Hashing

Yannis Kotidis

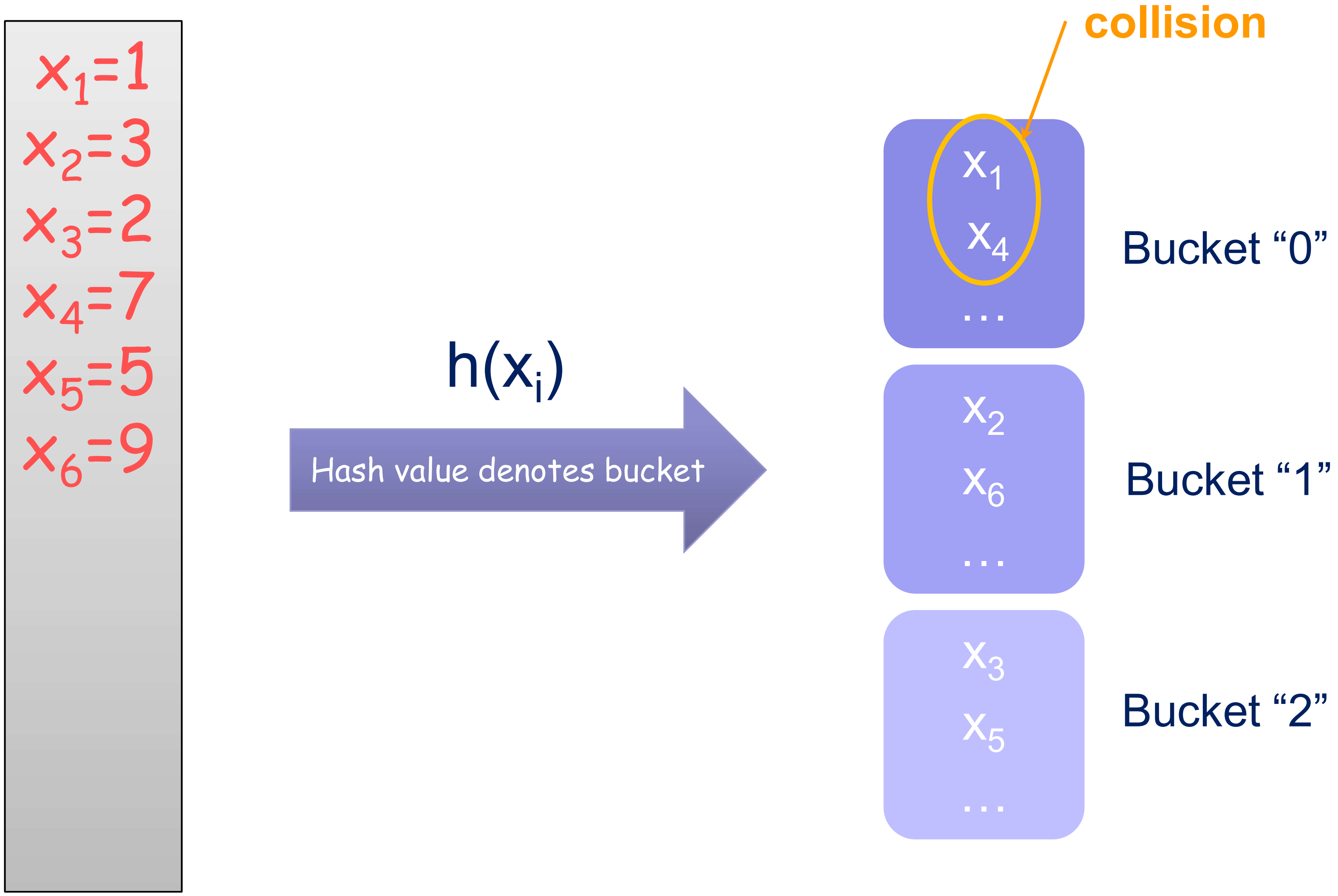
What is hashing?

- 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
  - We can 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: $h(x)=(2x+1)\ \%\ 3$



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

$h(x_i)$

Hash value denotes bucket

collision

x_1 x_4 … Bucket "0"

x_2 x_6 … Bucket "1"

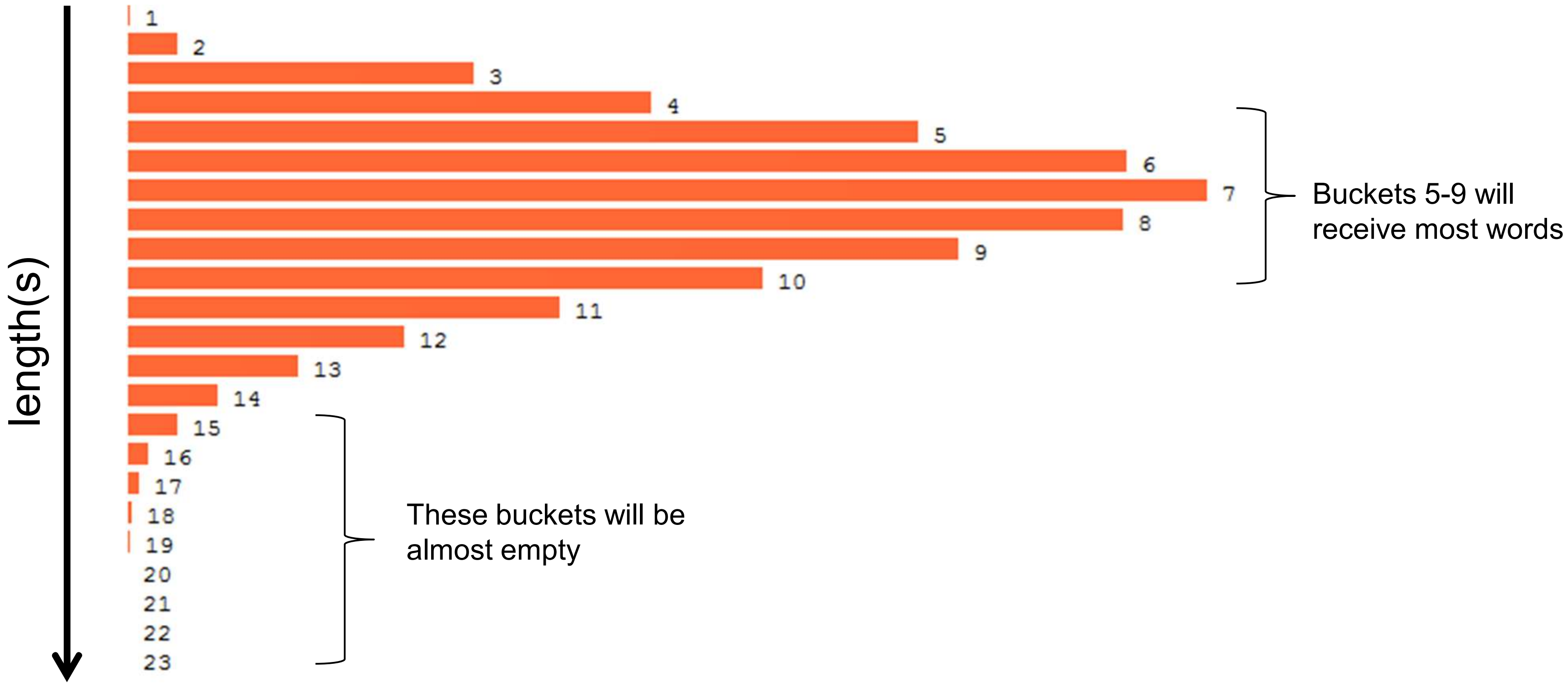
x_3 x_5 … Bucket "2"

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:



Buckets 5-9 will receive most words

These buckets will be almost empty

A better function (input string s)

- s[i] = ith character in string
- K=some (large) constant
- Recursively compute
  - 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=299)

- Ascii code of ‘a’,’b’ and ’c’ is 97, 98 and 99, respectively
- h(‘abc’) = (a*K+b)*K+c = aK^2+bK+c =$97*299^2+98*299+99=8701298$

- Compare to h(‘acb’) = … = 8701596

Note

- Previous function may return arbitrarily large numbers
  - h(‘supercalifragilisticexpialidocious‘)= 389236099458587451617003512335442884432133029560316 1825327689504791395104502384955 (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

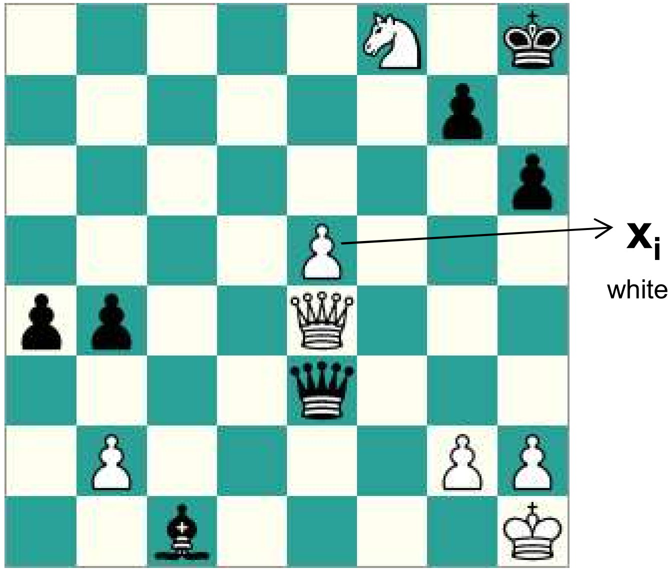
- Assume a, b are randomly chosen integers and $a \neq 0$
- Given a prime number p, with $p \geq m$
- Then, the following family of hash functions is universal:

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

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

Hashing as an index for Chess Games

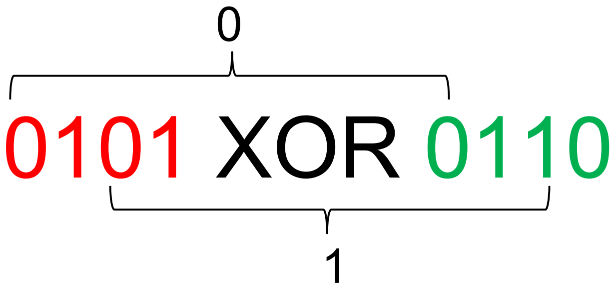
- Zorbist hashing:
  - Generate an array of 781 64 bit random numbers
  - One number for each piece at a position (2*6*64 total)
    - 6 pieces: king, queen, rooks, bishops, knights, pawns
    - 8*8 positions, 2 colors
  - 13 additional numbers encoding side to move, castling rights, etc
  - A position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



$$\underbrace{X = x_1 \text{ XOR } x_2 \text{ XOR } \dots x_{14}}_{\text{Use this single value to encode the position}}$$

More on x XOR y

- Result is 1 if input bits differ, 0 otherwise



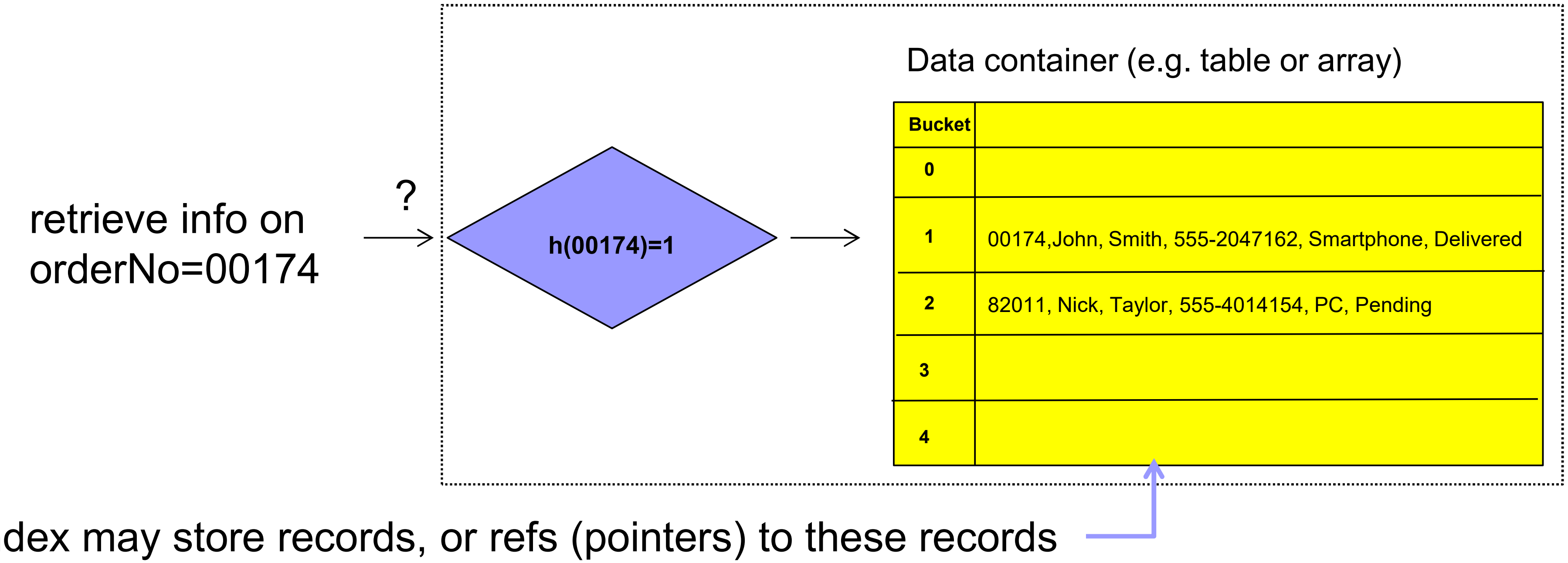
- 0101 XOR 0110 = 0011
- 0011 XOR 0110 = 0101

and

- 0011 XOR 0101 = 0110

Hashing as an index

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



retrieve info on orderNo=00174

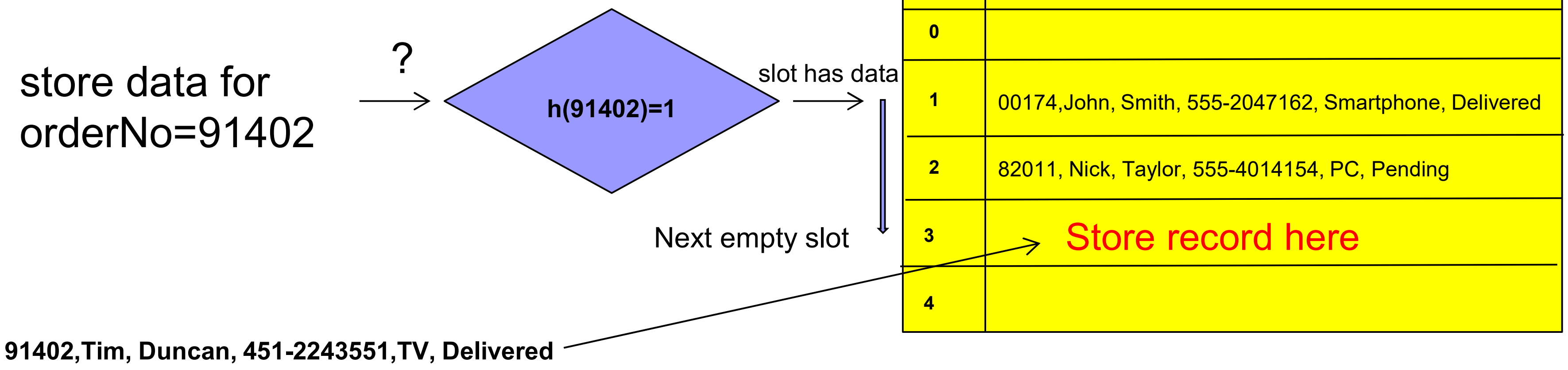
Data container (e.g. table or array)

Bucket	
0	
1	00174,John, Smith, 555-2047162, Smartphone, Delivered
2	82011, Nick, Taylor, 555-4014154, PC, Pending
3	
4	

Index may store records, or refs (pointers) to these records

Handling *collisions*

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



store data for orderNo=91402

Data container

Bucket	
0	
1	00174,John, Smith, 555-2047162, Smartphone, Delivered
2	82011, Nick, Taylor, 555-4014154, PC, Pending
3	Store record here
4	

91402,Tim, Duncan, 451-2243551,TV, Delivered

Deletions are complicated

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

Data container

Bucket	
0	
1	00~~1~~74,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)?

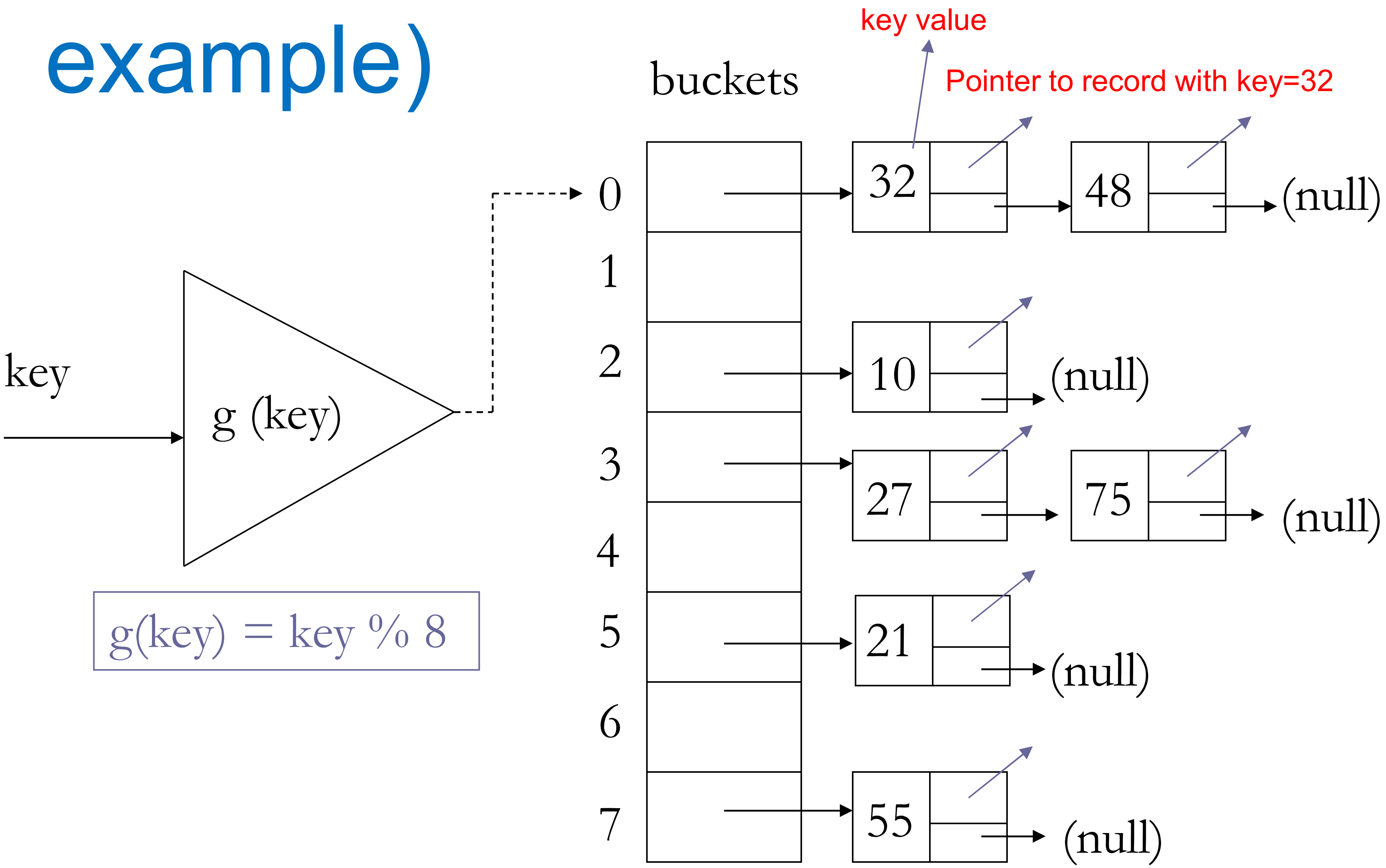
Better strategy

- 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	

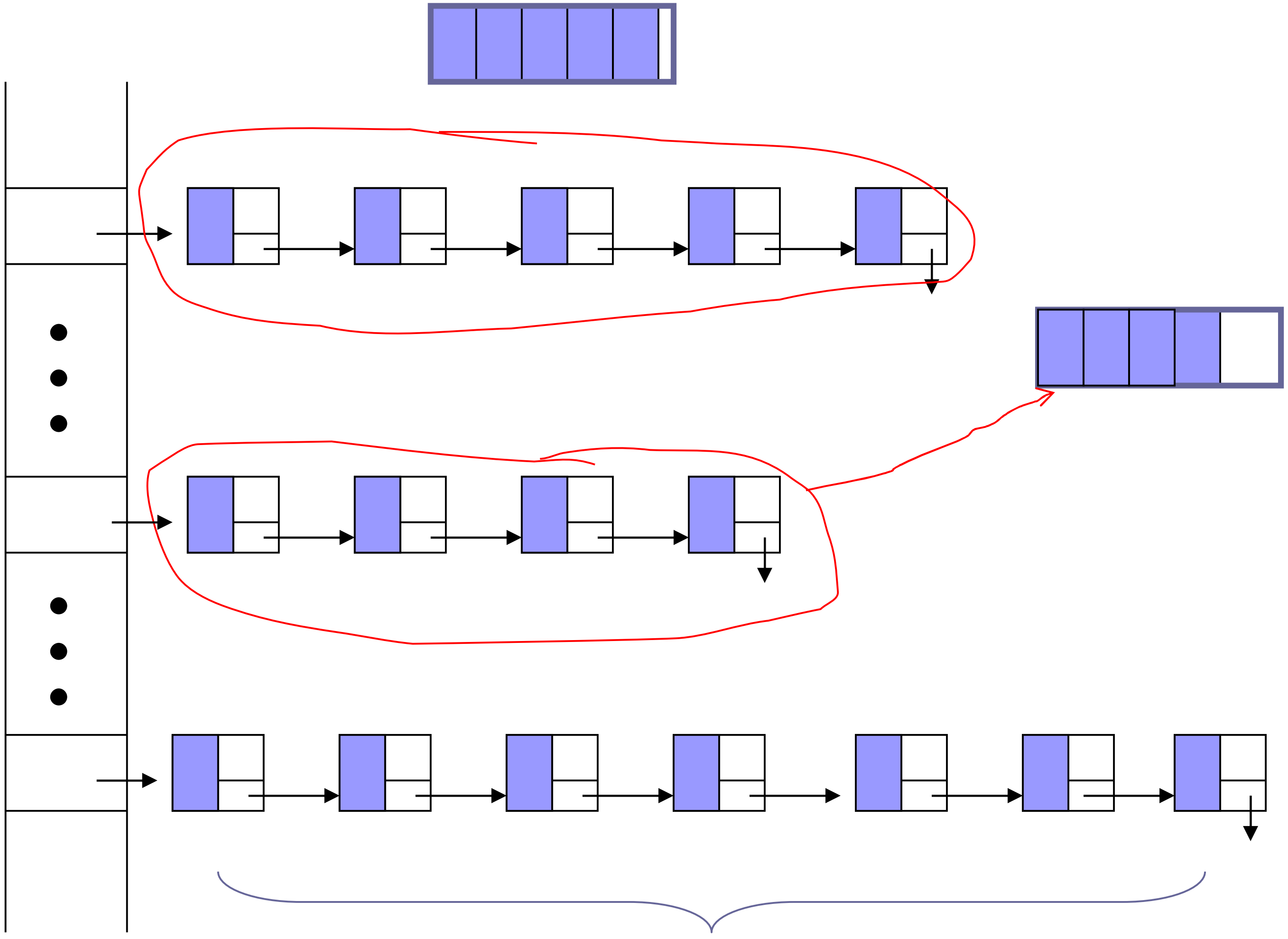
Chaining (Main Memory example)



Adapting to disk

- 1 Hash Bucket = 1 disk block (e.g. 4KB)
  - All keys that hash to bucket stored in the block
  - Intuition: keys in a bucket usually accessed together
  - No need for linked lists of keys …

Adapting to Disk

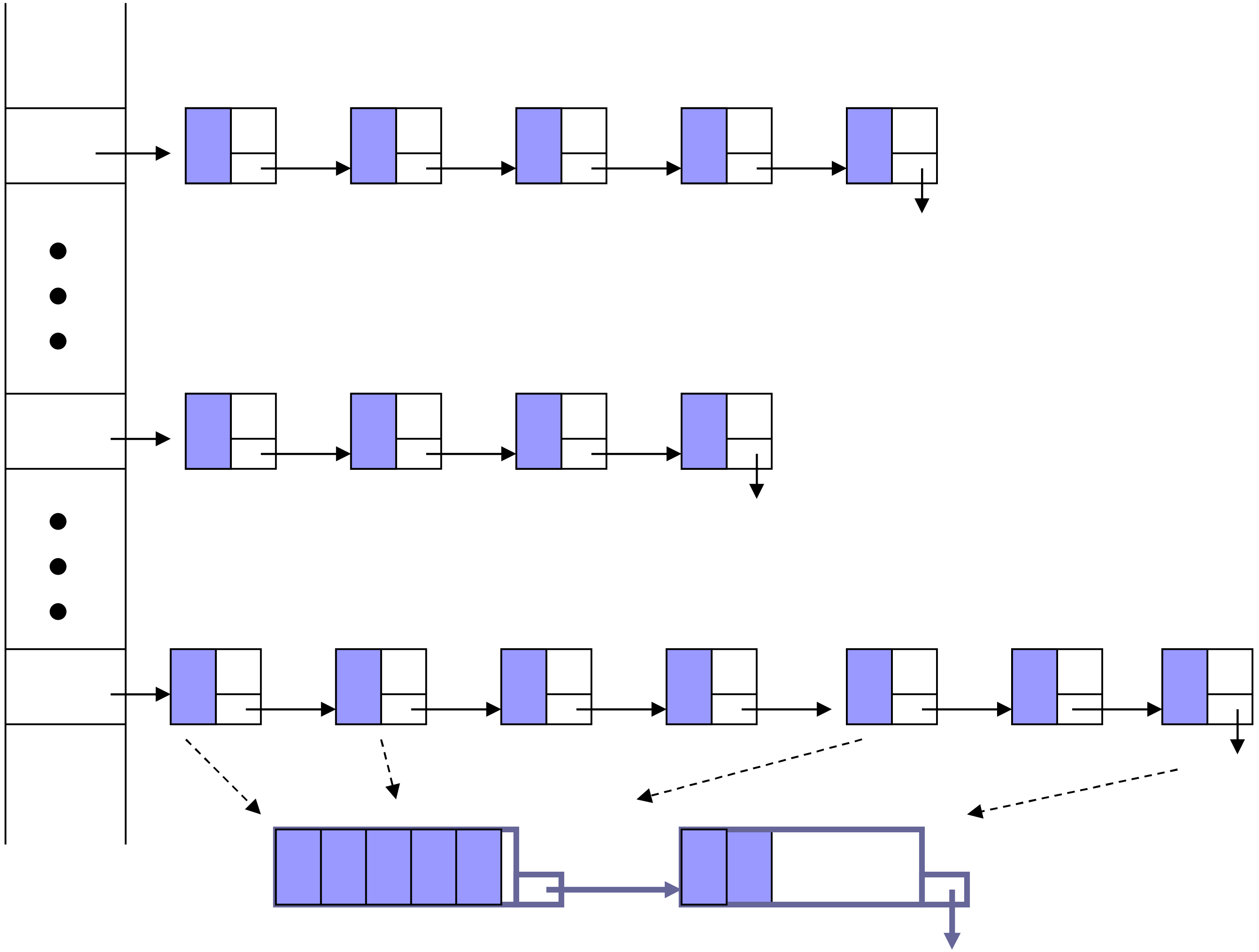


How do we handle this?

Adapting to disk

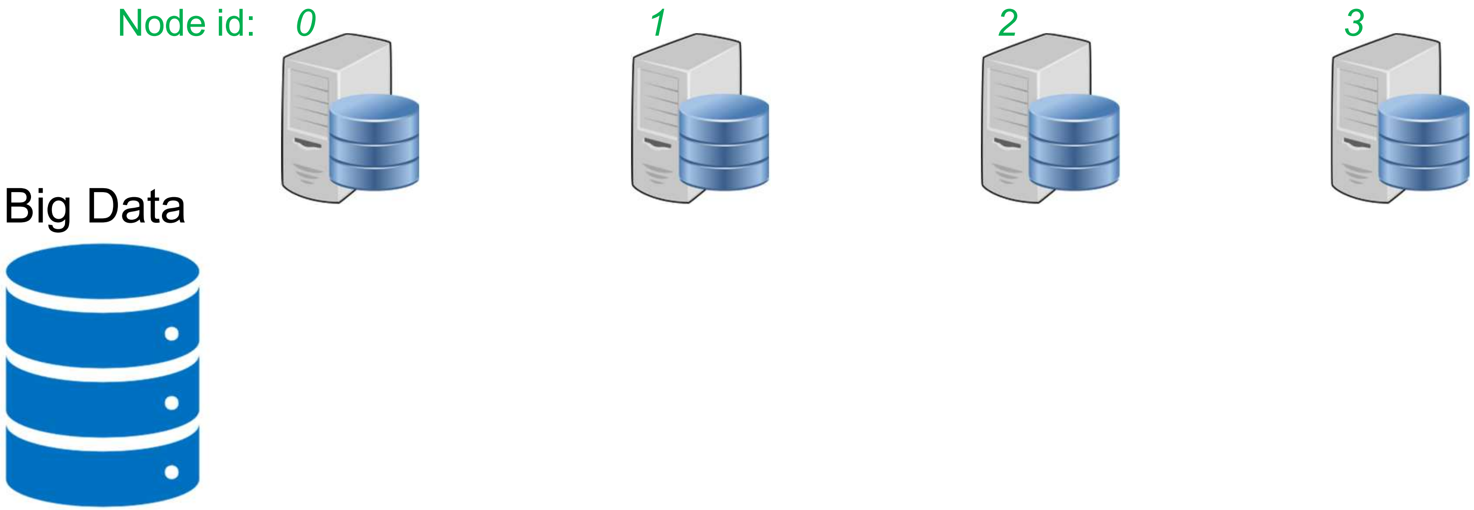
- 1 Hash Bucket = 1 Block
  - All keys that hash to bucket stored in the block
  - Intuition: keys in a bucket are usually accessed together
  - No need for linked lists of keys …
  - … but need linked list of blocks (overflow blocks)

Adapting to Disk



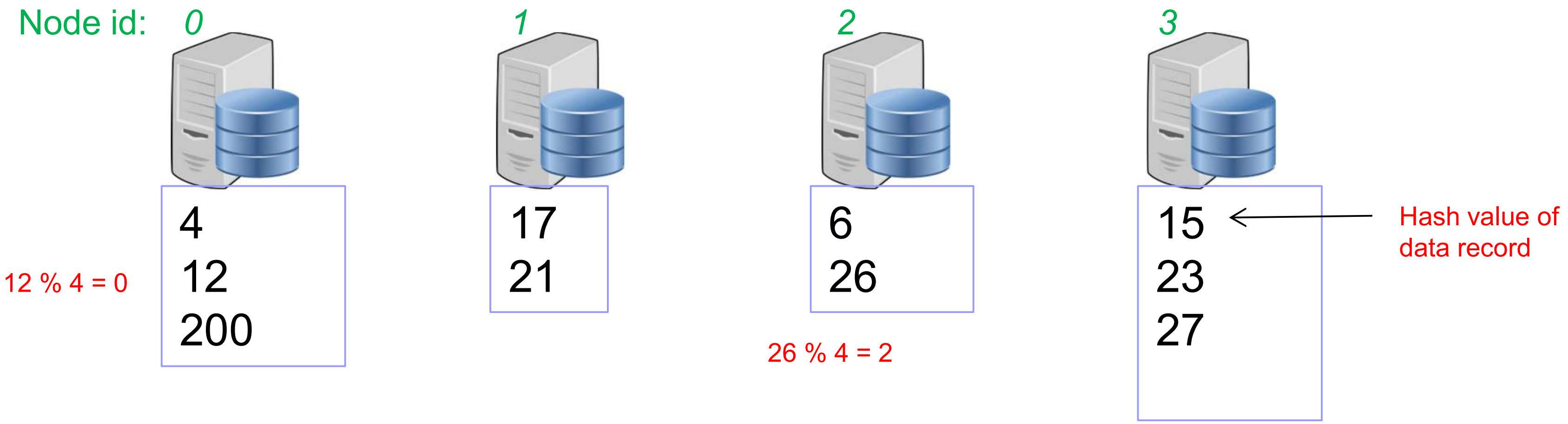
Hashing in distributed systems

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



Simple Hashing

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



Node id: *0*

4
12
200

12 % 4 = 0

1

17
21

2

6
26

26 % 4 = 2

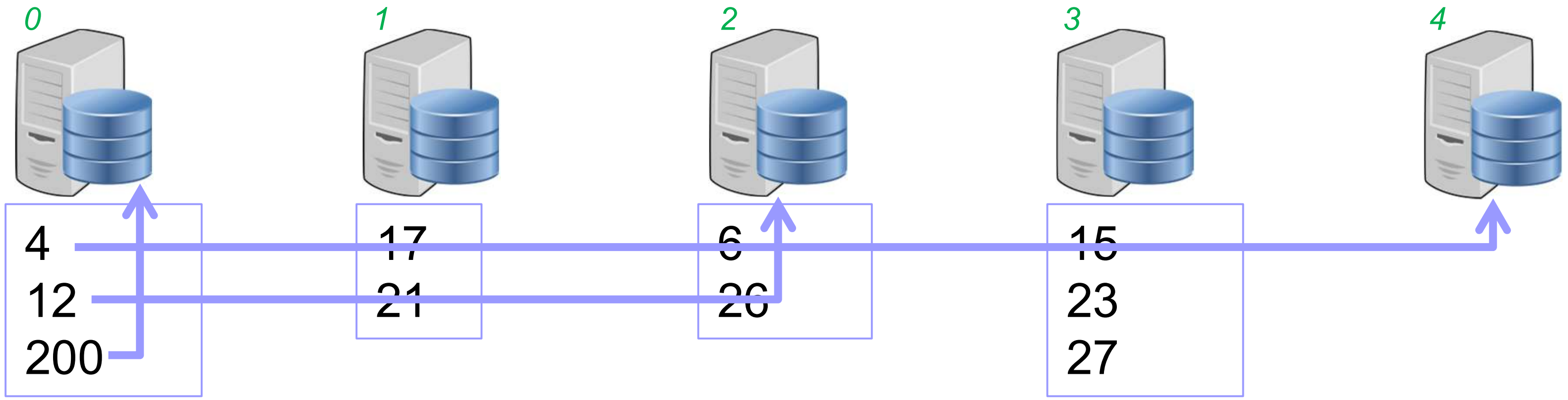
3

15
23
27

Hash value of data record

Addition/deletion of server nodes necessitates rehashing

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



0
4
12
200

1
17
21

2
6
26

3
15
23
27

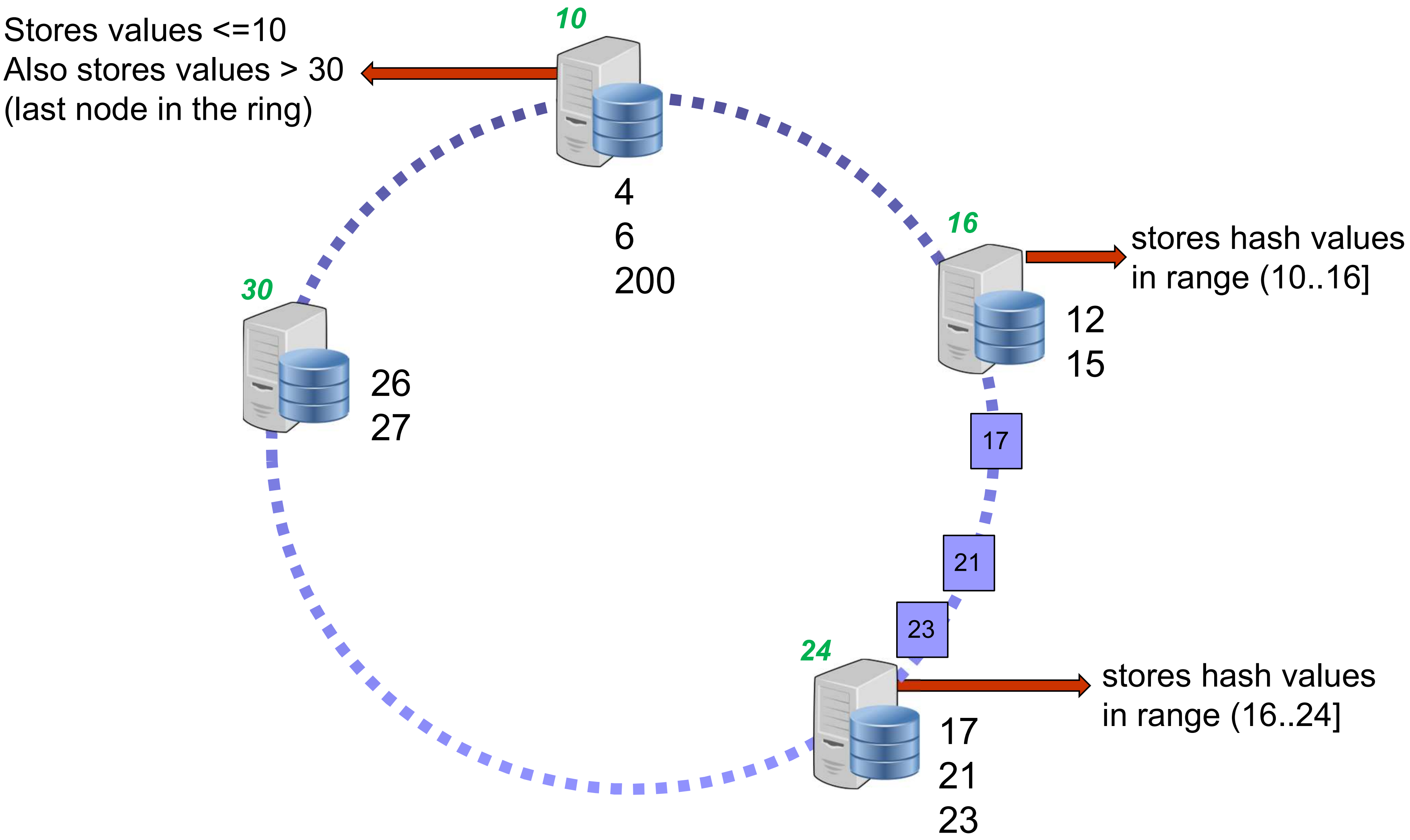
4

What needs to change in the picture above?

Consistent hashing

- Nodes (servers) 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.*

Consistent hashing



Stores values <=10
Also stores values > 30
(last node in the ring)

10

4
6
200

16

stores hash values
in range (10..16]

12
15

30

26
27

17

21

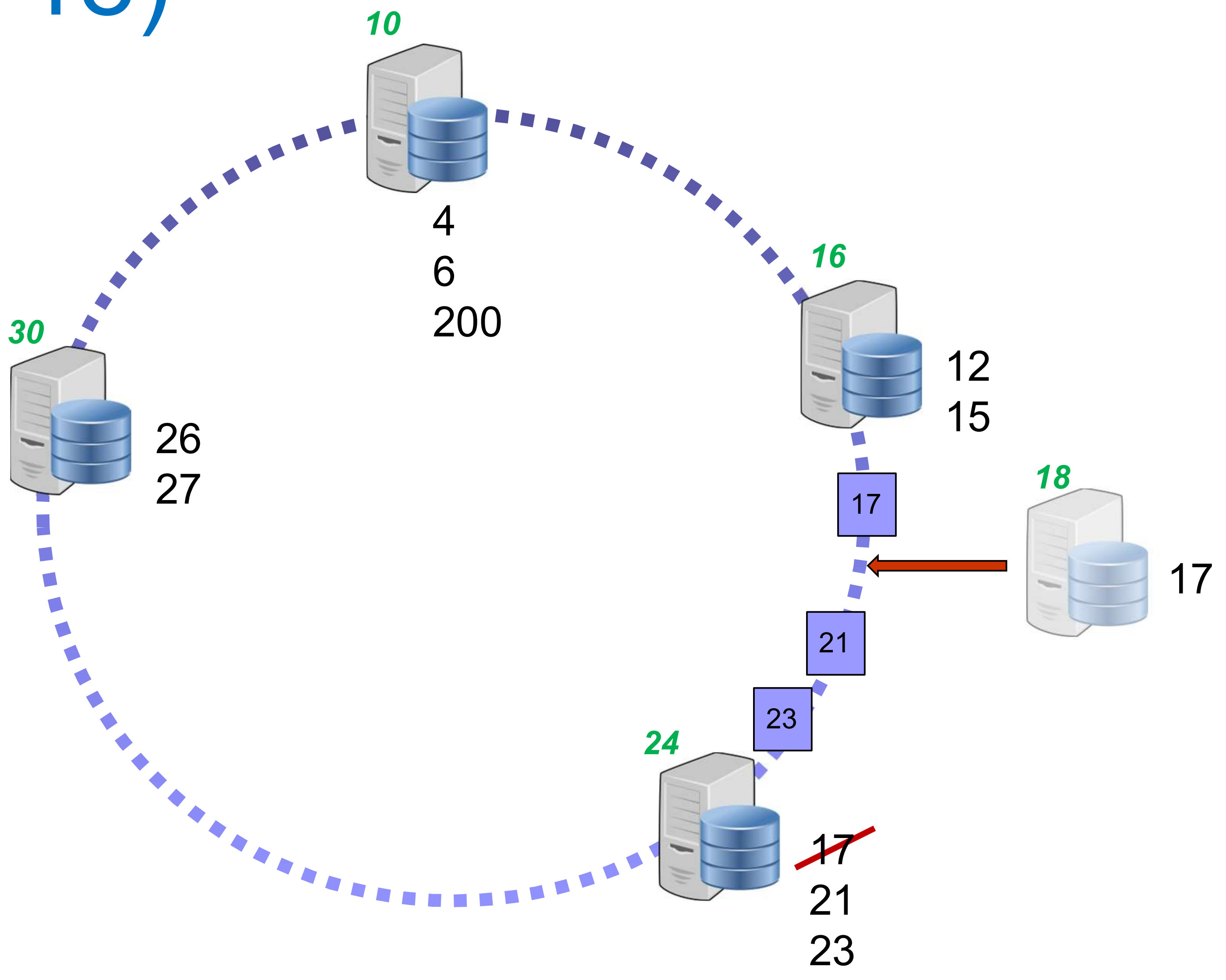
23

24

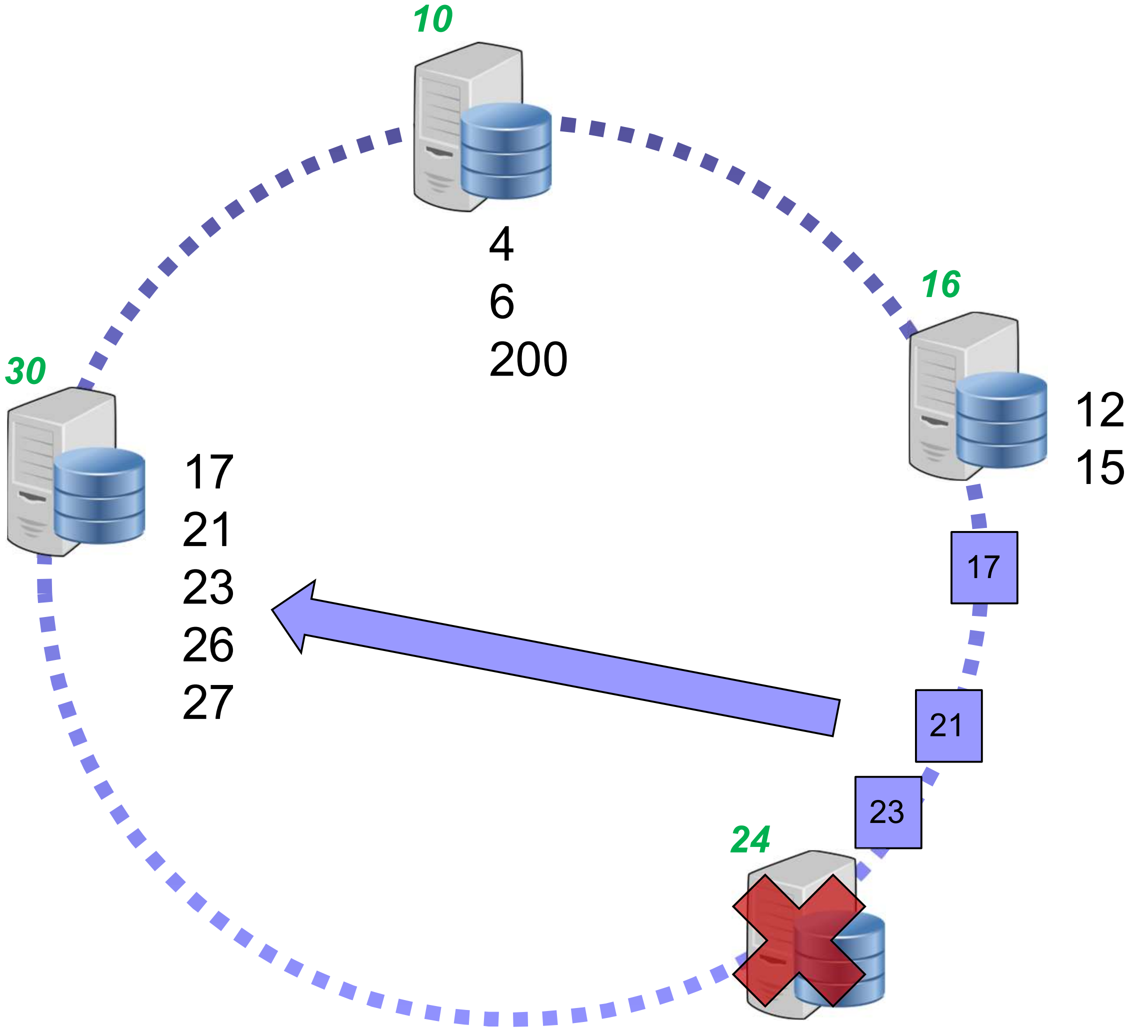
stores hash values
in range (16..24]

17
21
23

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



Removal of server 24



10

4
6
200

16

12
15

30

17
21
23
26
27

17
21
23

24

Scale-up + Scale out

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

Server 2 hashed twice