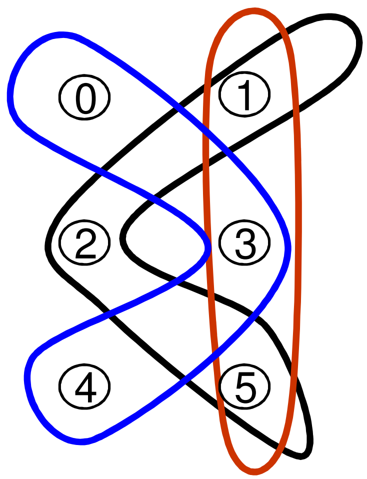
h_0(jan) = 1 h_1(jan) = 3 h_2(jan) = 5

h_0(feb) = 1 h_1(feb) = 2 h_2(feb) = 5

- 3-graph is induced by three uniform hash functions

Random Hypergraphs (r-graphs)

- 3-graph:



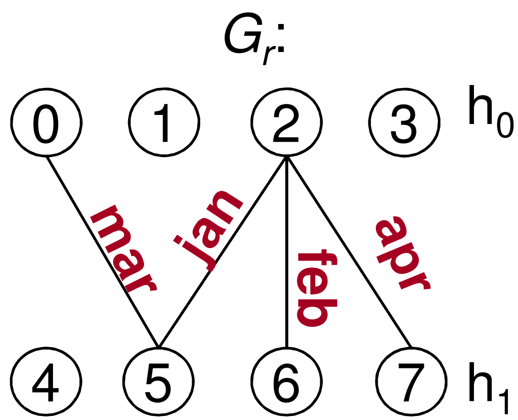
h_0(jan) = 1 h_1(jan) = 3 h_2(jan) = 5

h_0(feb) = 1 h_1(feb) = 2 h_2(feb) = 5

h_0(mar) = 0 h_1(mar) = 3 h_2(mar) = 4

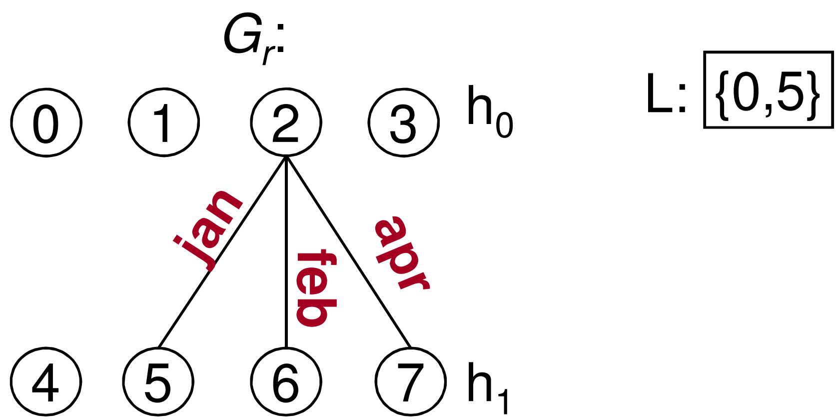
- 3-graph is induced by three uniform hash functions
- Our best result uses 3-graphs

Acyclic 2-graph

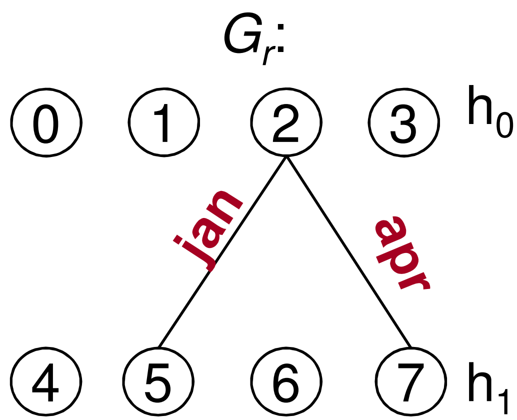


L:Ø

Acyclic 2-graph

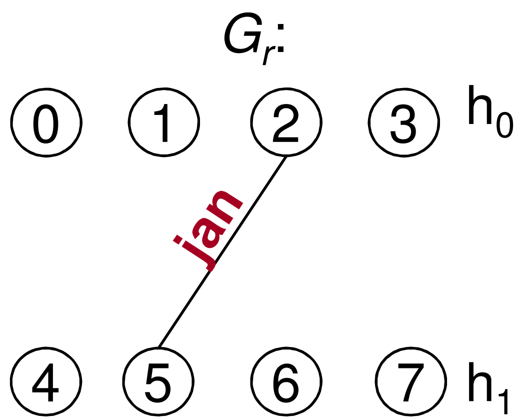


Acyclic 2-graph



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

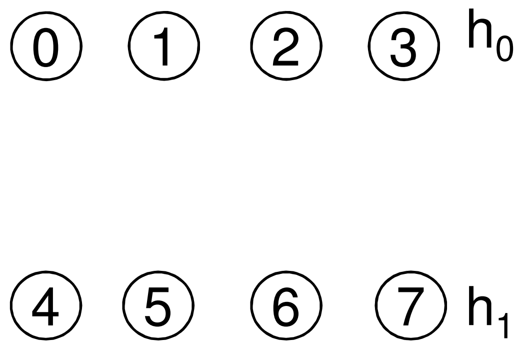
Acyclic 2-graph



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

Acyclic 2-graph

G_r:



G_r is acyclic

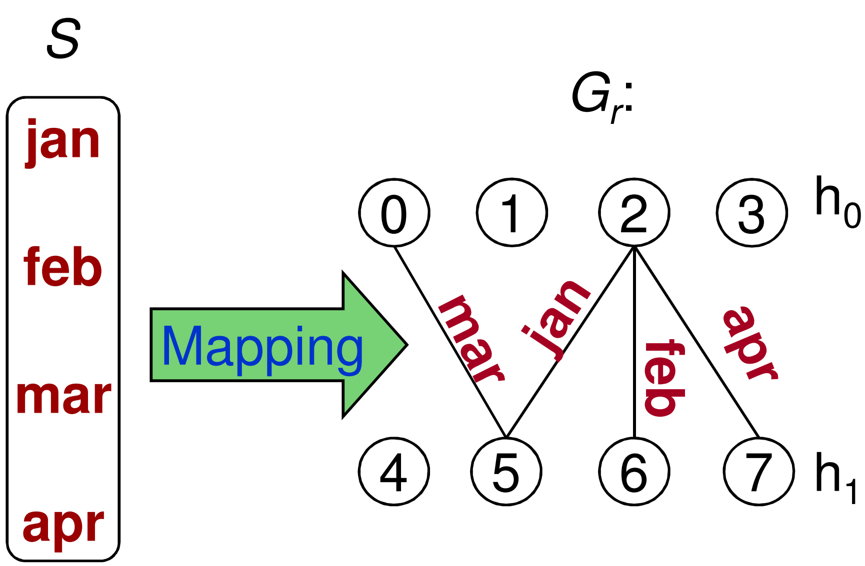
	0	1	2	3
L:	{0,5}	{2,6}	{2,7}	{2,5}

Internal Random Access Memory Algorithm (r=2)

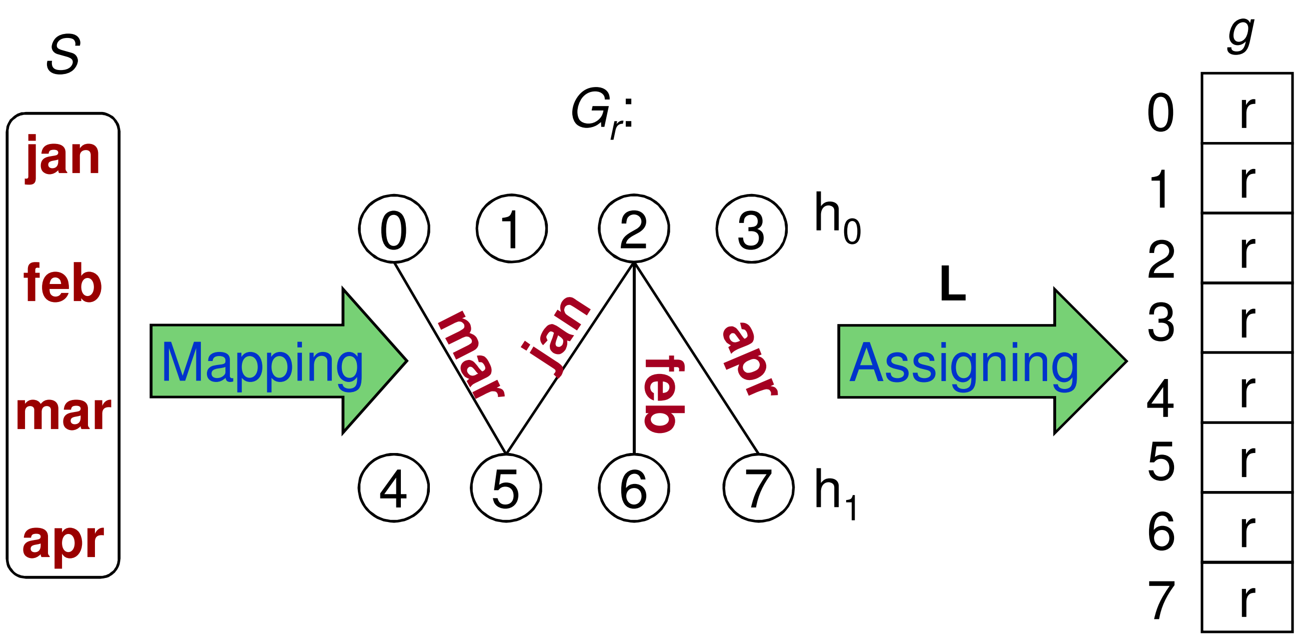
S

jan
feb
mar
apr

Internal Random Access Memory Algorithm (r=2)

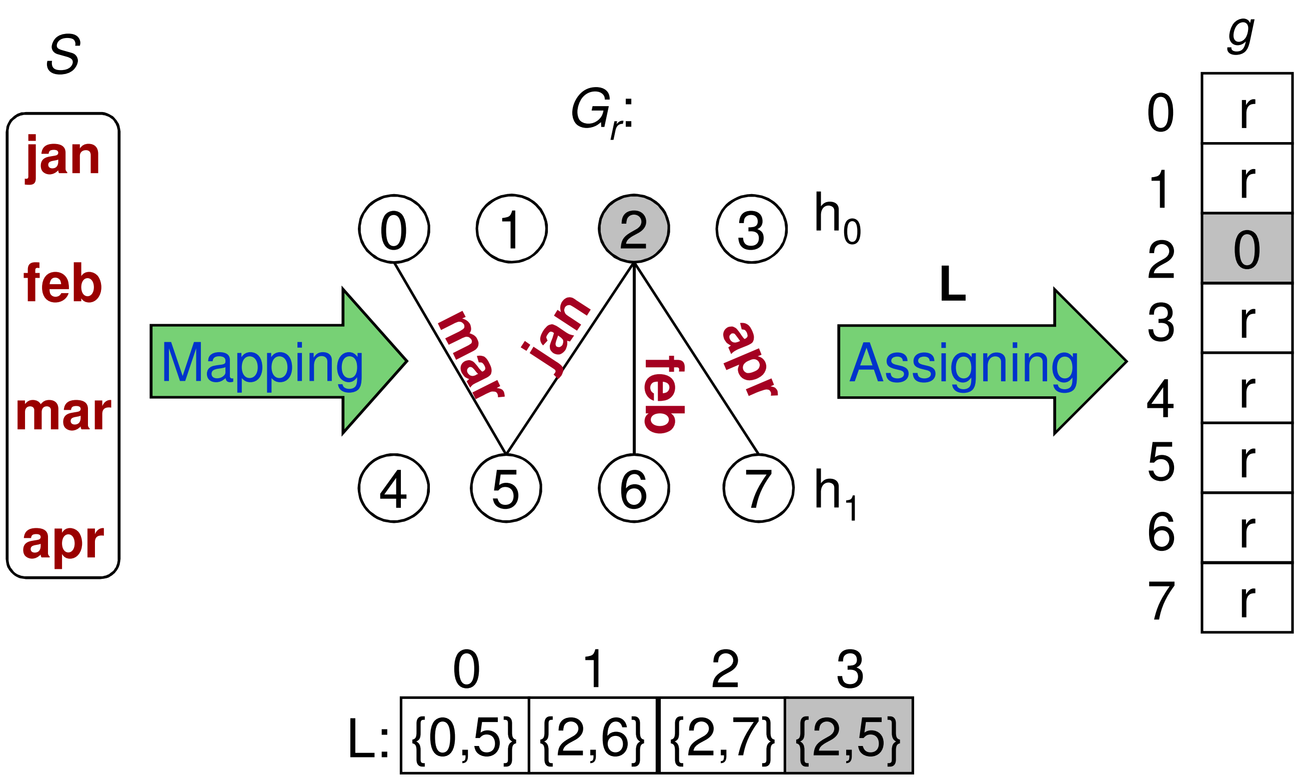


Internal Random Access Memory Algorithm (r=2)

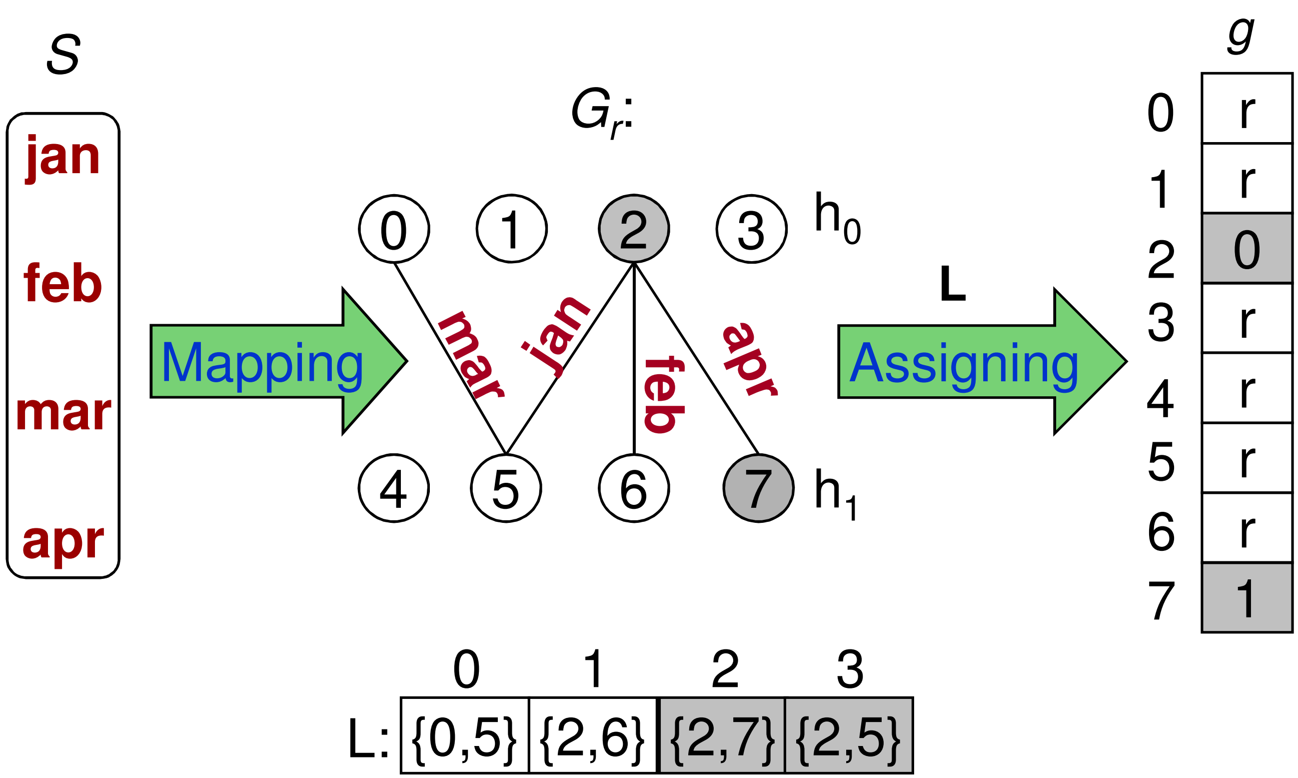


	0	1	2	3
L:	{0,5}	{2,6}	{2,7}	{2,5}

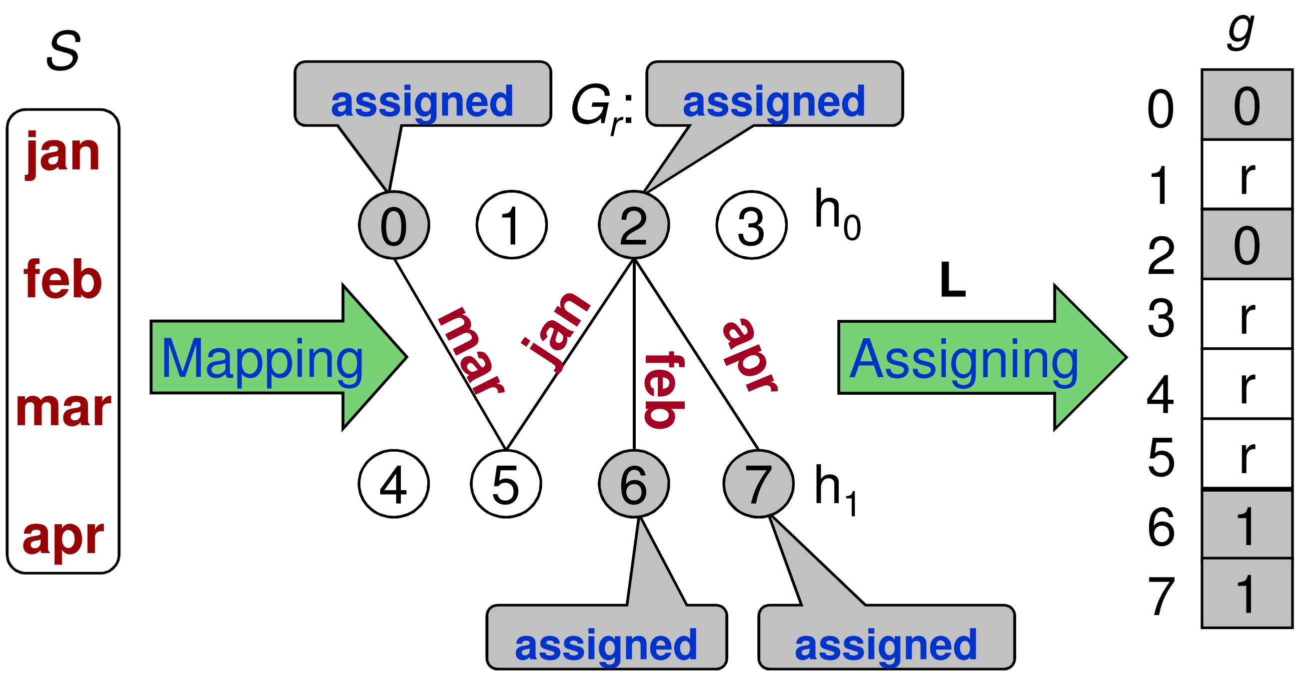
Internal Random Access Memory Algorithm (r=2)



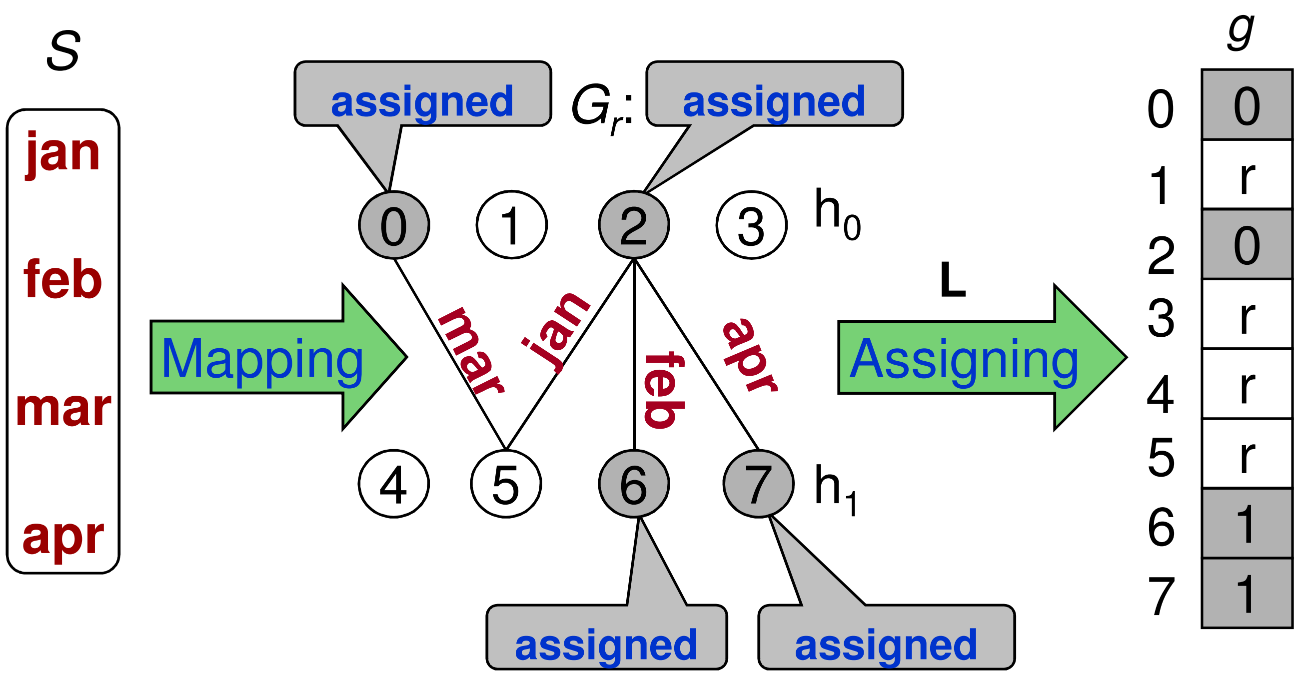
Internal Random Access Memory Algorithm (r=2)



Internal Random Access Memory Algorithm (r=2)

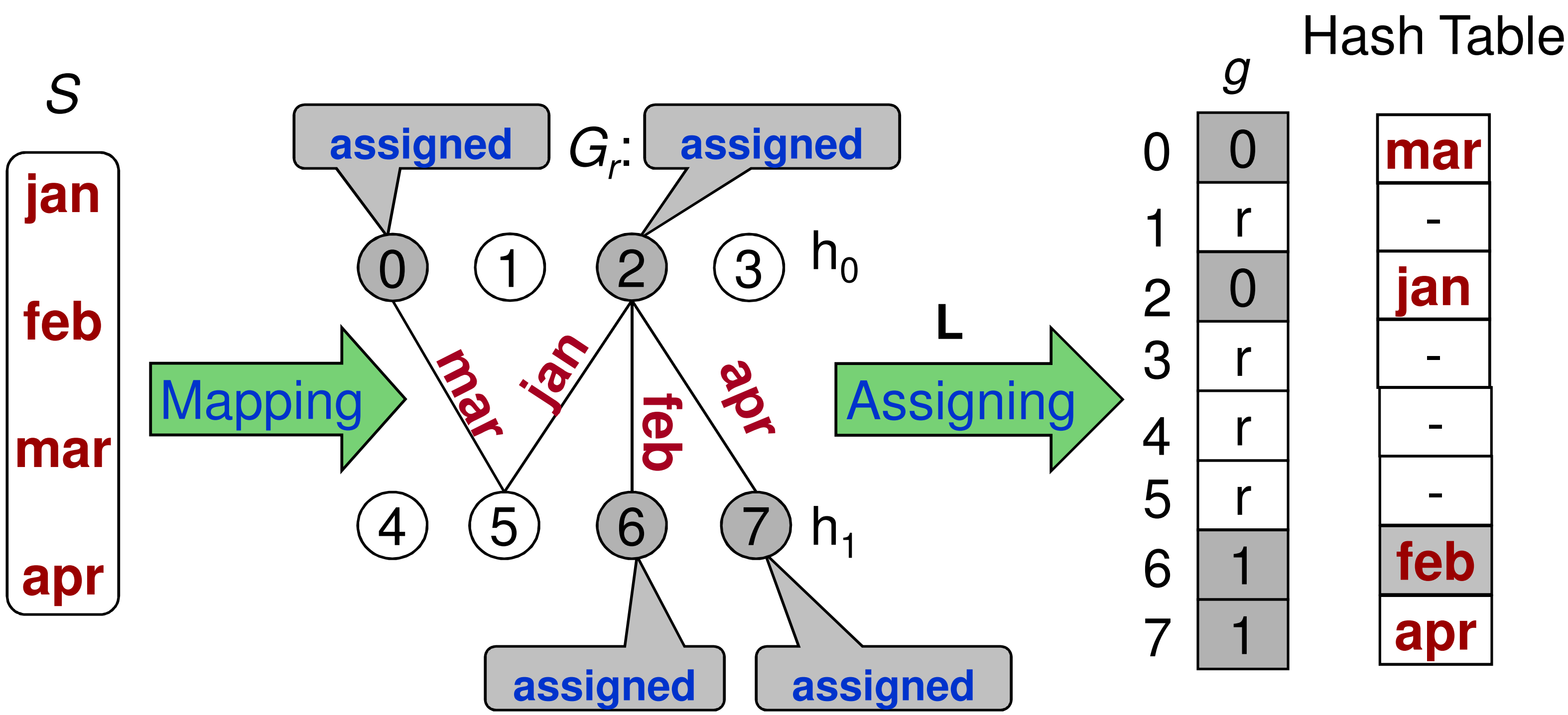


Internal Random Access Memory Algorithm (r=2)



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

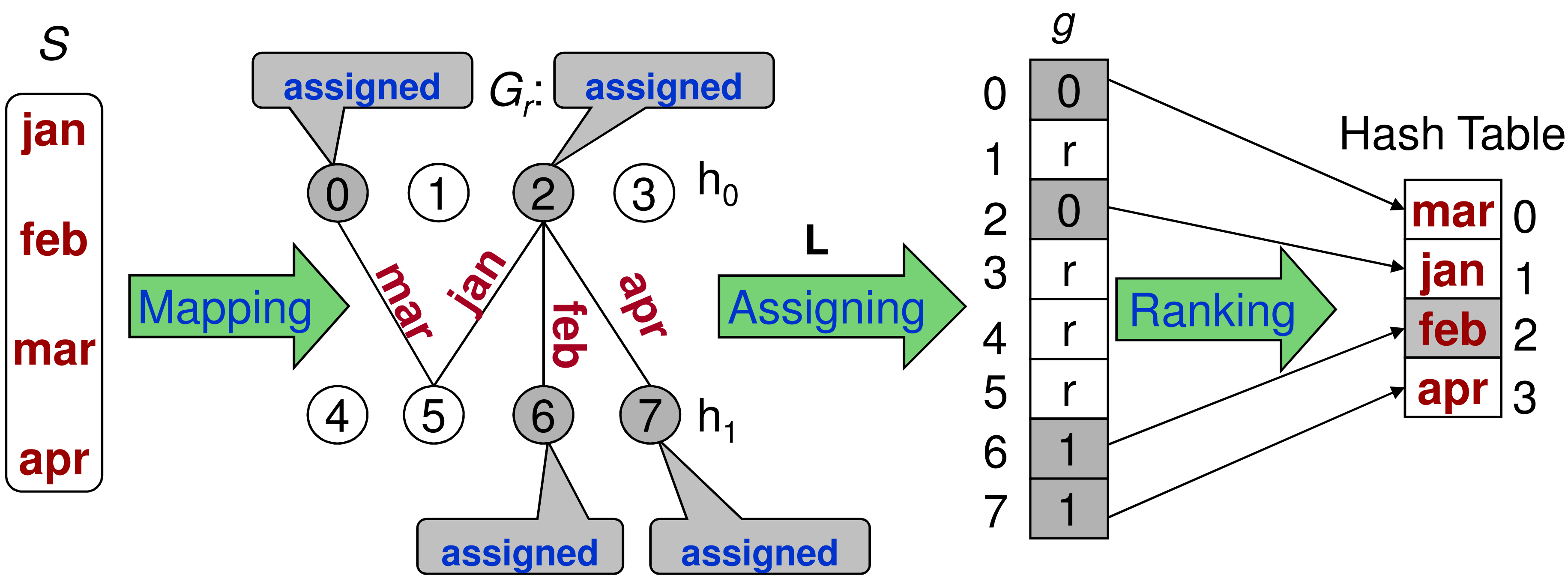
Internal Random Access Memory Algorithm: PHF



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

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

Internal Random Access Memory Algorithm: MPHF

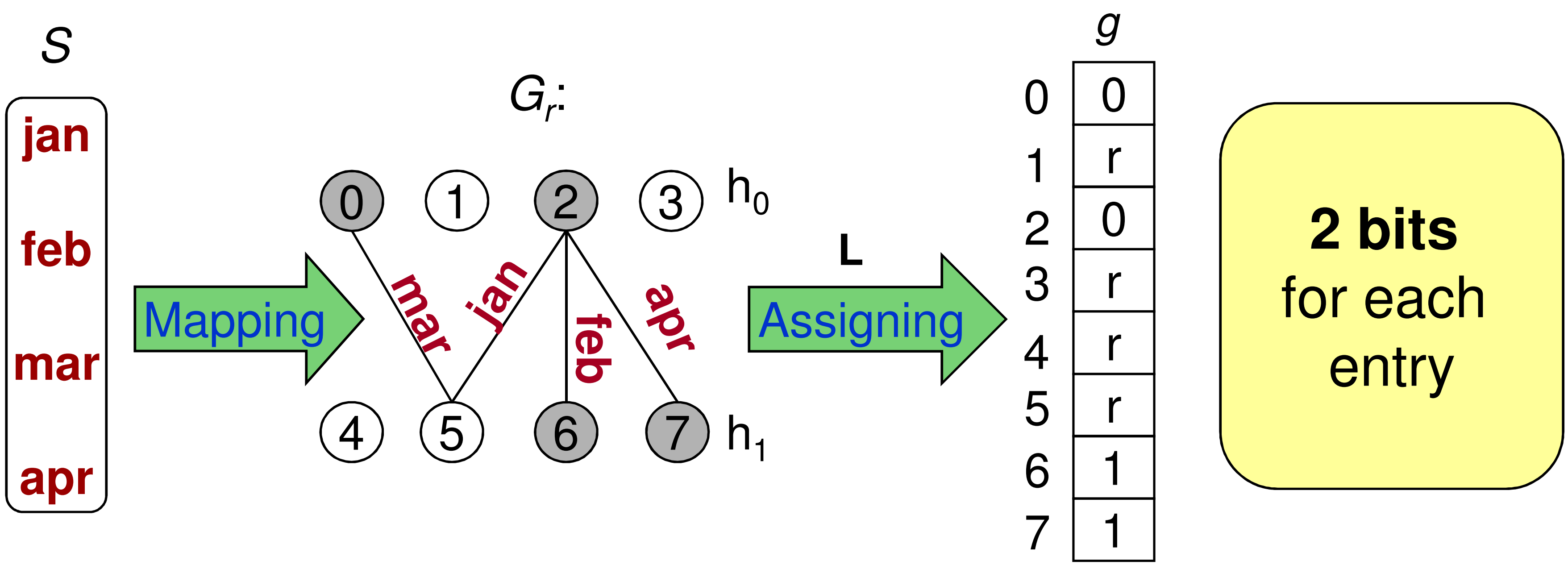


$i = (g[h_0(\text{feb})] + g[h_1(\text{feb})]) \bmod \mathbf{r} = (g[2] + g[6]) \bmod \mathbf{2} = 1$

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

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

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 $r = 3$, $m = 1.23n$: Pr_a tends to 1
- Number of iterations is $1/Pr_a = 1$

Experimental Results

- Metrics:
  - Generation time
  - Storage space for the description
  - 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)

n (millions)	32	128	512	1024
Sequential ECA	0.95 ± 0.02	5.1 ± 0.01	22.0 ± 0.13	46.2 ± 0.06

Related Algorithms

- Fox, Chen and Heath (1992) – FCH
- Majewski, Wormald, Havas and Czech (1996) – MWHC

All algorithms coded in the same framework

Generation Time

Algorithms	Generation Time (sec)
Internal (r = 3)	6.7 ± 0.01
External	6.3 ± 0.11
MWHC	7.18 ± 0.01
FCH	2,400.1 ± 711.6

3,541,615 URLs

Generation Time and Storage Space

Algorithms	Generation Time (sec)	Space (bits/key)
Internal (r = 3)	6.7 ± 0.01	2.6
External	6.3 ± 0.11	3.1
MWHC	7.18 ± 0.01	26.76
FCH	2,400.1 ± 711.6	4.2

3,541,615 URLs

Generation Time, Storage Space and Evaluation Time

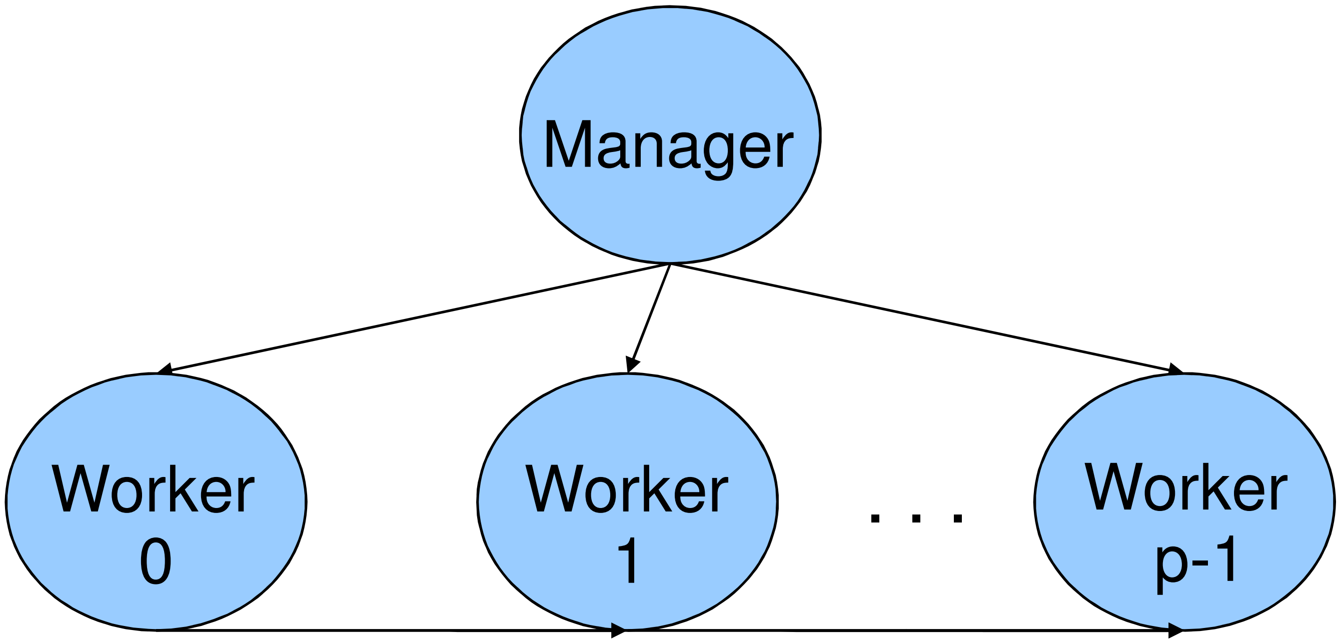
Algorithms	Generation Time (sec)	Space (bits/key)	Evaluation time (sec)
Internal (r = 3)	6.7 ± 0.01	2.6	2.1
External	6.3 ± 0.11	3.1	2.7
MWHC	7.18 ± 0.01	26.76	2.46
FCH	2,400.1 ± 711.6	4.2	1.6

3,541,615 URLs

Key length = 64 bytes

The Distributed External Memory Based Algorithm ...

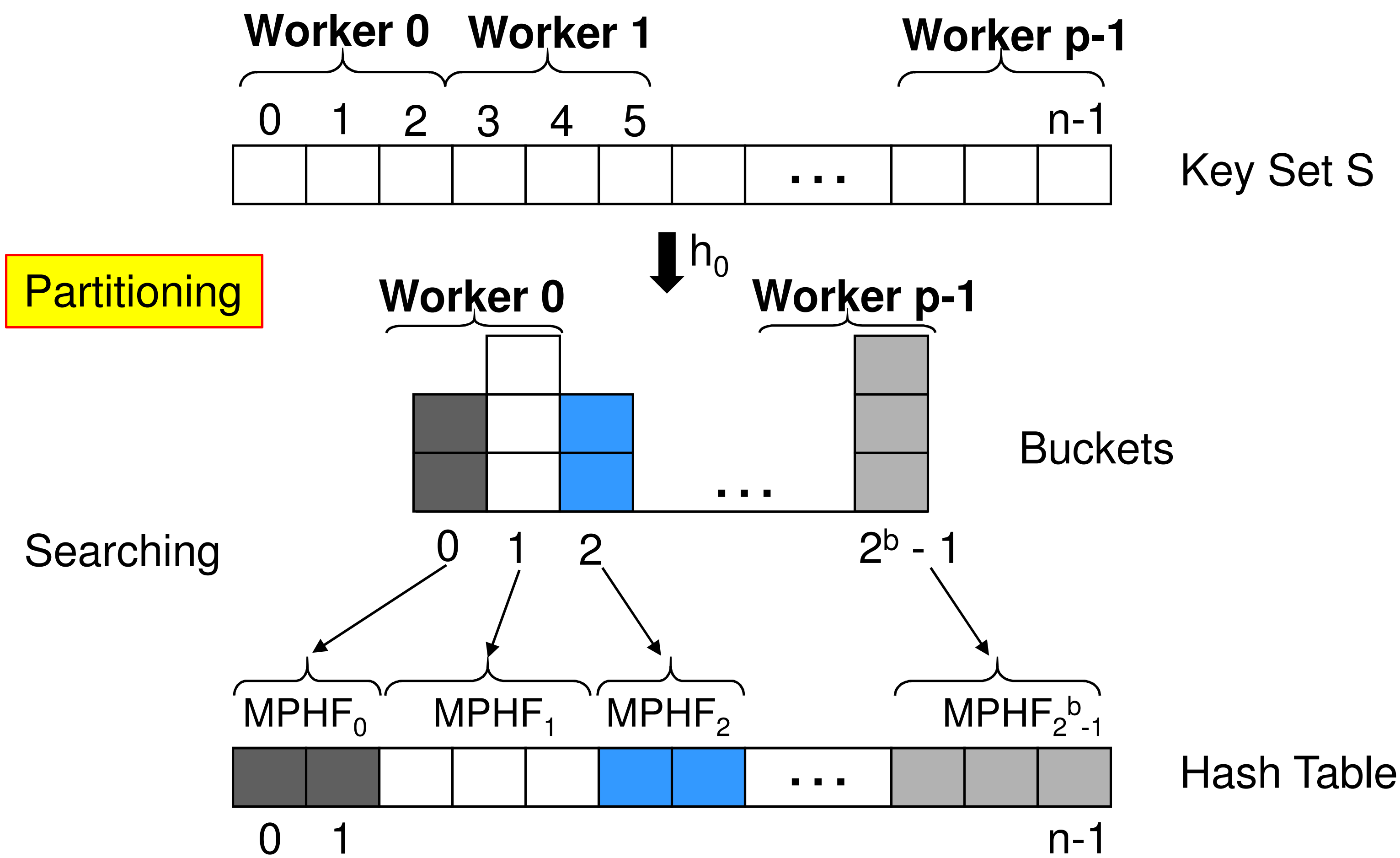
Distributed Construction of the MPHFs



Distributed Construction of MPHFs

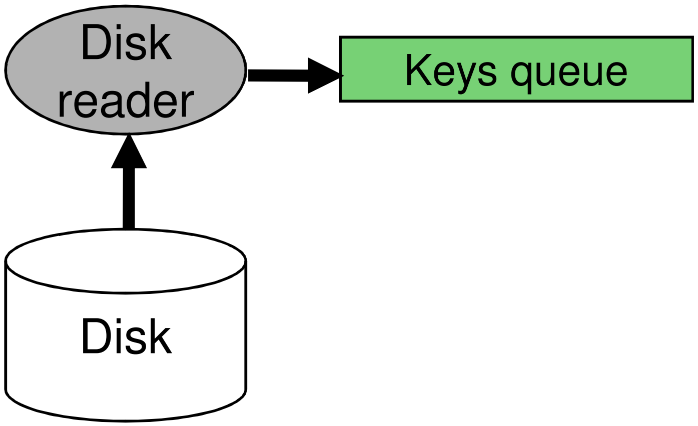
- Manager :
  - Assign tasks to workers
  - Determine global values during execution
  - Dump resulting MPHFs to disk
- Worker :
  - Has a partition of the keys on disk
  - Creates its buckets from the keys ($B_{pw} = N_b/p$)
    - Migrate data whenever necessary
  - Constructs a MPHF for each bucket

Sequential External Perfect Hashing Algorithm

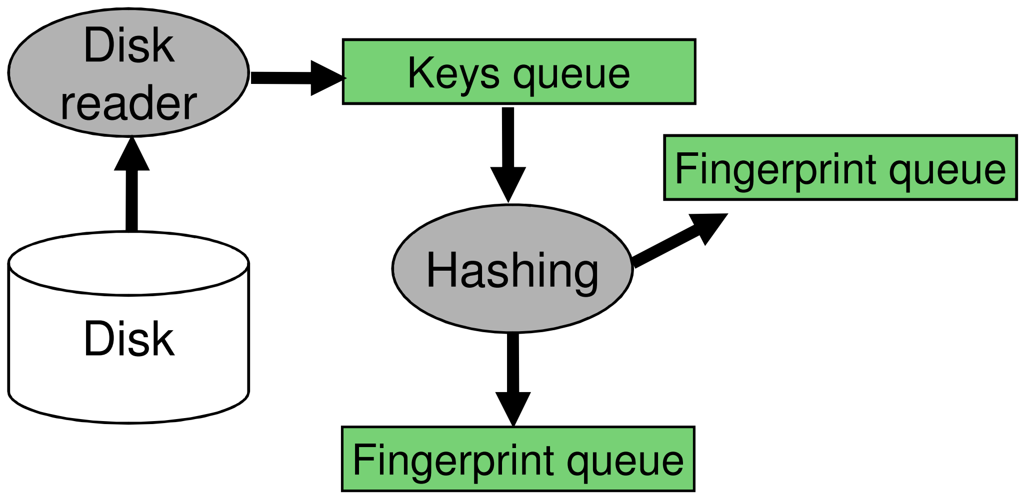


$$MPHF(x) = MPHF_i(x) + offset[i];$$

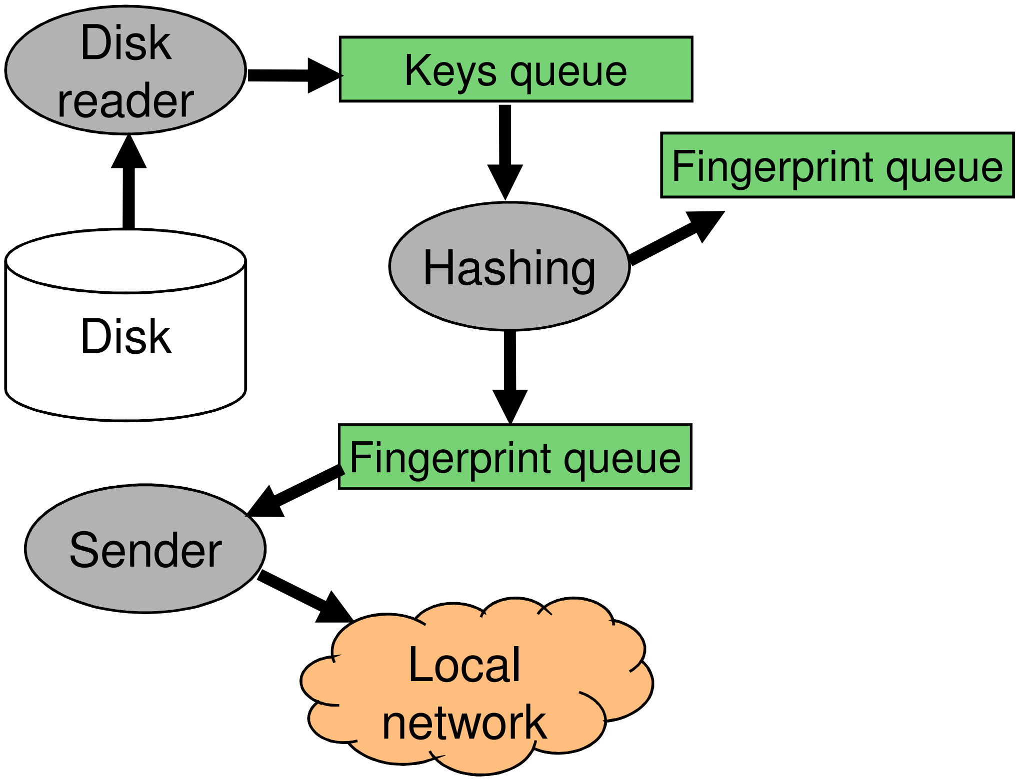
Partitioning Step in Each Worker



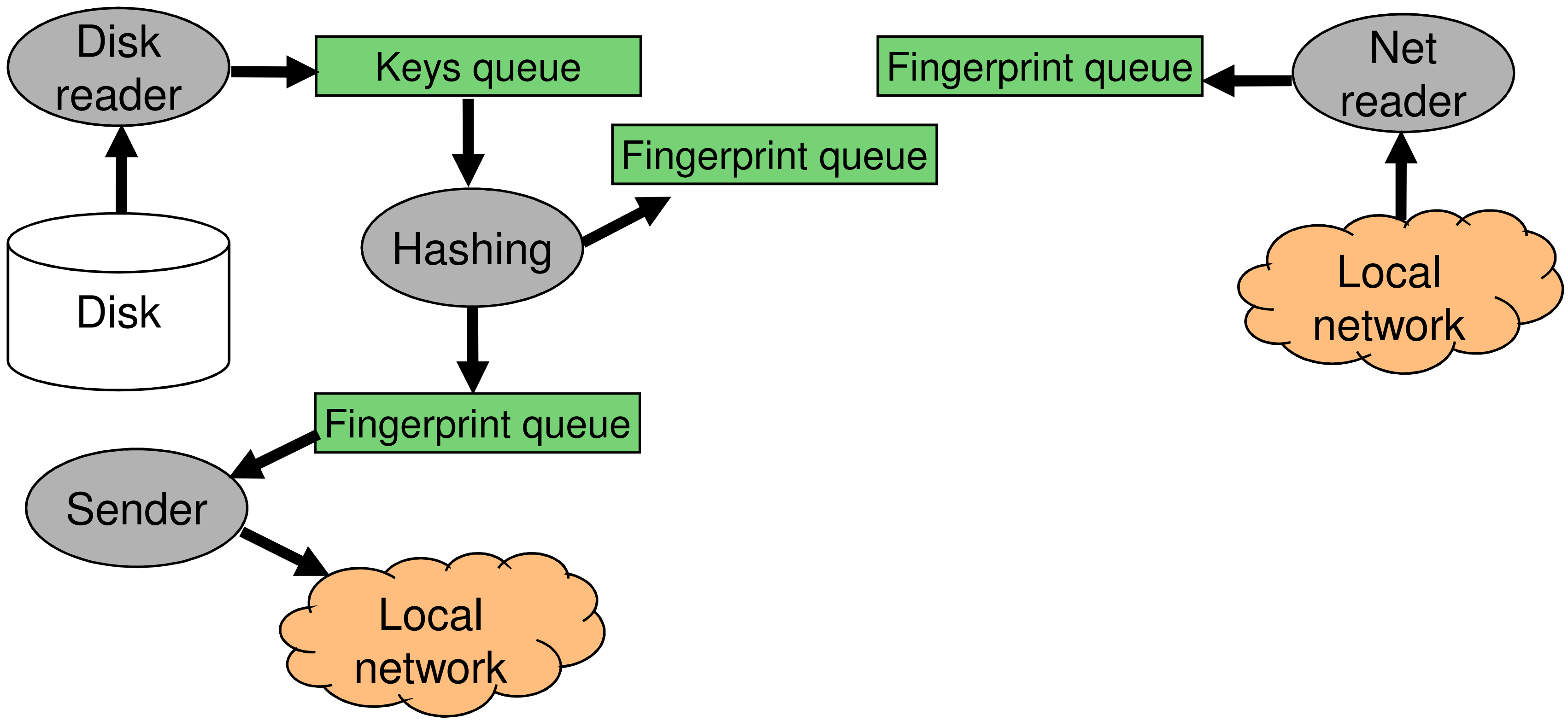
Partitioning Step in Each Worker



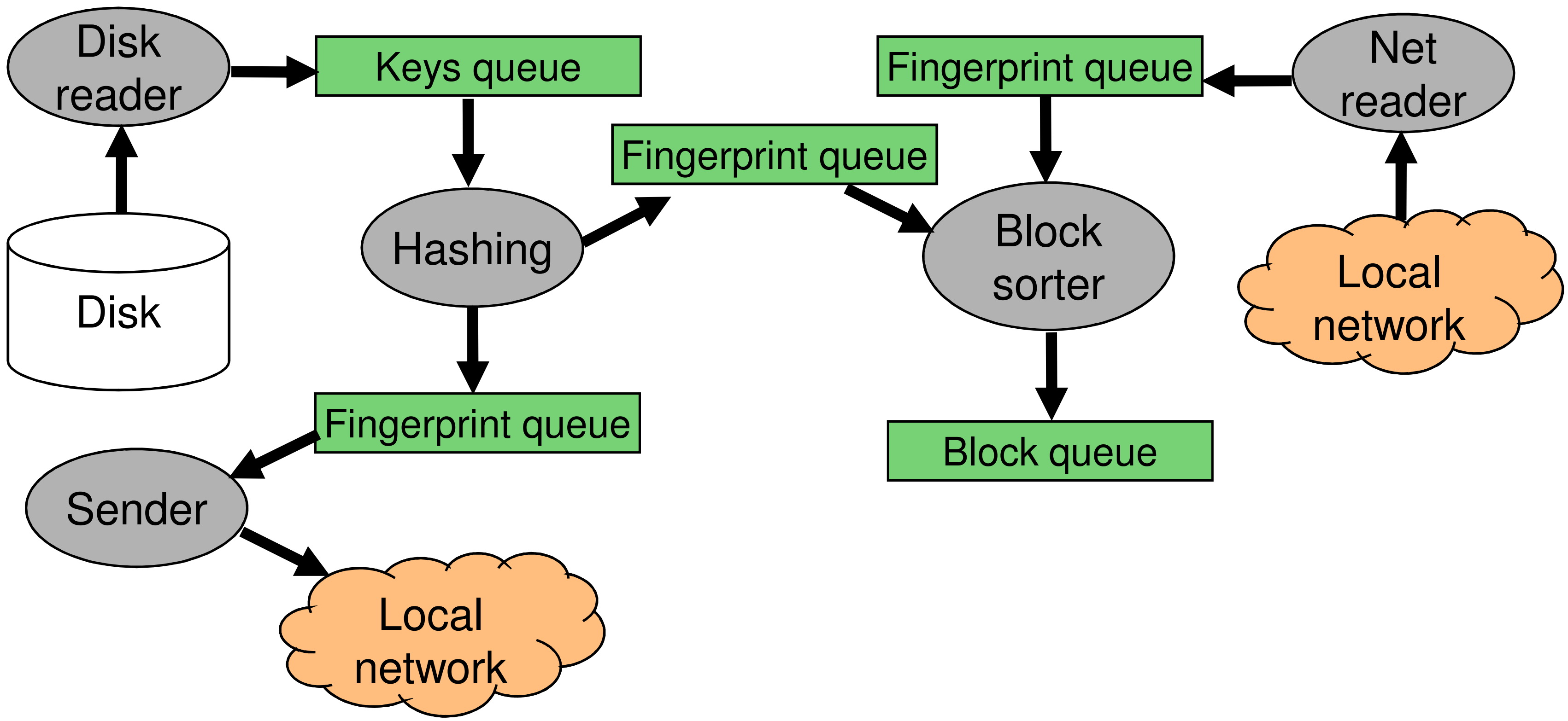
Partitioning Step in Each Worker



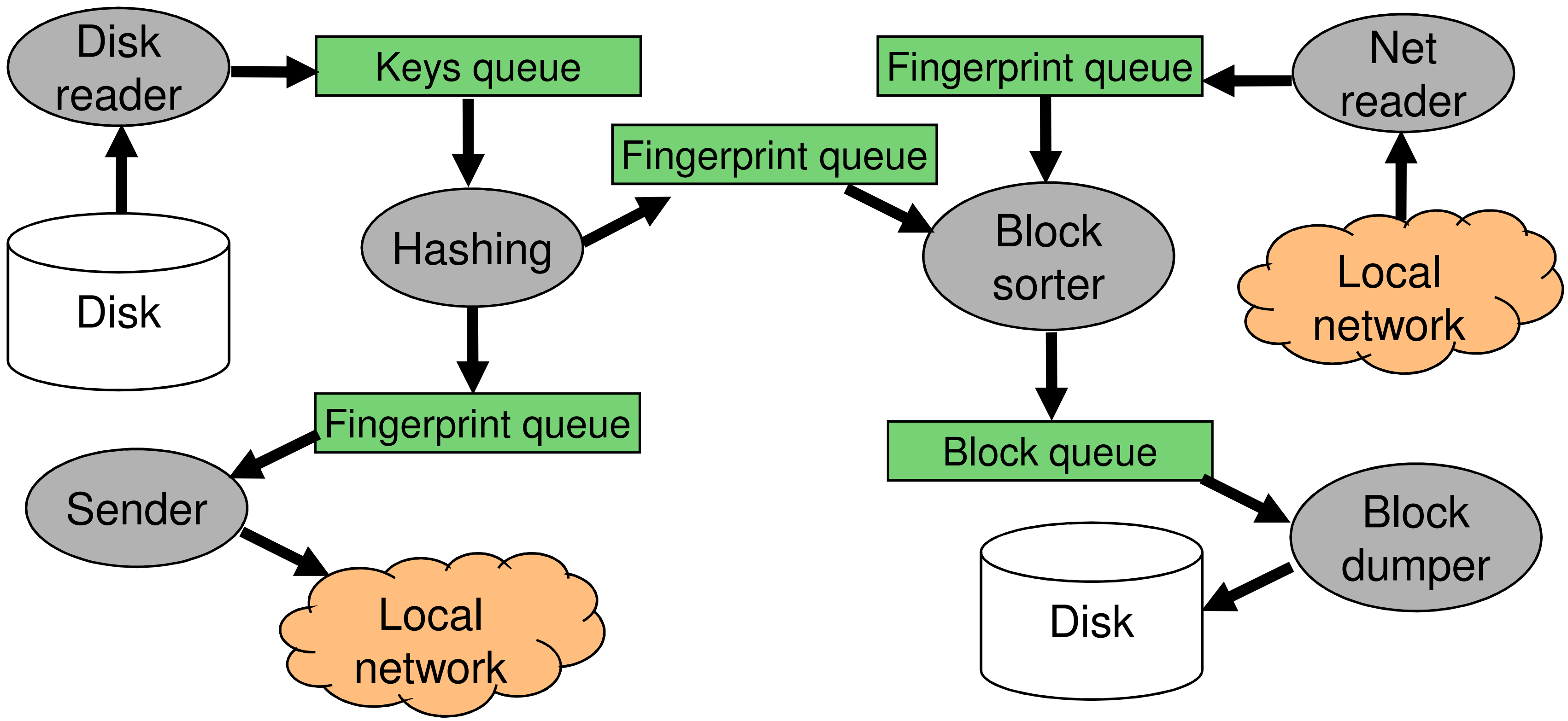
Partitioning Step in Each Worker



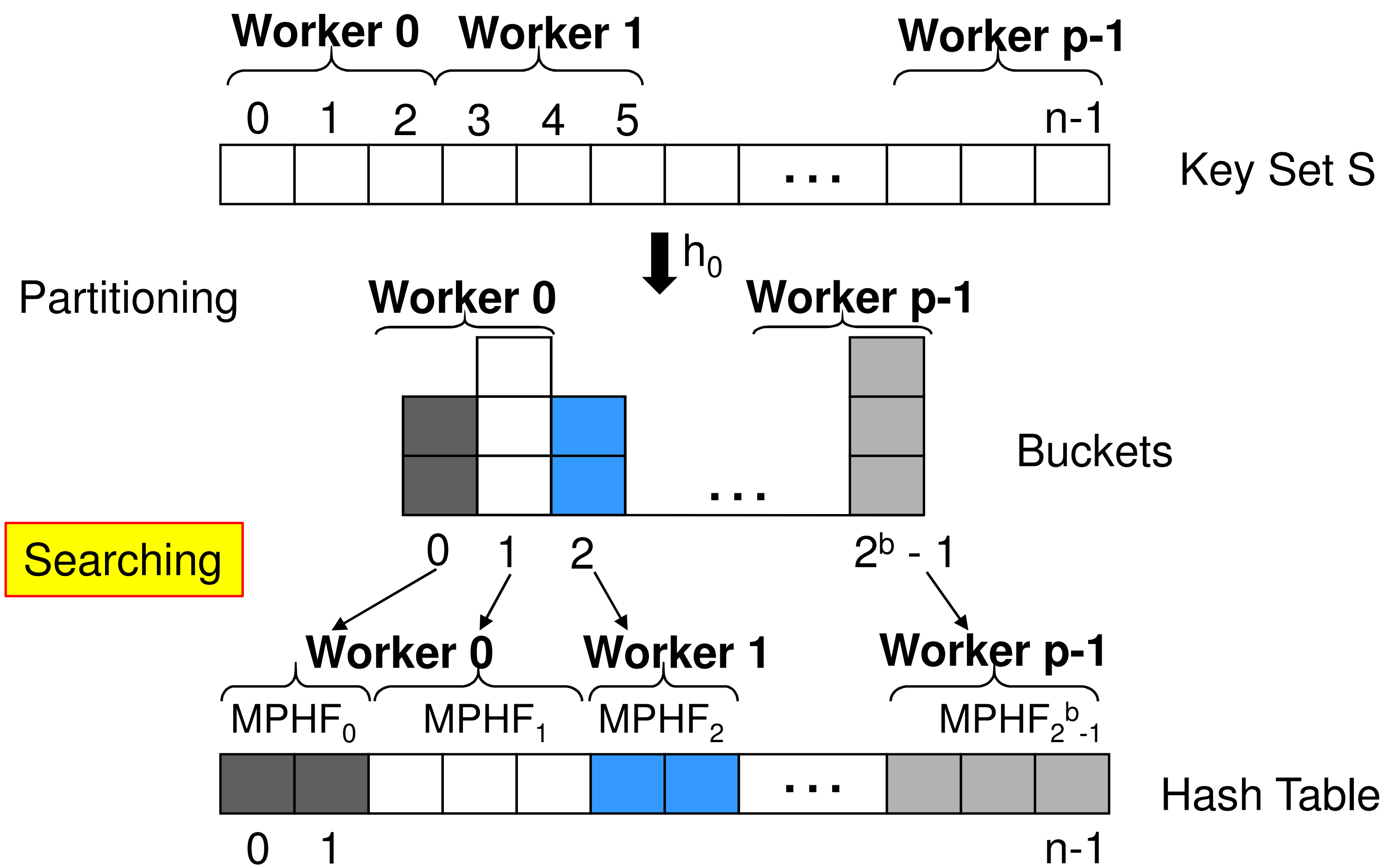
Partitioning Step in Each Worker



Partitioning Step in Each Worker

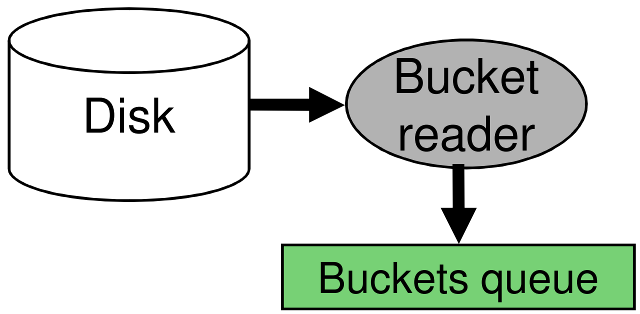


Sequential External Perfect Hashing Algorithm

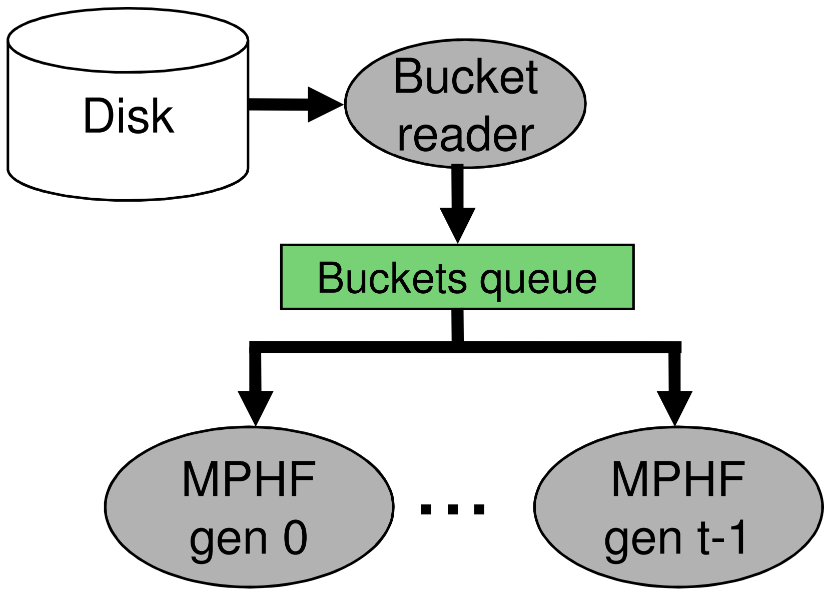


$MPHF(x) = MPHF_i(x) + offset[i];$

Searching Step in Each Worker



Searching Step in Each Worker



Description and Evaluation of a MPHF

- *Centralized* in one machine
- *Distributed* among the participating machines

Centralized Description and Evaluation of MPHFs

- End of the partitioning step:
  - Worker sends size of each bucket to manager
  - Manager calculates the *offset* array
- End of searching step (construction of MPHFs):
  - Worker sends *MPHFs* of its buckets to manager
  - Manager writes sequentially final *MPHF* to disk
- $MPHF(x) = MPHF_i(x) + offset[i]$

Distributed Description and Evaluation of MPHFs

- Description of MPHFs of a bucket
  - Stays in the bucket
- Evaluation of a MPHF
  - Locate the key inside the bucket:
    $MPHF_{partial}(k) = MPHF_i(k) + localoffset[i]$
  - Add this to the number of keys before worker *w:*
    $MPHF(k) = MPHF_{partial}(k) + globaloffset[w]$
  - A key stream is evaluated in parallel

Advantages of Distributed Evaluation of MPHFs

- No need to send the MPHFs of a bucket to manager
- They are written to disk in parallel by the workers
- Final function is stored in a distributed way
  - Size of the description of the MPHF grows linearly with the size of the input key

Communication Overhead

- On average, the number of keys τ sent through the net during the execution is:

$$\tau = \frac{n(p-1)}{p}$$

p	Keys sent by a worker to the net		
	Max (%)	Min (%)	τ (%)
4	75.008	74.994	75.000
10	90.009	89.991	90.000
14	92.864	92.849	92.857

Experimental Setup

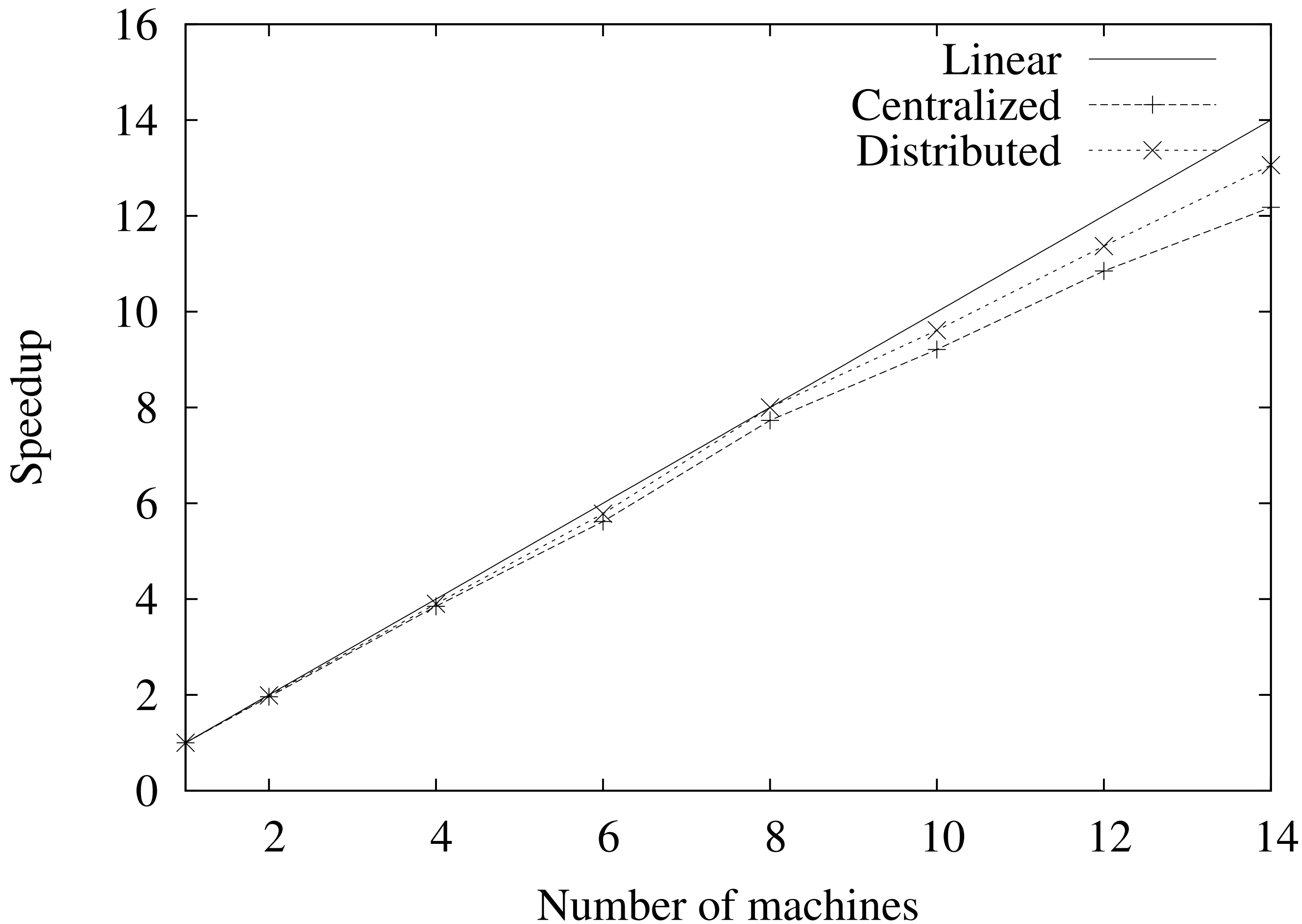
- Three collections

Collection	Avg. Key Size	*n* (billions)
URLs	64	1.024
Random Integers	16	1.024
	8	1.024

- Cluster with 14 equal 64 bits single core machines
  - 2 gigabytes of main memory
  - 2.13 gigahertz
  - Linux operating system version 2.6

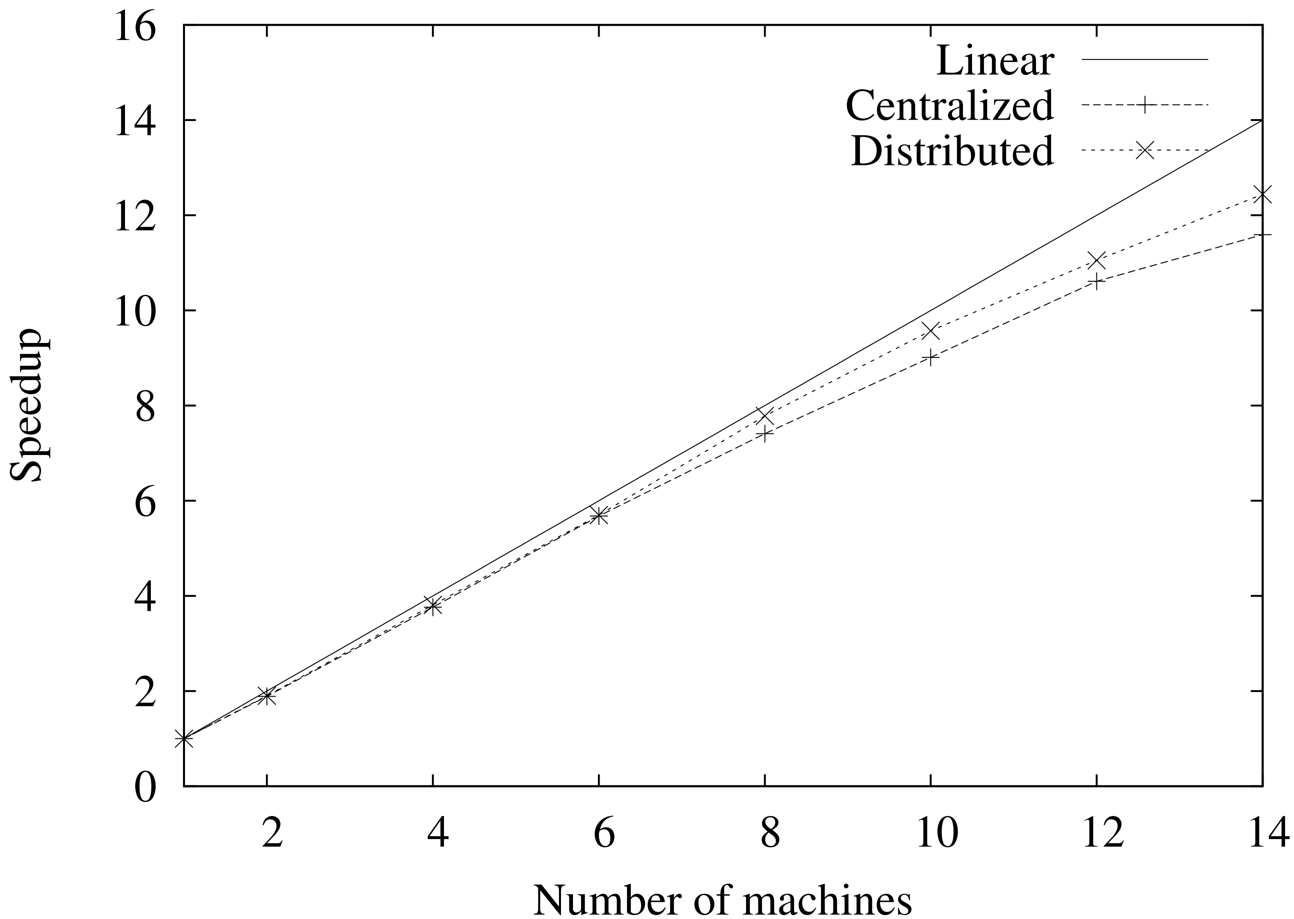
Speedup

- 64 bytes URLs



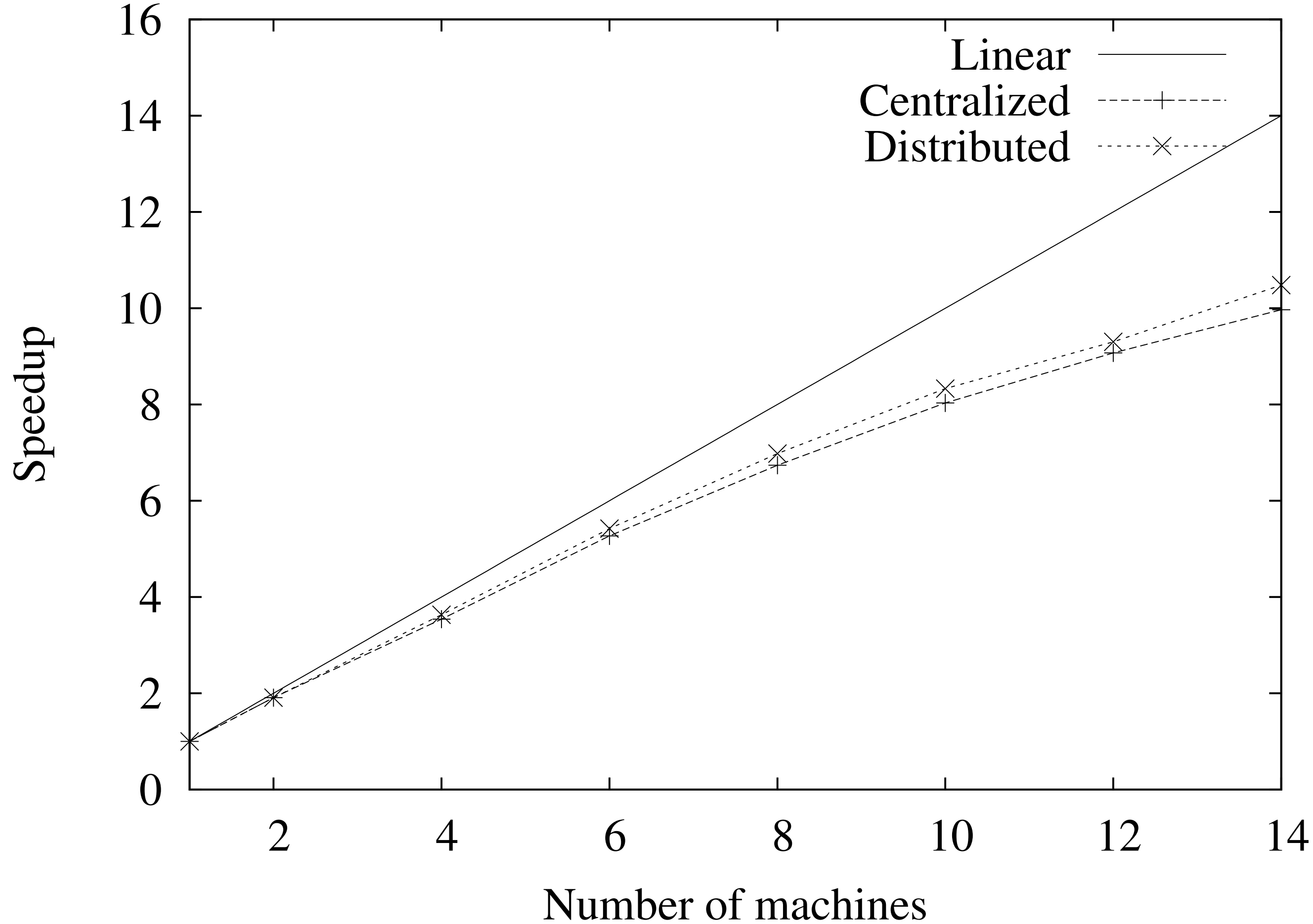
Speedup

- 16 bytes random integers



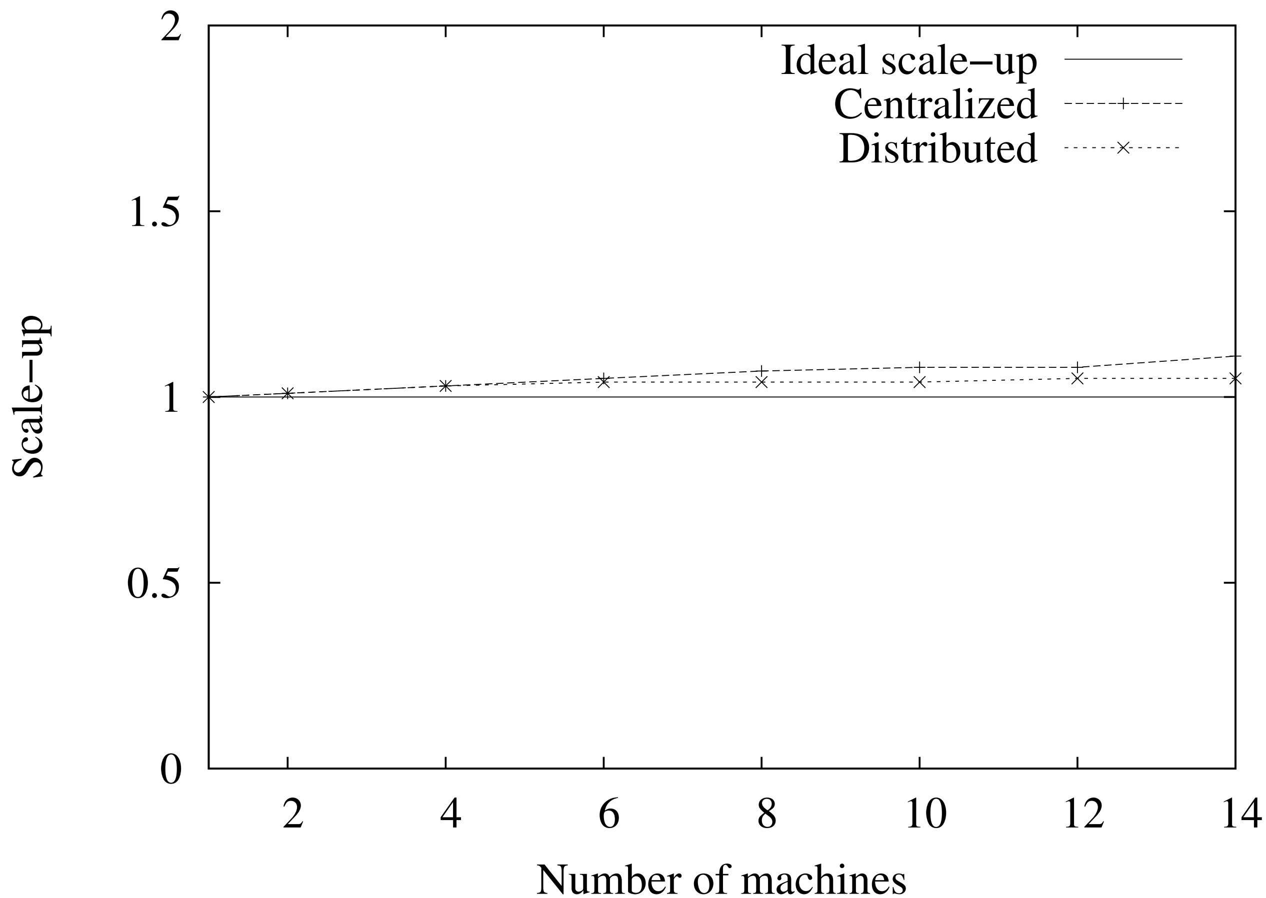
Speedup

- 8 bytes random integers



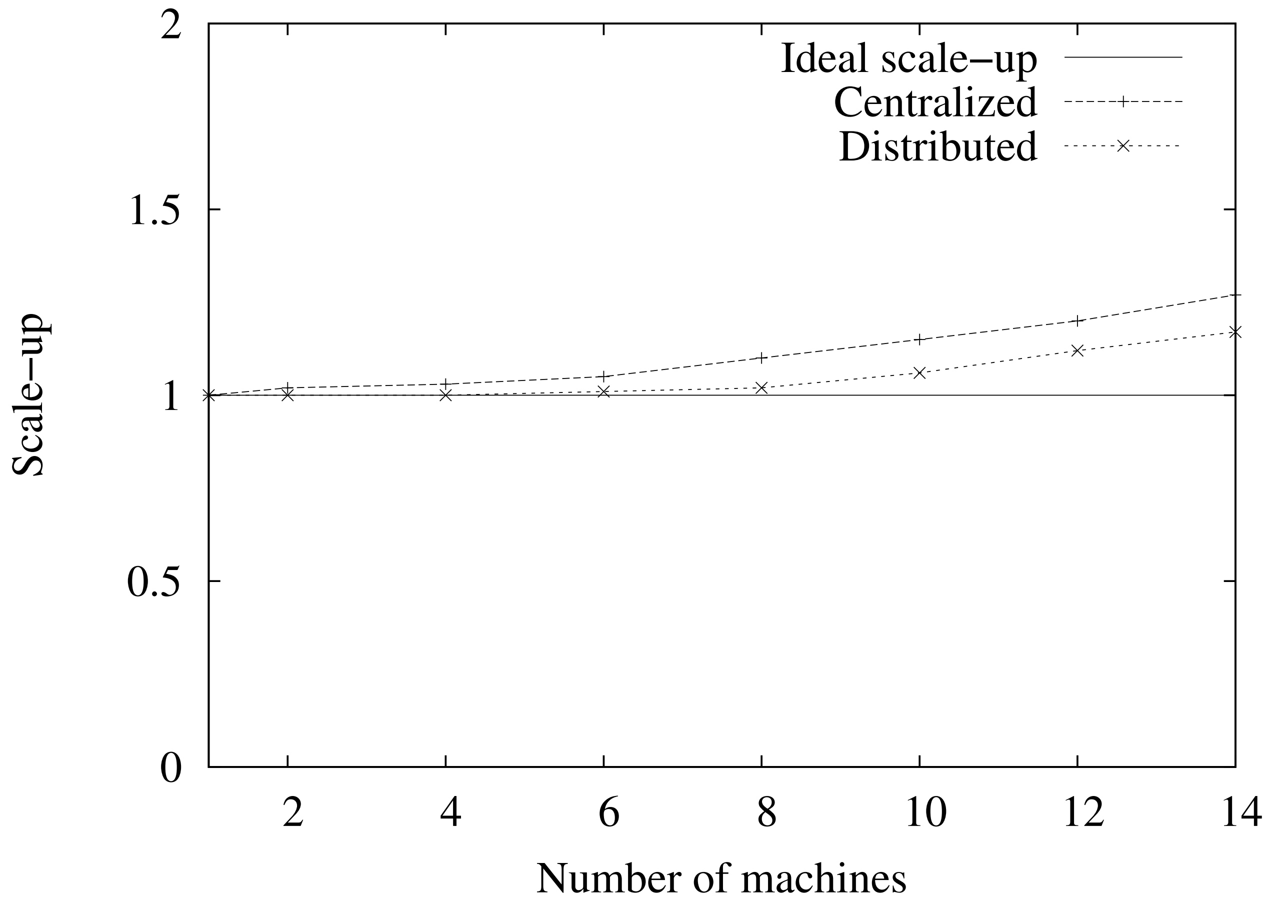
Scale-up

- 64 bytes URLs



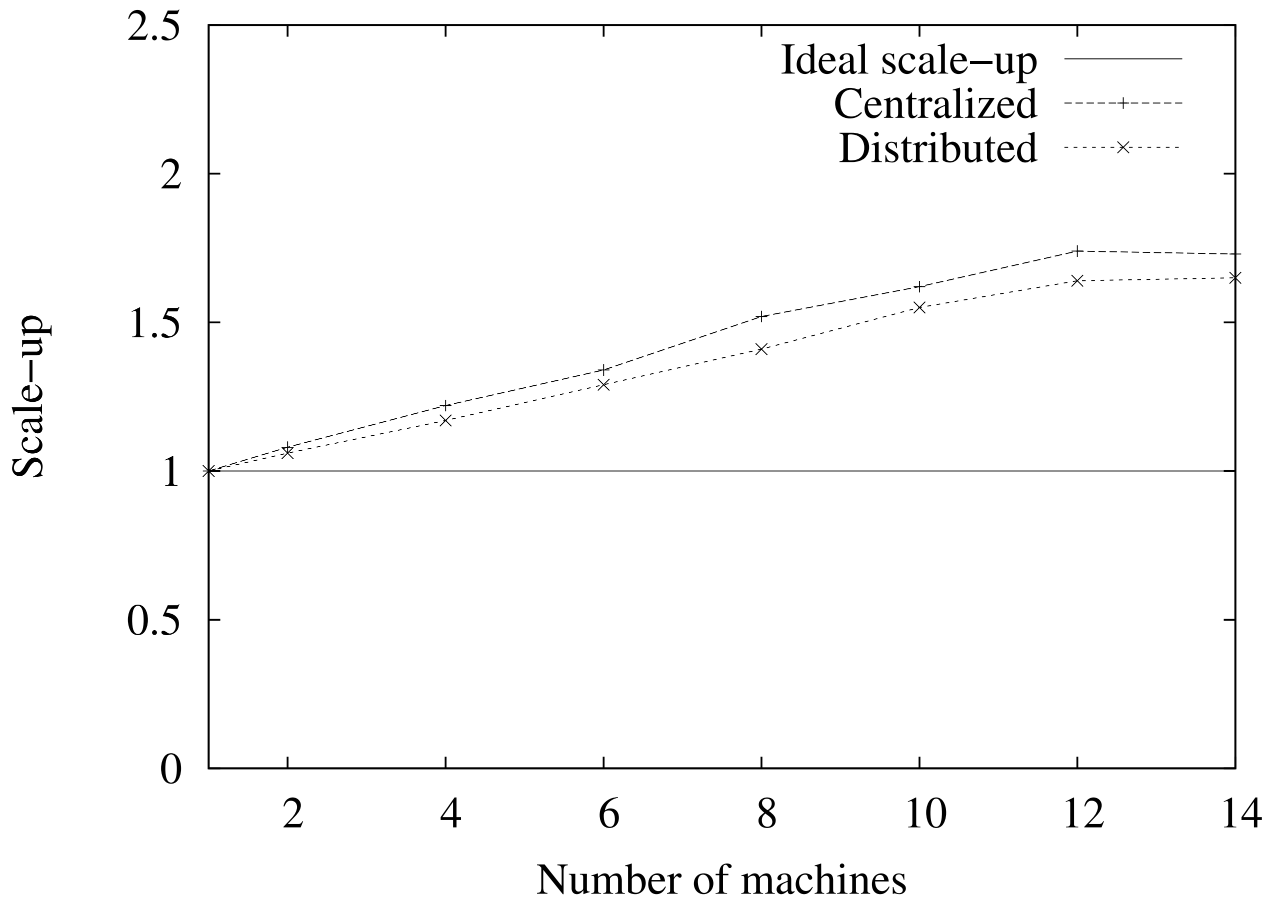
Scale-up

- 16 bytes random integers



Scale-up

- 8 bytes random integers



Scale-up

- 14.336 billion random integers
- 14 machines (each with 1.024 billion keys)

n (billions)	Random Integer Collections	Construction time		
		Seq.	Distrib.	U_p
14.336	16-byte	41.17	49.5	1.20
	8-byte	34.58	58.00	1.68

Load Balancing

- Execution time: fastest minus slowest (in minutes)

p	t_{fw}	t_{sw}	$t_{sw} - t_{fw}$
4	12.78	12.86	0.09
10	4.32	4.40	0.07
14	3.76	3.84	0.08

Distributed Evaluation

- 1.024 billion key stream taken at random (minutes)

Collection	Evaluation time (min)	
	Centr. Eval.	Distrib. Eval.
64-byte URLs	33.1	21.7
16-byte Integers	24.5	11.5
8-byte Integers	18.2	10.1

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:
  - 2,243 downloads (until May 27th, 2008)
  - Incorporated by Debian
- Library address: http: //cmph.sourceforge.net

Conclusions

- Sequential and parallel perfect hashing algorithm
- Near space-optimal functions in linear time
- Function evaluation in time O(1)
- The algorithms are simpler and has much lower constant factors than existing theoretical results
- Outperforms the main practical general purpose algorithms found in the literature

Conclusions

- Construction time: 14 machines, 1 billion URLs
  - Sequential algorithm: 50 minutes
  - Parallel algorithm: 4 minutes
- Speedup > 90% for keys with more than 16 bytes
- Description and evaluation of MPHF:
  - Centralized
  - Distributed: fast evaluation for key streams