Yannis Kotidis

What is hashing?

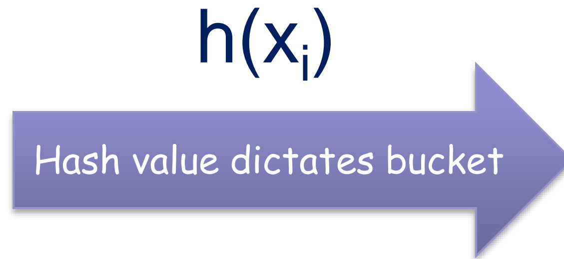
- Hashing generally takes records whose key values come from a large range and maps those records in a “hash table” with a relatively smaller number of slots called buckets
- Collisions occur when two records with different keys hash to the same bucket

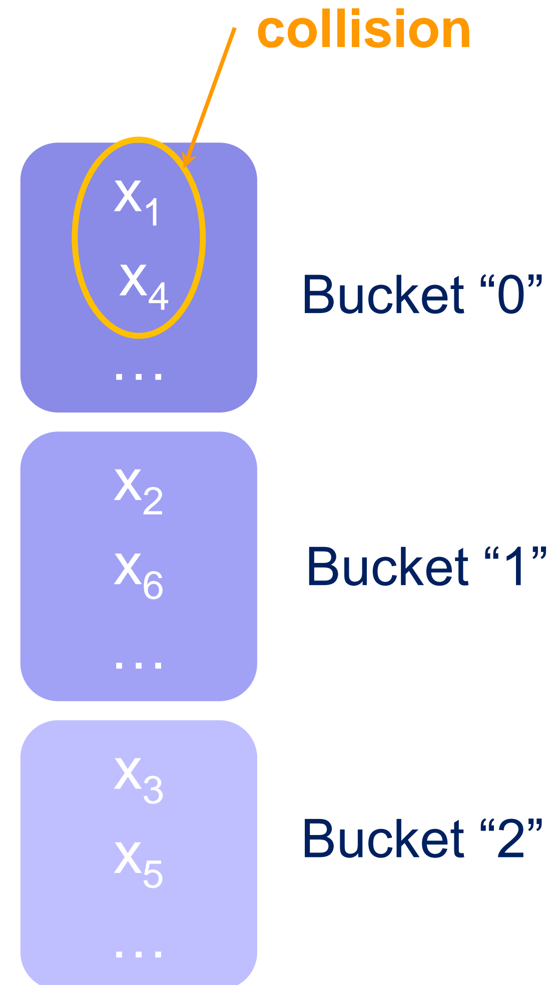
Hash function h()

- Maps arbitrary items (keys) into integers
  - If the result of the hash function is an k-bit integer, then there are 2^k possible outcomes (buckets/slots)
    - For instance using 32-bit arithmetic results in 2^{32} = 4.294.967.296 slots
  - We often limit the number of slots by using modulo arithmetic: slot = h() % N
    - %N returns values in range [0..N-1], e.g. 19 % 7 = 5
- (For most applications) a good has function should
  - be easy (fast) to compute
  - provide a uniform distribution across the hash table and should not result in clustering of keys (unless this is desirable)
  - avoid collisions (to the extend possible)

Example: $h(x)=(2x+1) \% 3$

$x_1=1$
$x_2=3$
$x_3=2$
$x_4=7$
$x_5=5$
$x_6=9$



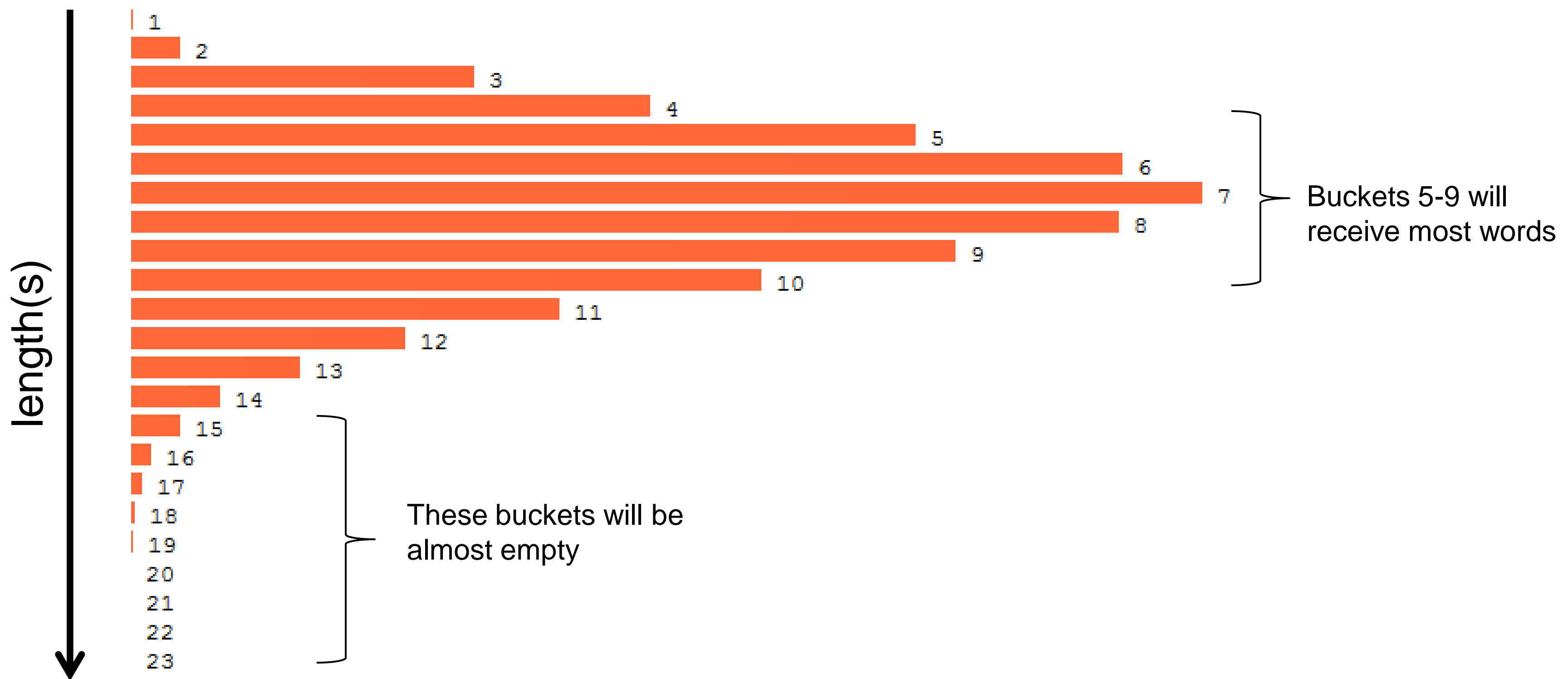


Careful

- Often key values exhibit skew
  - *Age* of my customers used as key. But most of my customers are young
- Prefer hash functions that distribute records uniformly among the buckets
- Example: want to hash strings extracted from a document

Assume h(s) = length of string s

- English Word Length Distribution:



A better function (input string s)

- s[i] = i^{th} character in string
- K=some (prime) constant
- Recursive computation
  - h(1) = s[1] /*** first character in string ***/
  - h(i) = h(i-1)*K + s[i] , for i>1
- **Return h[length(s)]**

Example for s='abc'

- $h(1)=a$
- $h(2)=h(1)K+b=aK+b$
- $h(3)=h(2)K+c=aK^2+bK+c$
- Thus:

$$h(\text{'abc'})=aK^2+bK+c$$

Example continued (assume K=31)

- Ascii codes of ‘a’,’b’ and ’c’ are 97, 98 and 99, respectively
- h(‘abc’) = (a*K+b)*K+c = aK^2+bK+c $=97*31^2+98*31+99=96354$

- Another example: h(‘acb’) = … = 96384

Note

- Previous function may return arbitrarily large numbers
  - h('supercalifragilisticexpialidocious')= 389236099458587451617003512335442884432133029560316182532768950479139510450238495 5 (for K=257)
- Quite often you want to restrict the range of buckets in an implementation
  - For instance if a bucket maps to a physical entity like a page in main memory or disk
  - Assume you want to create N=1024 buckets. How to modify the hashing function?

Universal Hashing

- Informally: derive a family of hash functions H with low probability of collisions
- Assume keys (data) are drawn from a universe U and there are m slots in the hash table.
- For every hash function h∈H, the following property should hold:

$$\forall x, y \in U,\ x \neq y: \quad \Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}$$

Universal Hashing Example

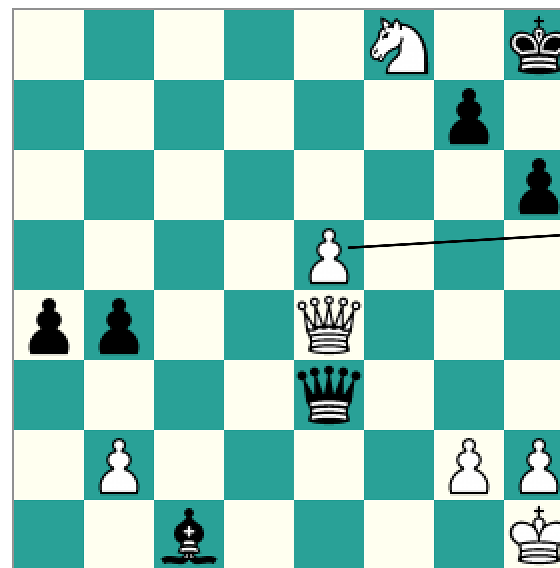
- Assume a, b are randomly chosen integers and $a \neq 0$
- Given a prime number p, with $p \geq m$
- Then, the following family of hash functions is universal:

$$h_{a,b}(x) = ((ax+b) \% p) \% m$$

- Note: commonly used families of hash function use bit-arithmetic instead of modulo operations for efficiency

Hashing as an index for Chess Games

- Zorbist hashing:
  - Generate an array of 781 64 bit random numbers
  - One number for each piece at a position (2*6*64 total)
    - 6 pieces: king, queen, rooks, bishops, knights, pawns
    - 8*8 positions, 2 colors
  - 13 additional numbers encoding side to move, castling rights, etc
  - A position is hashed to a bucket by XORing appropriate random numbers
    - Need 64bits to describe a board
    - Very small rate of collisions



$$X = x_1 \text{ XOR } x_2 \text{ XOR } \dots x_{14}$$

Use this single value to encode the position

More on x XOR y

- Result is 1 if input bits differ, 0 otherwise
- 0101 XOR 0110 = 0011

  (brace labels: 0 over the first bits "0" and "0"; 1 under the third bits "0" and "1")
- 0011 XOR 0110 = 0101

and

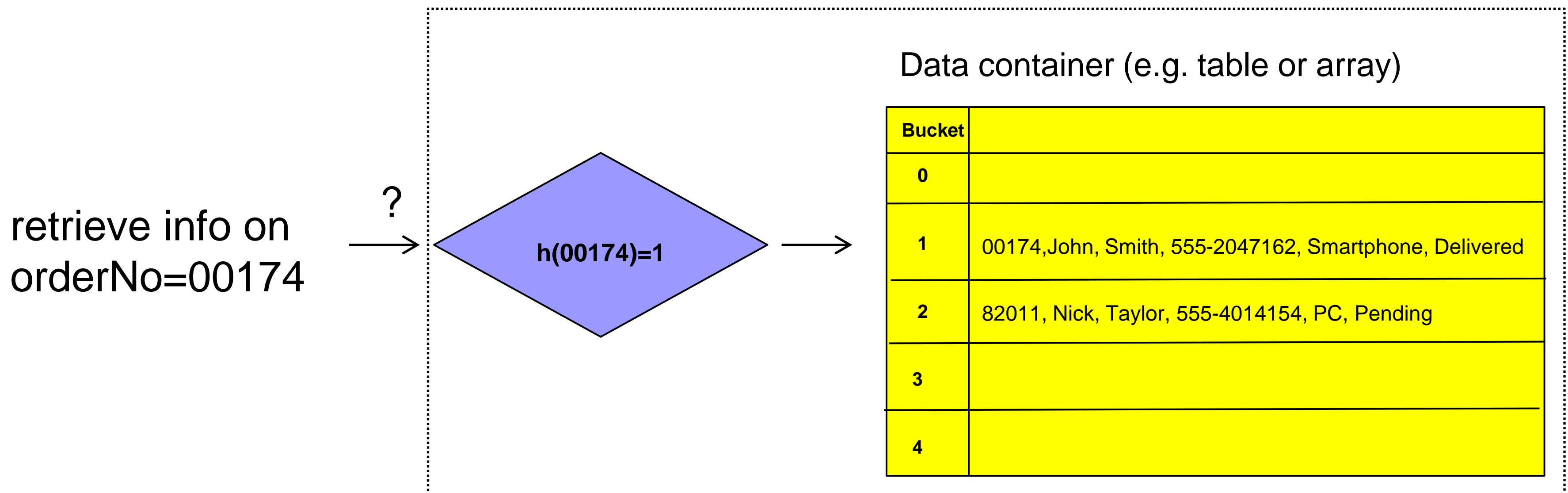
- 0011 XOR 0101 = 0110

Closed vs Open hashing

- Close hashing: objects are stored directly within the hash table
- Open hashing: we may “leave” the hash table (for instance by traversing a data structure like a linked list) in order to locate the placement of an object

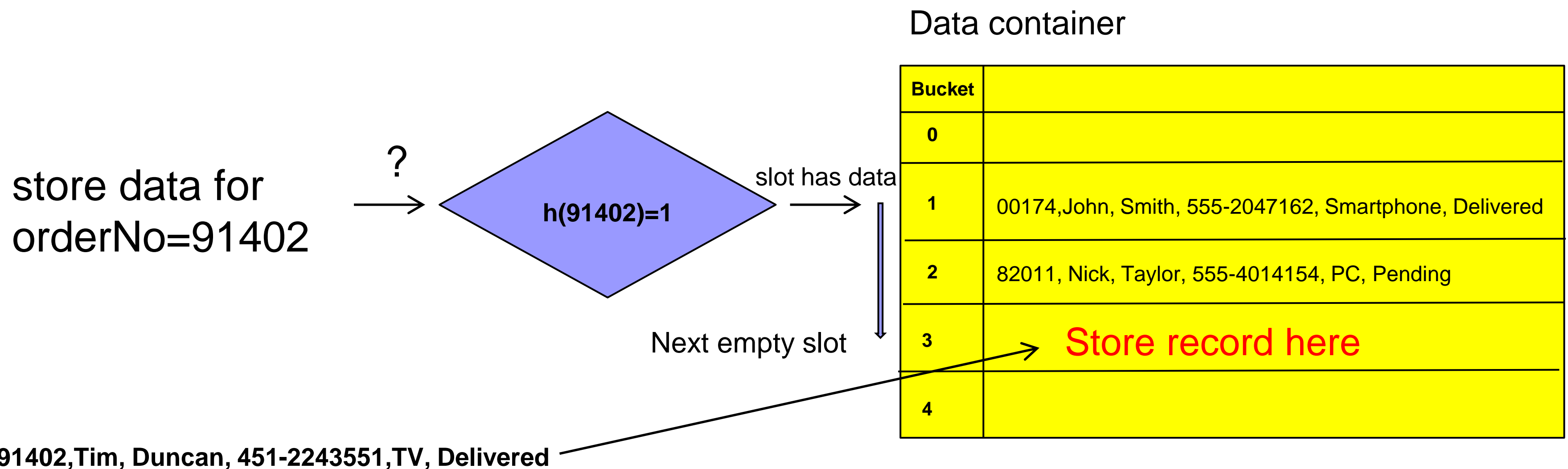
Closed hashing example

- Organize your data to quickly locate records based on attribute's x value
  - Data may be stored in memory or on disk



Handling *collisions*

- Another key hashes to the same position
  - Linear probing: scan for next available slot
  - How to search this table?



Deletions are complicated

- Assume that orderNo 00174 is deleted
- How to update the hash-table?

Data container

Bucket	
0	
1	00174,John, Smith, 555-2047162, Smartphone, Delivered
2	82011, Nick, Taylor, 555-4014154, PC, Pending
3	91402,Tim, Duncan, 451-2243551,TV, Delivered
4	

One of these (or both) will have to be moved up (why)?

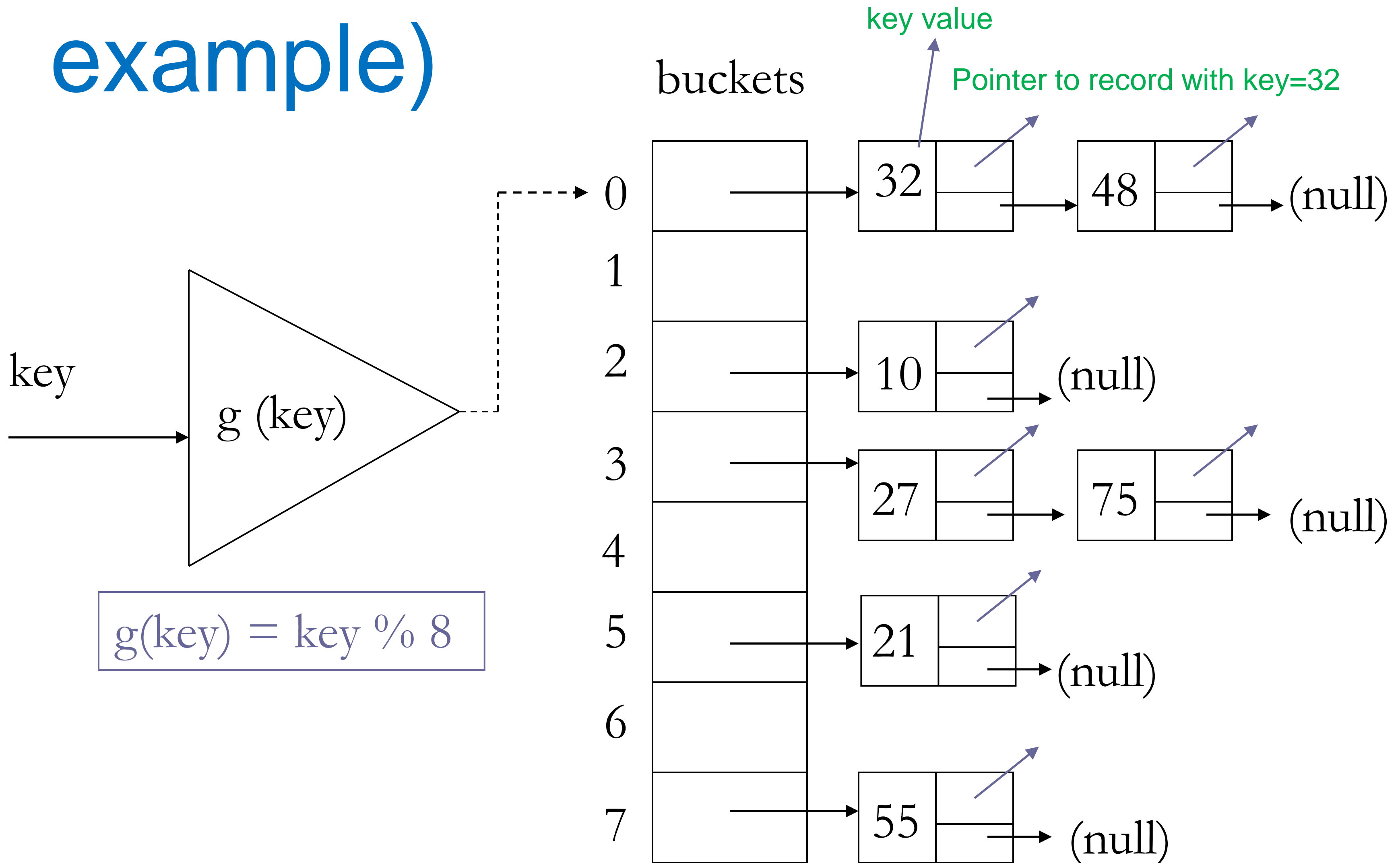
Better (w.r.t maintenance) strategy

- Assume that orderNo 00174 is deleted
  - Mark corresponding slot as "available" using a special marker
  - Periodically perform a clean up
    - remove available markers and reinsert items

Data container

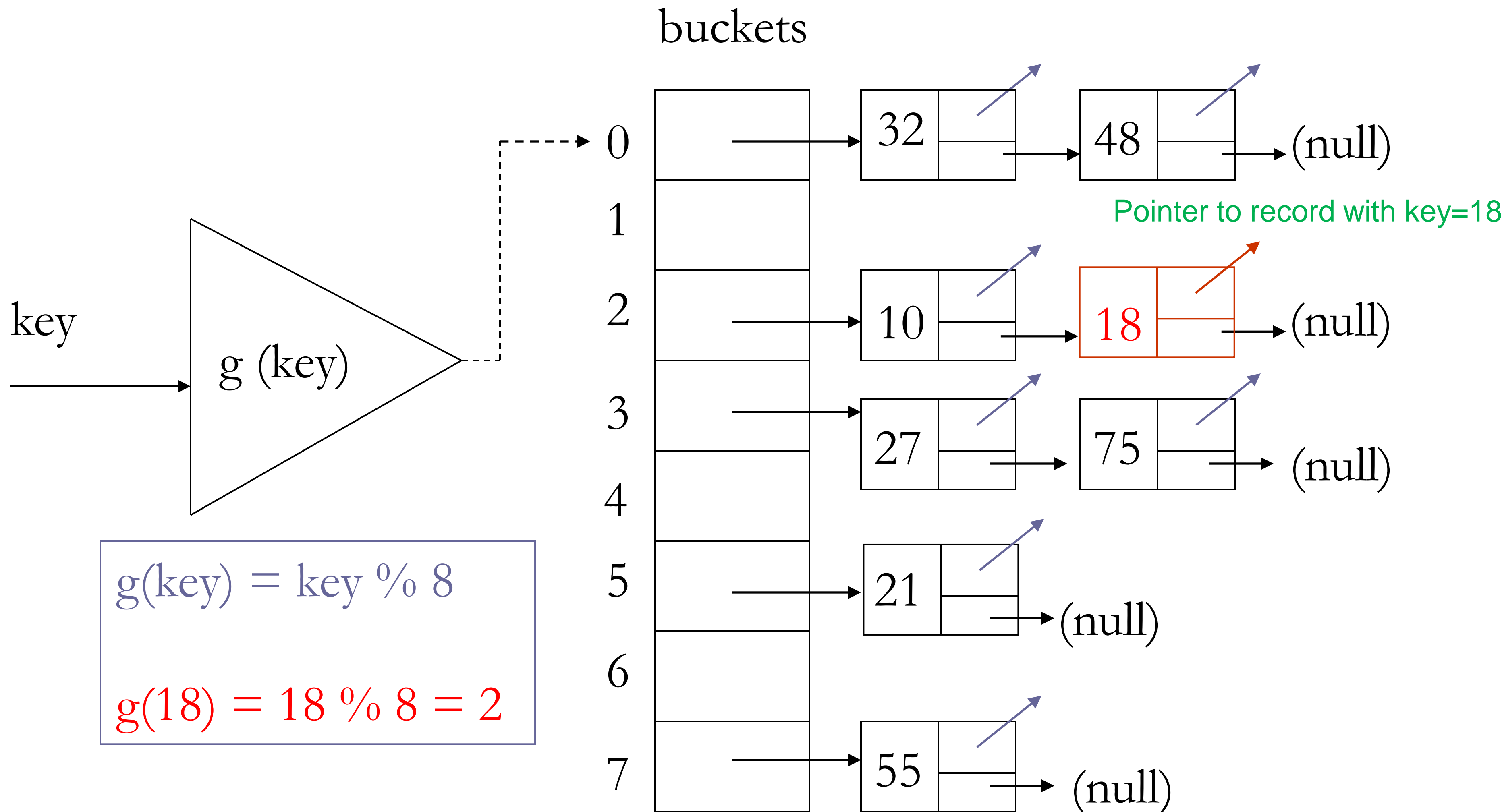
Bucket	
0	
1	AVAILABLE
2	82011, Nick, Taylor, 555-4014154, PC, Pending
3	91402,Tim, Duncan, 451-2243551,TV, Delivered
4	

Chaining (Main Memory example)

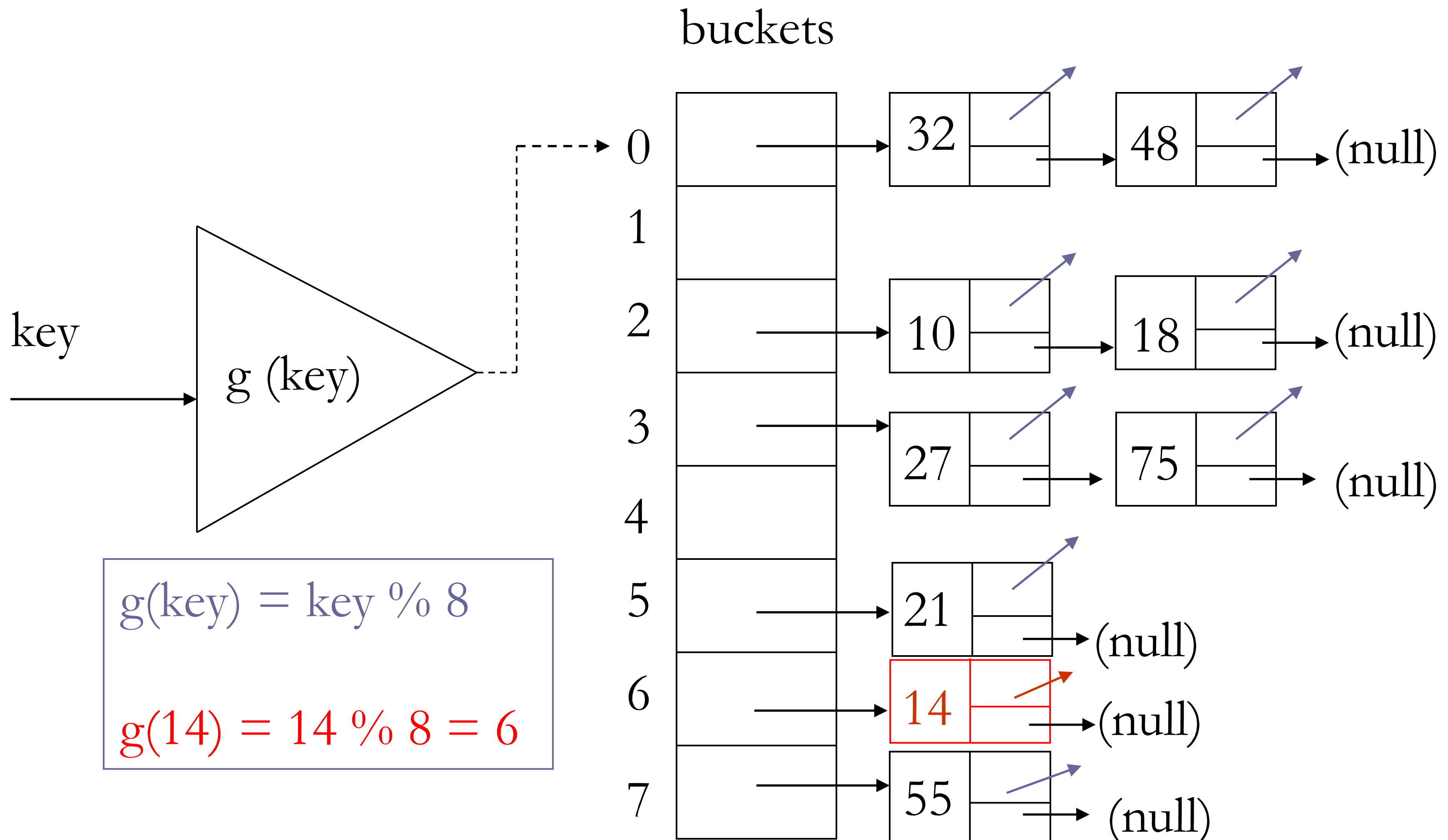


Insert record with key = 18?

Insert record with key = 18



Insert record with key = 14



Hashing in distributed systems

- Hashing can be used to disperse a large dataset across several nodes (*workers*)
- For example to bypass the storage limitations of using a single node (scale out)

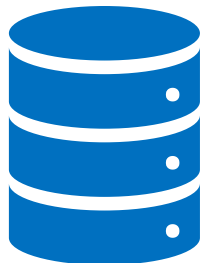
Node id: *0*

1

2

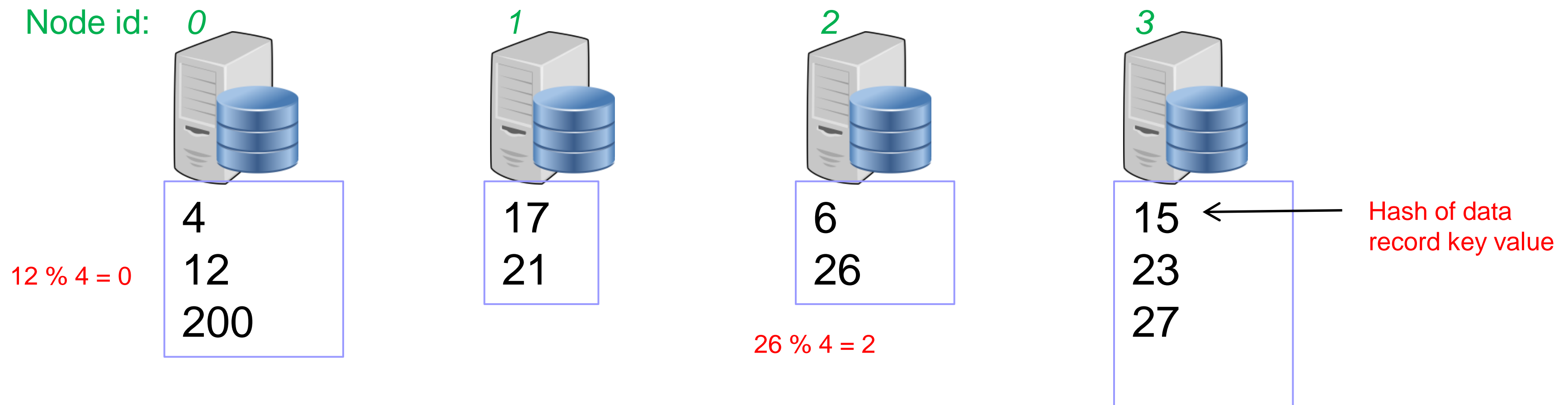
3

Big Data



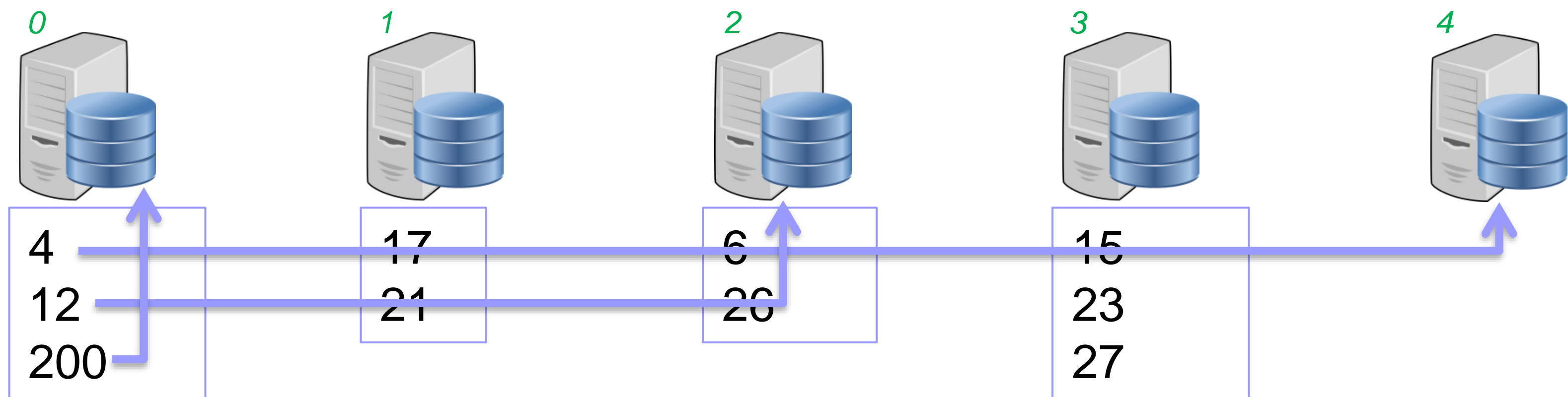
Simple Hashing

- Simple hashing: place record with key = x at location h(x)%N, where N=4 is the number of available nodes in the example bellow



Addition/deletion of nodes necessitates rehashing

- Assume a new (5th) node is added
- Now the placement strategy changes to h(x) % 5
  - What about existing records?

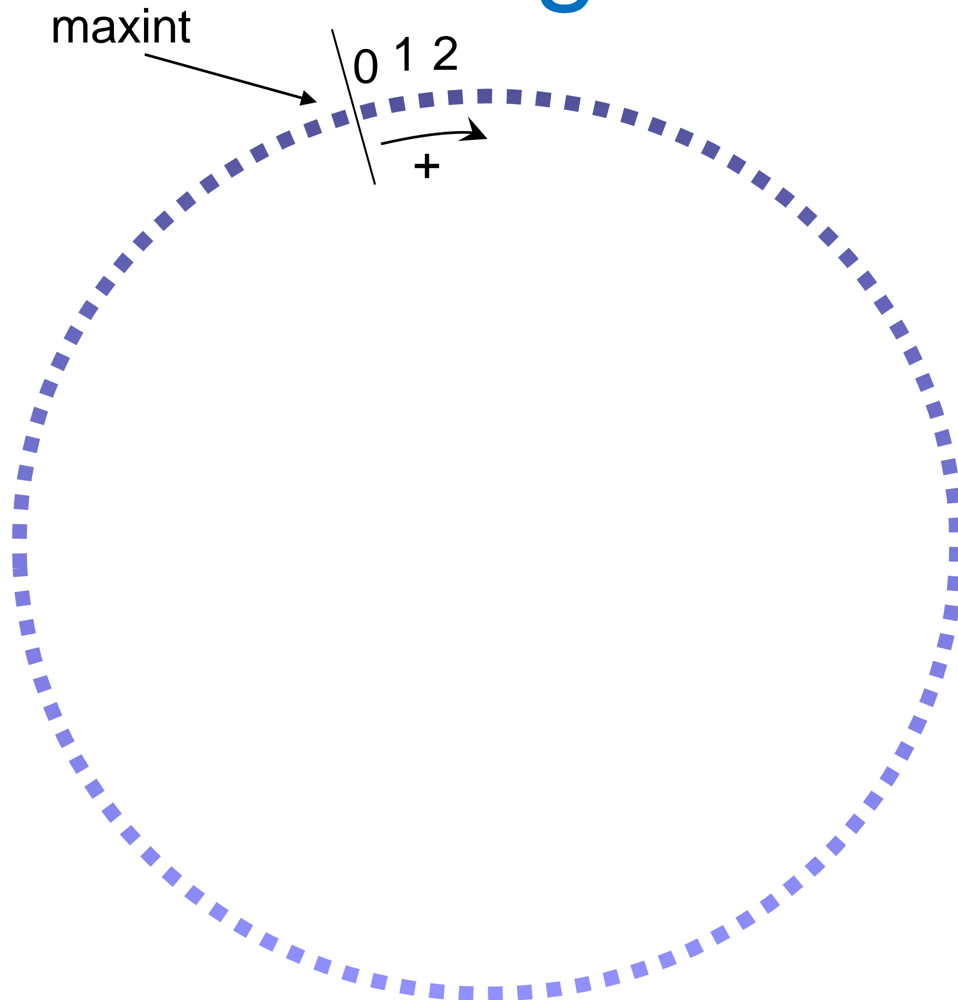


What needs to change in the picture above?

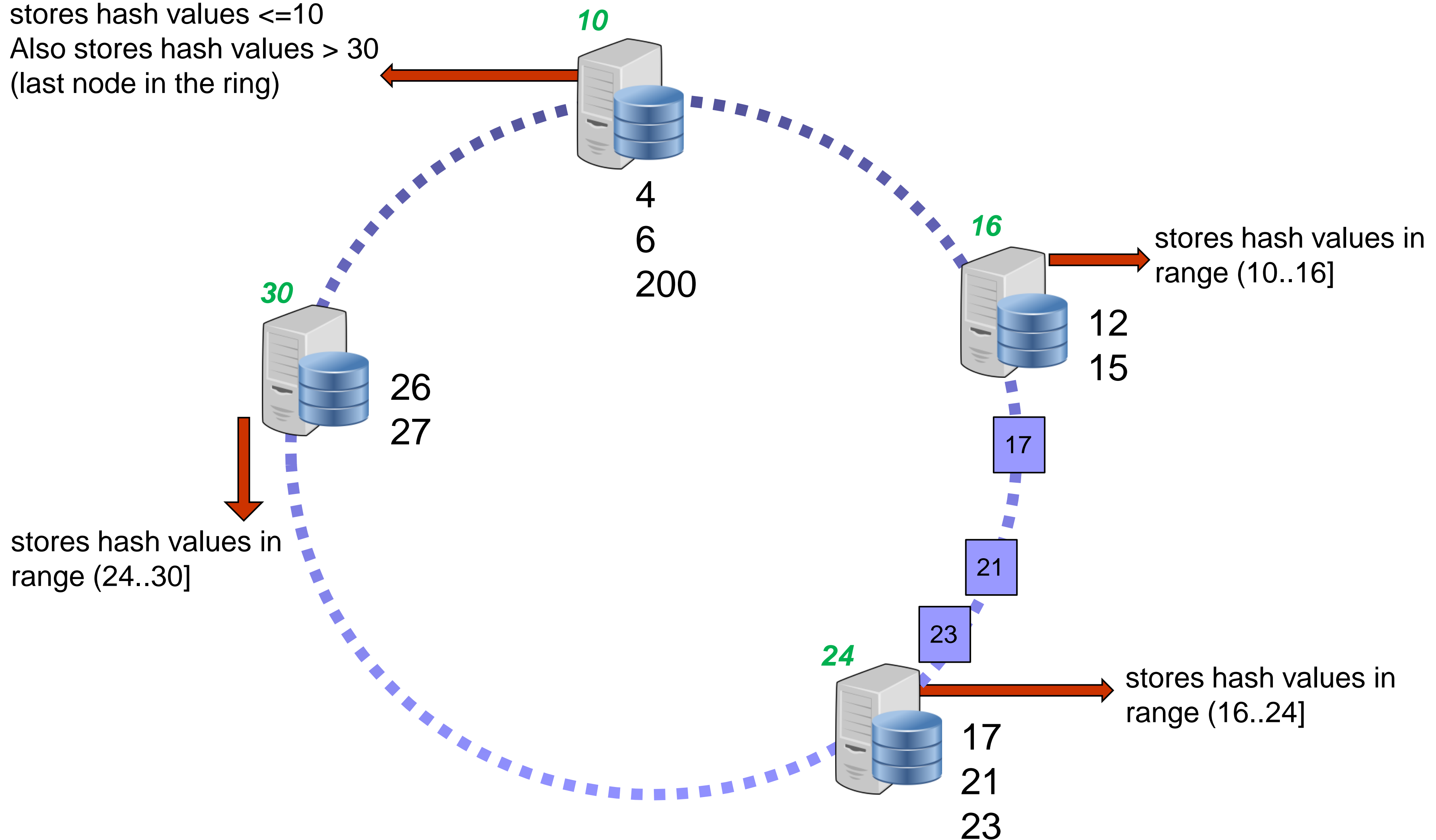
Consistent hashing

- Nodes are hashed in the same domain with data using some unique identifier (their id, mac address etc.)
- Nodes are placed in a virtual ring
- A node with position p is responsible for an individual set of data items whose keys are hashed to an arc (or partition) of the ring between *p.predecessor+1 and p.*

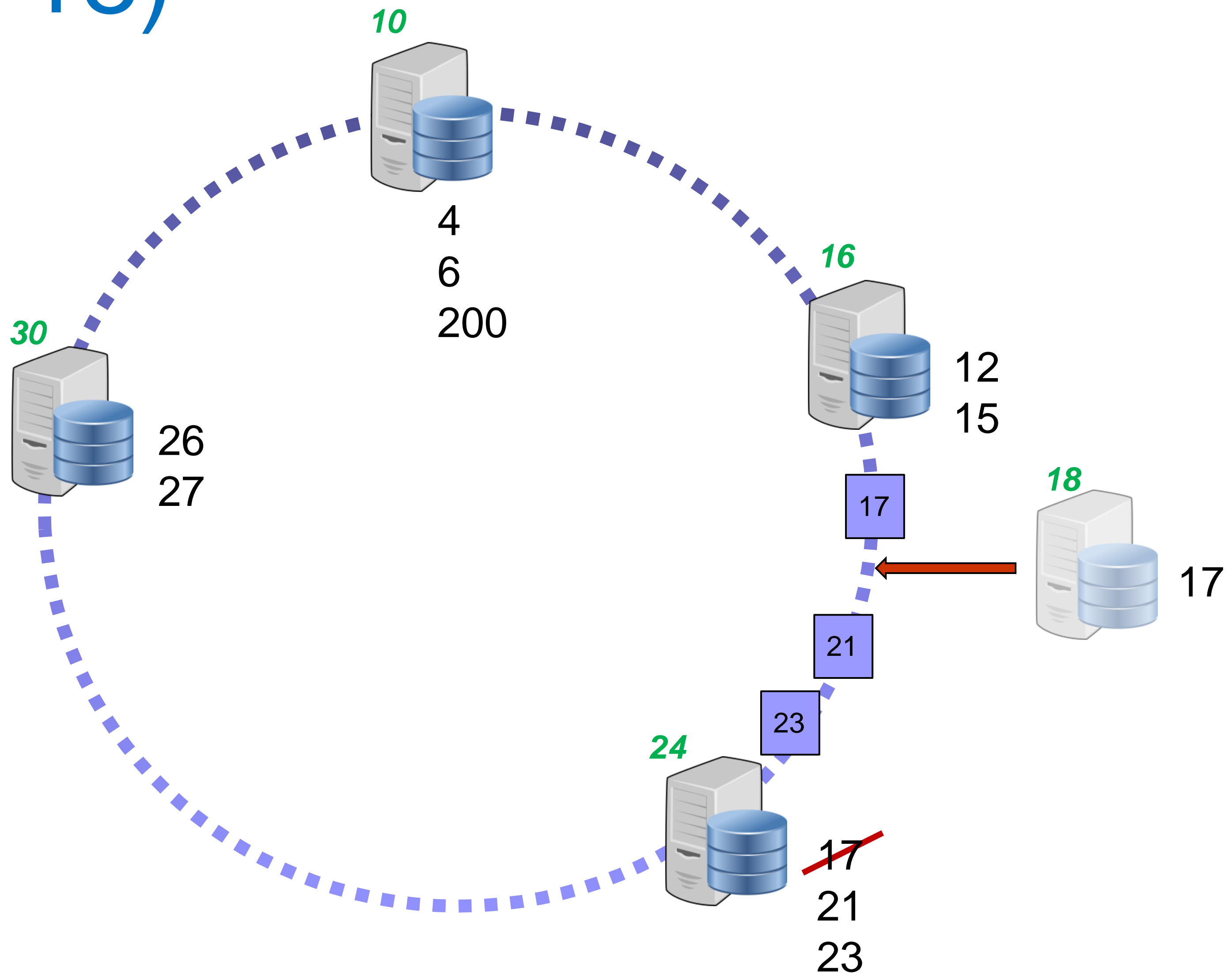
Map integer domain of hash function to a ring



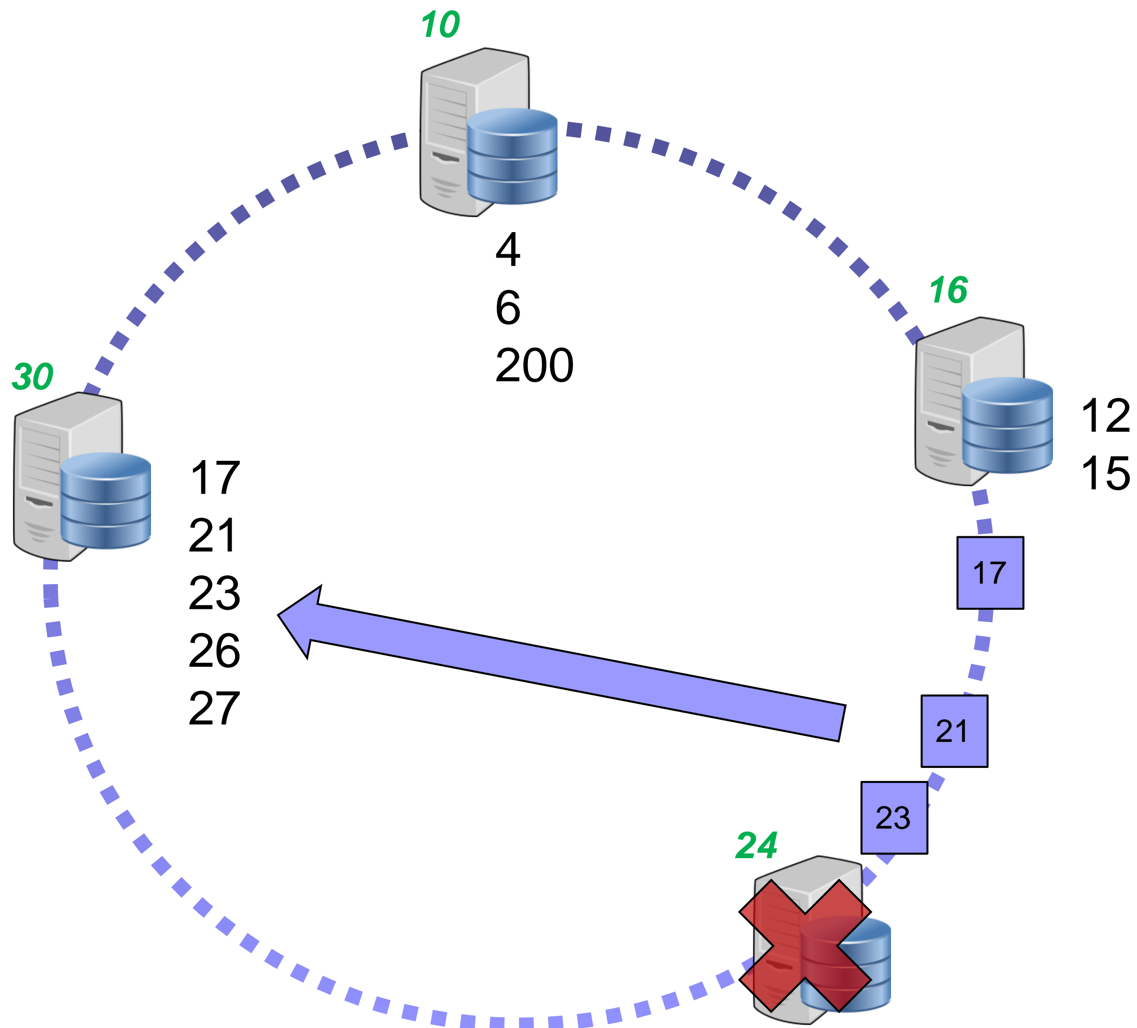
Consistent hashing: hash both data and nodes



Addition of a new node (e.g. 18)



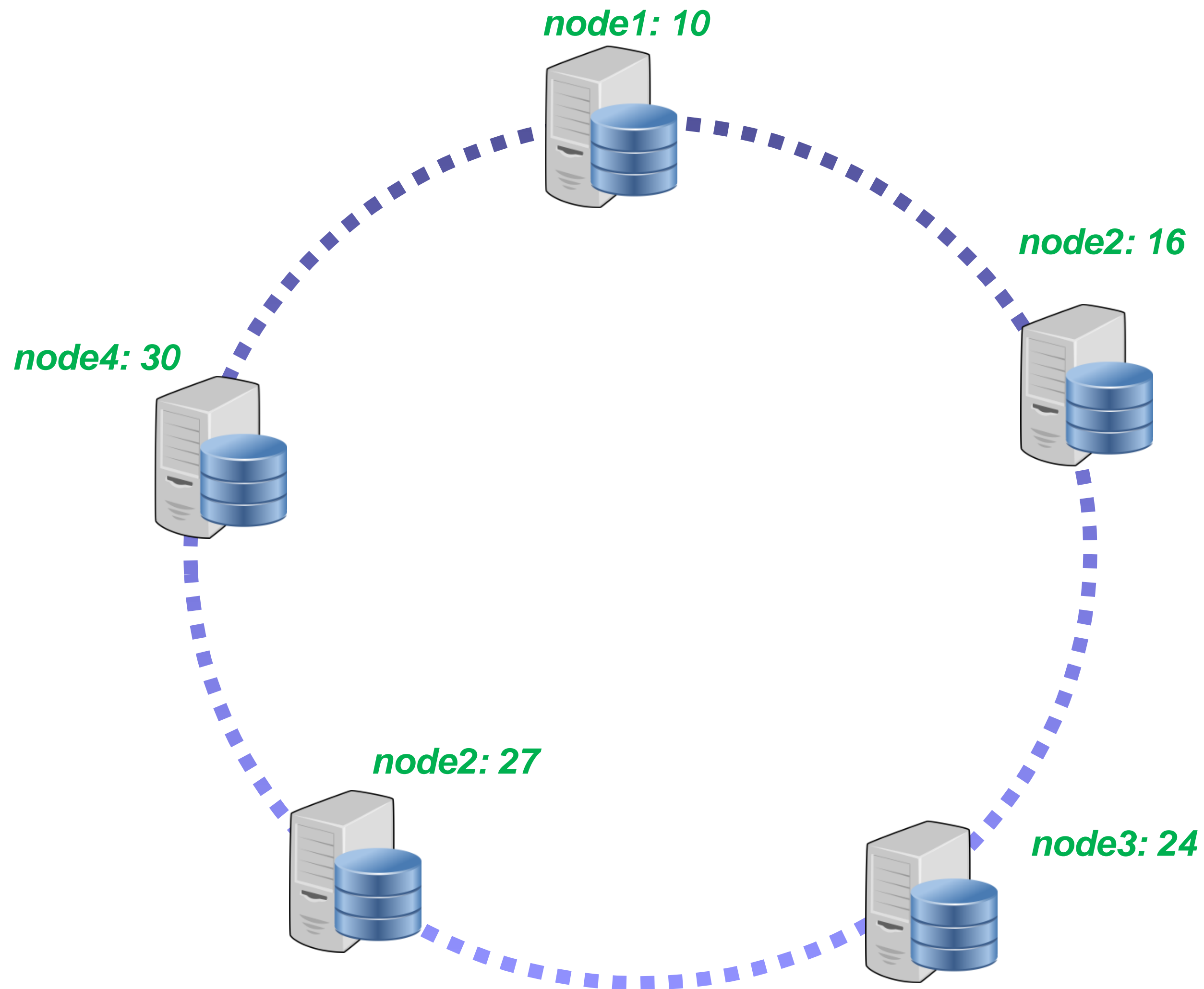
Removal of node 24



Scale-up + Scale out

- Assume that nodes have different capacities
- For instance, assume that node 2 is twice as powerful compared to the rest of the server pool
- Idea: hash node multiple times (twice in this example) so that it receives more data

Node 2 hashed twice



Hashing as in "*divide and conquer*"

- Wikipedia:

  Break down a tough problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Relational Join (⋈) Example:

R

B	C
a	10
a	20
b	10
d	30

⋈

C	D
10	cat
40	dog
15	bat
20	rat

S

Join output:

(a,10,10,cat)

(a,20,20,rat)

(b,10,10,cat)

When does this fall under the tough problem category?

Εφαρμογή: R JOIN S

- Ας υποθέσουμε αρχικά ότι και οι 2 σχέσεις βρίσκονται στην κύρια μνήμη
  - Στη συνέχεια θα δούμε τι γίνεται όταν οι εγγραφές βρίσκονται στο δίσκο

Table R	Table S
1	1
8	0
2	4
3	2
7	7
7	9

Ας χρησιμοποιήσουμε «nested loops»

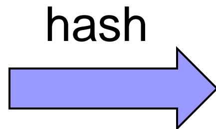
CPU-Cost = number of comparisons = ?

Divide and Conquer

hash function
h(x) = x mod 2

Table R

1
8
2
3
7
7



R.bucket-0

8 2

R.bucket-1

1 3 7 7

S.bucket-0

0 4 2

S.bucket-1

1 7 9

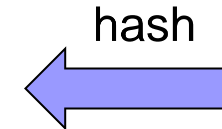


Table S

1
0
4
2
7
9

CPU-Cost = ????

Hash Join - Ιδέα

- Κατακερματίζουμε τις 2 σχέσεις χρησιμοποιώντας την ίδια συνάρτηση h() στο κοινό γνώρισμα της σύζευξης (έστω x)
  - Μία εγγραφή r της σχέσης R κάνει join με την εγγραφή s της σχέσης S → h(r.x)=h(s.x)
  - Επομένως θα ελέγξω αν κάνουν join μόνο εγγραφές των 2 σχέσεων που αντιστοιχίζονται στο ίδιο bucket μέσω της h()
    - Για τί τύπου join ισχύει το παραπάνω;

Αλγόριθμος Hash Join

- Hash join (conceptual)
  - Hash function h, range 1 → k
  - Buckets for R1: G1, G2, ... Gk
  - Buckets for R2: H1, H2, ... Hk

<u>Algorithm</u>

(1) Hash R1 tuples into G1--Gk

(2) Hash R2 tuples into H1--Hk

(3) For i = 1 to k do

Match tuples in Gi, Hi buckets

Notation

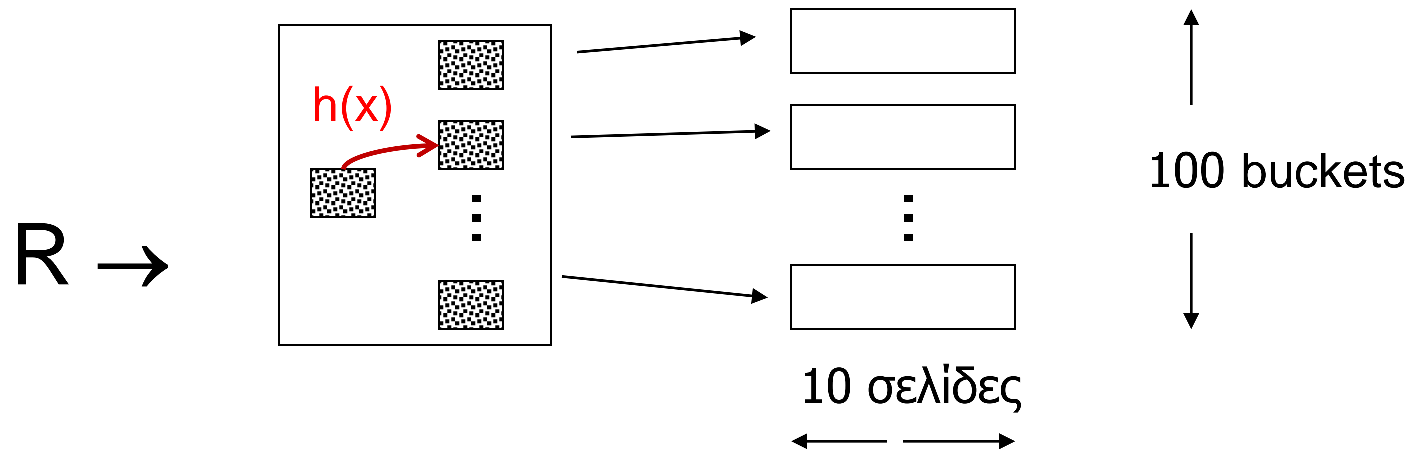
- B(R) = #blocks που καταλαμβάνουν οι εγγραφές της σχέσης R στο δίσκο
- T(R) = #tuples = αριθμός πλειάδων (εγγραφών) της σχέσης R

Παράδειγμα Hash Join

- B(R)=1000, B(S)=500

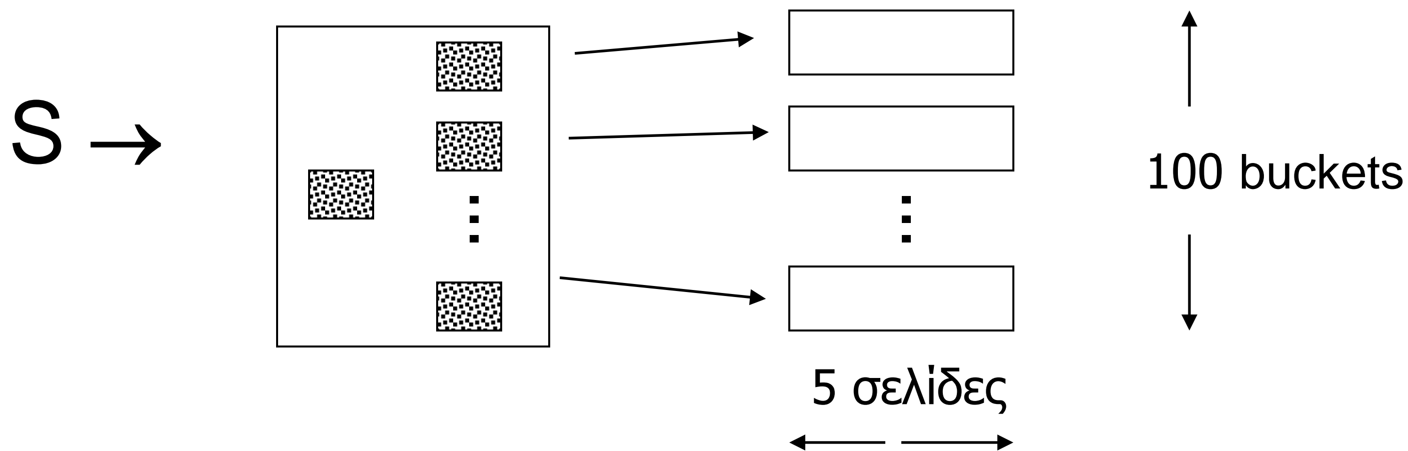
→ Use k=100 buckets

→ Read R, hash, + write buckets



Ίδια διαδικασία για την S

B(S)=500 σελίδες



Επόμενο βήμα

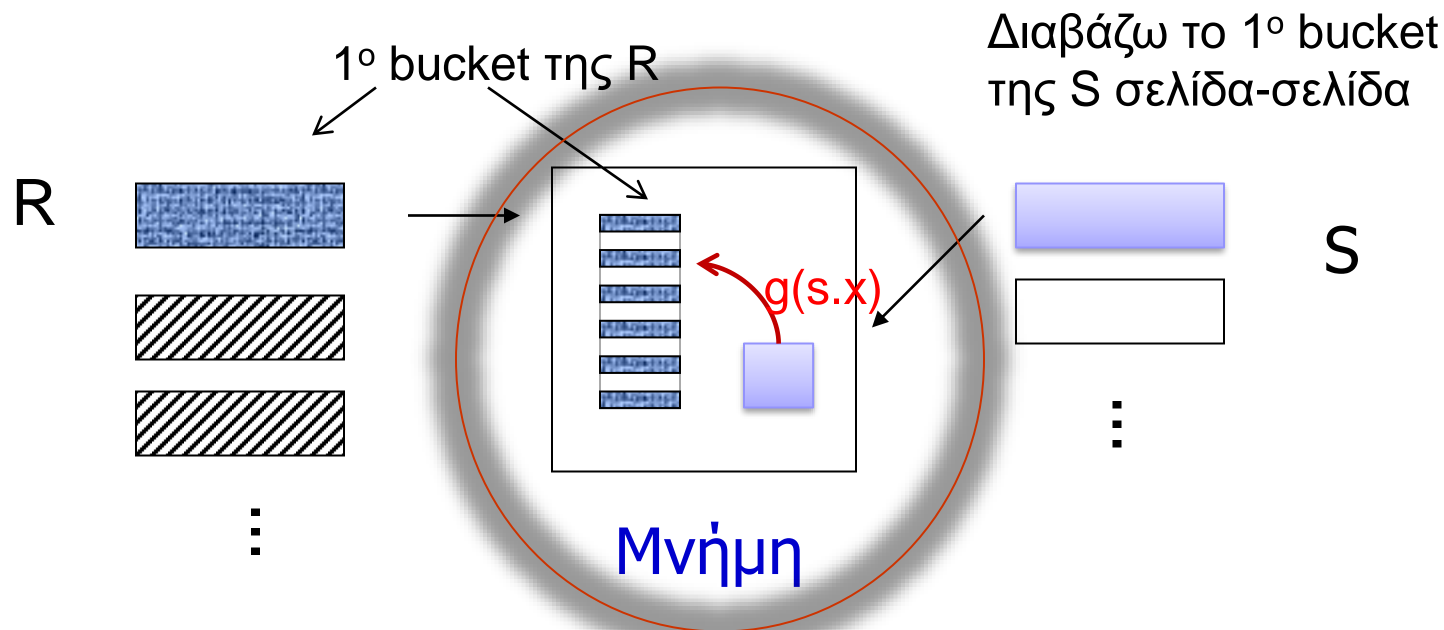
- Διάβασε το πρώτο bucket της R (~10 σελίδες)
  - Διάβασε μία μία τις σελίδες του πρώτου bucket της S
  - Έλεγξε αν οι εγγραφές της R στη μνήμη κάνουν join με τις εγγραφές στη μνήμη της S
  - Πως;

Hash-Join στη μνήμη

- Διάλεξε μία διαφορετική συνάρτηση κατακερματισμού g() και κάνε hash τις εγγραφές στη μνήμη από το πρώτο bucket της R
  - Οι κάδοι που φτιάχνονται είναι στη μνήμη
- Για κάθε εγγραφή s από τη σελίδα της S που έχεις στη μνήμη ψάξε στο bucket g(s.x) της R για εγγραφές που κάνουν join
  - Το πρώτο bucket της R στη μνήμη έχει σπάσει σε μικρότερα buckets με βάση την g()

-->Read one R bucket; build memory hash table
[R is called the build relation of the hash join]

-> Read **corresponding** S bucket + hash probe
[S is called the probe relation of the hash join]



Επαναλαμβάνω την ίδια διαδικασία για τα υπόλοιπα buckets

Κόστος Hash-Join:

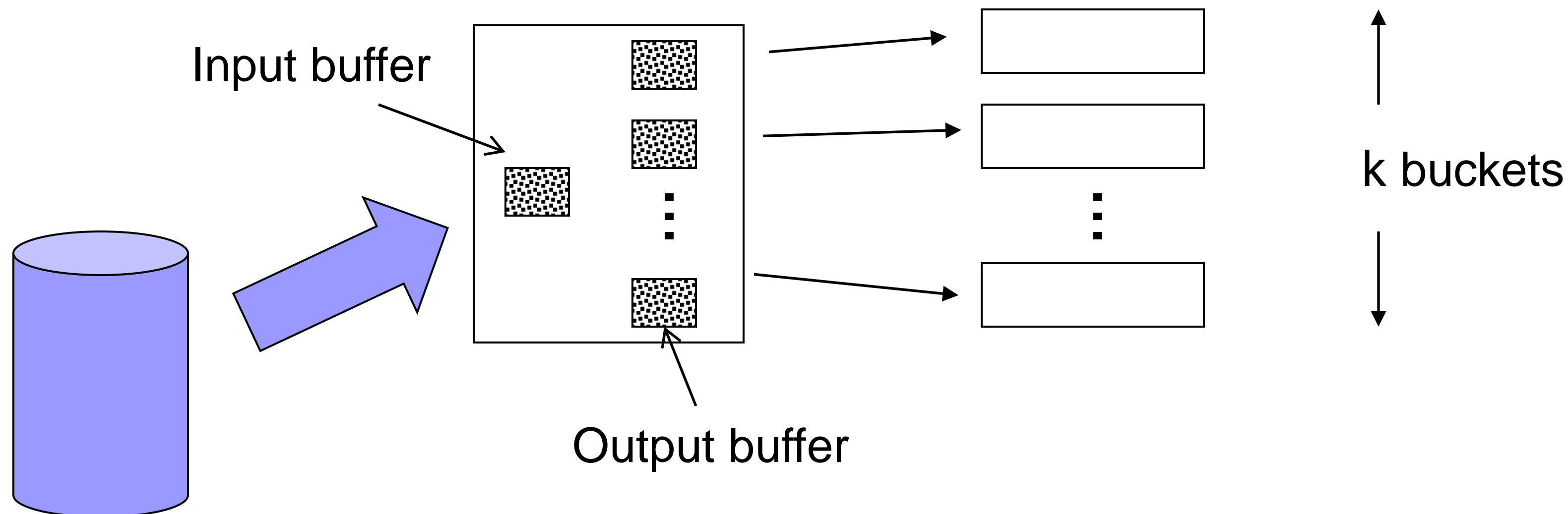
"Bucketize:"	Read R + write buckets
	Read S + write buckets
Join:	Read R, S buckets

Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Απαιτήσεις σε μνήμη

- Αριθμός buckets $k \leq M-1$
  - Όταν φτιάχνω τα buckets θέλω output buffer 1 σελίδα για κάθε bucket + 1 σελίδα input buffer



Απαιτήσεις σε μνήμη

- Μέγιστος αριθμός buckets $k = M-1$(μιας και θέλω 1 output buffer για το καθένα στη πρώτη φάση, δες προηγούμενο σχήμα)
- Στη δεύτερη φάση τα buckets της R πρέπει να χωράνε στη μνήμη
- (Αναμενόμενο) μέγεθος bucket της R σε σελίδες: $B(R)/k$

  $= B(R)/(M-1) \leq M-1 \rightarrow B(R) \leq (M-1)^2$

  $\rightarrow M > \sqrt{B(R)}$
- Το μέγεθος της σχέσης S δεν παίζει ρόλο
  - Άρα ως «εξωτερική» σχέση διαλέγω τη μικρότερη!

Απαιτήσεις σε μνήμη Hash-Join

$M > \sqrt{\min(B(R),B(S))}$

Στο παράδειγμα μας $M > \sqrt{500} = 23$ σελίδες

Παρατήρηση

- Αν μία από τις δύο σχέσεις χωράει στη μνήμη (πχ η R) το join μπορεί να εκτελεστεί σε ένα πέρασμα
  - Κάνω hash την R στη μνήμη
  - Διαβάσω μία μία τις εγγραφές της S
    - Κάνω probe με βάση την τιμή h(s.x)
    - Αν στον κάδο που θα καταλήξω βρω εγγραφές της R που κάνουν join επιστρέφω το αποτέλεσμα
- Κόστος=B(R)+B(S)

Βελτιστοποιήσεις

- Έστω η σχέσεις Plays3D(Title, Cinema, Address) και Movies(Title, Actor, Trailer)
- Το γνώρισμα Trailer είναι ένα MP4 αρχείο μερικών MB
- Όταν κάνω Sort ή Hash Join τις εγγραφές πχ με βάση το γνώρισμα Title δε κρατάμε όλη την εγγραφή της Movies αλλά το ζεύγος <Title,Rec_pointer>
  - Rec_pointer είναι δείκτης προς την εγγραφή στο δίσκο
  - Μειώνουμε τις απαιτήσεις σε μνήμη (πιθανών πλέον η σχέση να χωράει στη μνήμη και το sort/hash να γίνει σε ένα πέρασμα)
- Μειονέκτημα: στο τέλος για όσες εγγραφές κάνουν join πρέπει να ακολουθήσω τους pointers για να πάρω τα υπόλοιπα γνωρίσματα

Παράδειγμα

- Plays3D(Title, Cinema, Address)
  - B(Plays3D)=1000 σελίδες, T(Plays3D)=10000 εγγραφές
- Movies(Title, Actor, Trailer)
  - B(Movies)=10000 σελίδες, T(Movies) =100 εγγραφές
- 20 ζεύγη <Title,Rec_pointer> χωράνε σε κάθε σελίδα
- M=50 σελίδες
- Αλγόριθμος Hash-Join

Συμβατική εκτέλεση

- Plays3D(Title, Cinema, Address) B(Plays3D)=1000 σελίδες, T(Plays)=10000 εγγραφές
- Movies(Title, Actor, Trailer) B(Movies)=10000 σελίδες, T(Movies) = 100 εγγραφές

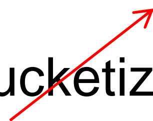
- "Εξωτερική" σχέση η Plays3D
  - $B(Plays3D)=1000 < M^2 = 50*50=2500$
- Κόστος = 3*(1000+10000) = 33000 I/O

Με το τρικ των pointers

- Χωράνε 20 ζεύγη <Title,Rec_pointer> σε κάθε σελίδα
- Οι 100 εγγραφές της Movies δημιουργούν 100 ζεύγη και χωράνε σε 5 σελίδες !
- Hash Join σε ένα πέρασμα:
  - Φτιάχνω και κρατάω τα buckets της Movies στη μνήμη
  - Διαβάζω μία-μία τις σελίδες της Plays3D. Για κάθε εγγραφή υπολογίζω την τιμή h(Title) και ψάχνω στο αντίστοιχο bucket της Movies για εγγραφές με τον ίδιο τίτλο
  - Αν βρω εγγραφή, ακολουθώ τον rec_pointer για να διαβάσω τα 2 γνωρίσματα που λείπουν (Actor και Trailer).
  - Έστω 50 τέτοιες εγγραφές στο αποτέλεσμα της σύζευξης

Με τη βελτιστοποίηση

- Κόστος =

  Read Movies + Bucketize στη μνήμη

  + Read Plays3D

  + Follow 50 Pointers

  = 10000 + 1000 + 50 = 11050

  Σύγκρινε το με το 33000 του συμβατικού αλγόριθμου.

  …επίσης πολύ λιγότερες απαιτήσεις σε μνήμη (πόσες?)