So far…

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
  - If the smaller relation fits in memory, then cost:
    - $b_r + b_s$
    - This is the best we can hope if we have to read the relations once each
  - CPU cost of the inner loop is high
- Index Nested-loops join
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - select balance from customer, accounts where customer.name = “j. s.” and customer.SSN = accounts.SSN

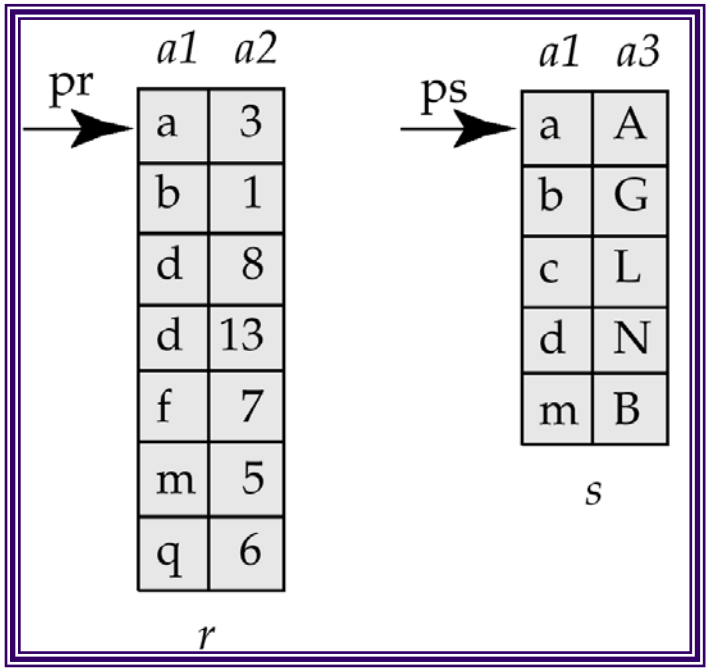
Merge-Join (Sort-merge join)

- Pre-condition:
  - equi-/natural joins
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this
- Called "sort-merge join" sometimes

*SELECT **
FROM r, s
WHERE r.a1 = s.a1

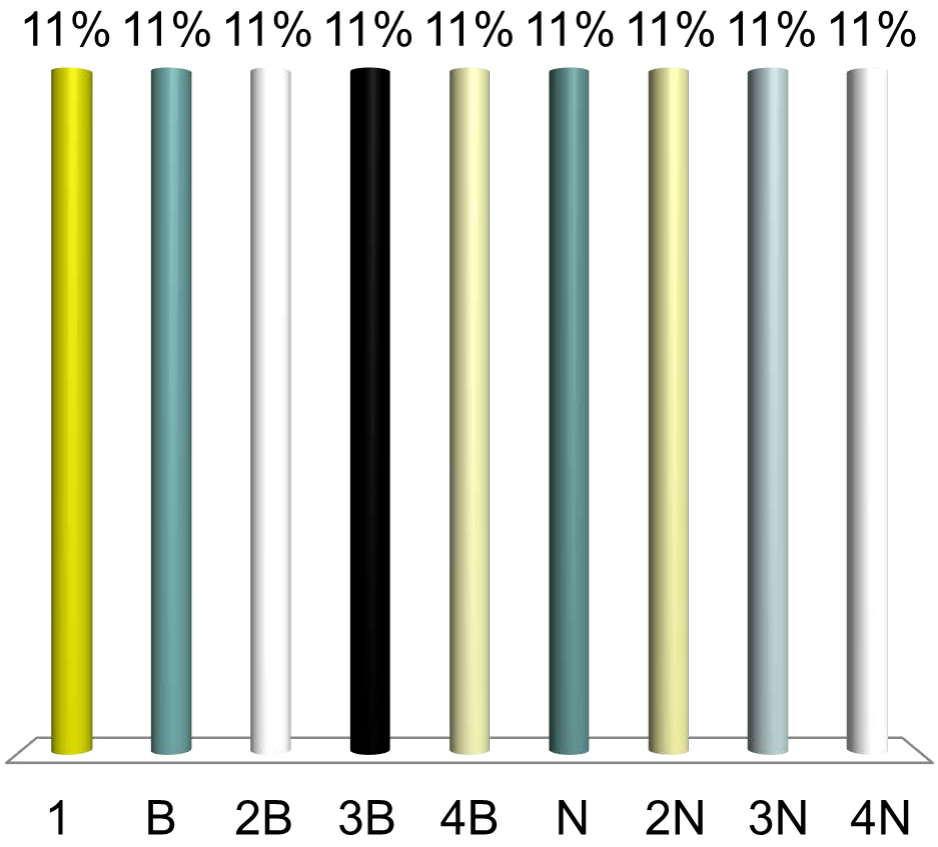
Step:

1. *Compare the tuples at pr and ps*
2. *Move pointers down the list*
   - *Depending on the join condition*
3. *Repeat*



How many blocks of R are read or written during merge join?
Assume R is B blocks and N tuples; already sorted by join attribute

A. 1
B. B
C. 2B
D. 3B
E. 4B
F. N
G. 2N
H. 3N
I. 4N

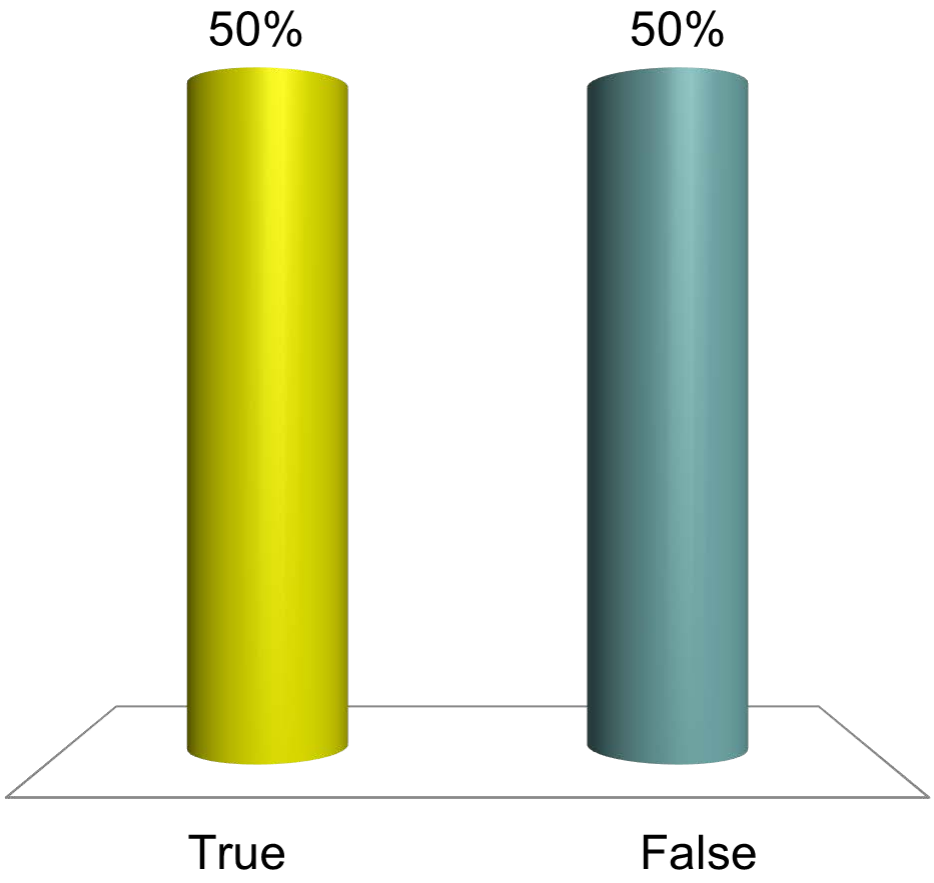


If R is B blocks and already sorted, it is impossible that more the B blocks of R will be read during merge join

A. True

B. False

May need to do a block nested loops join during the merge join if many tuples repeat the value of the join attribute. See textbook, page 555.



Merge-Join (Sort-merge join)

- Cost:
  - If the relations sorted, then just
    - $b_r + b_s$ block transfers, some seeks depending on memory size
  - What if not sorted ?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on *a1*
  - This might make further operations easier to do
    - E.g. duplicate elimination

So far…

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
- Index Nested-loops join
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - select balance from customer, accounts where customer.name = “j. s.” and customer.SSN = accounts.SSN
- Merge joins
  - Join algorithm of choice when the relations are large
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC

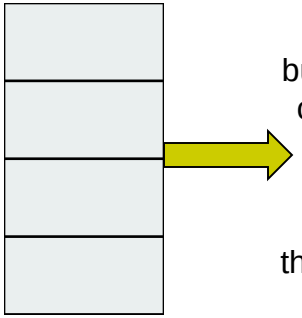
Hash Join

9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2

JOIN

23
32
15
7
12
2
9
1
31
16
10
21
25
4

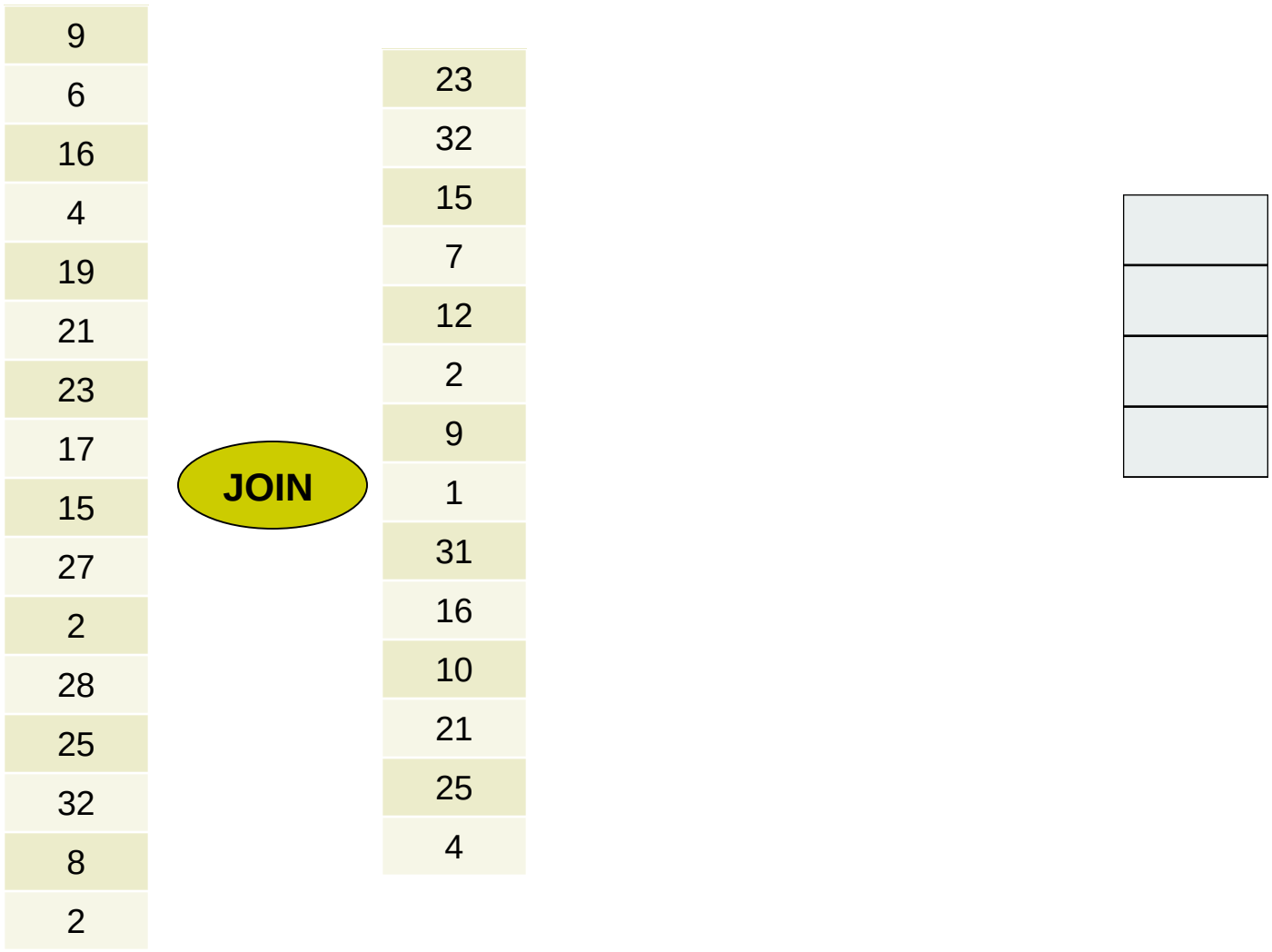
(Assume R is smaller table) Create sqrt(b_R) buckets in memory. Use hash function that hashes into this many buckets)



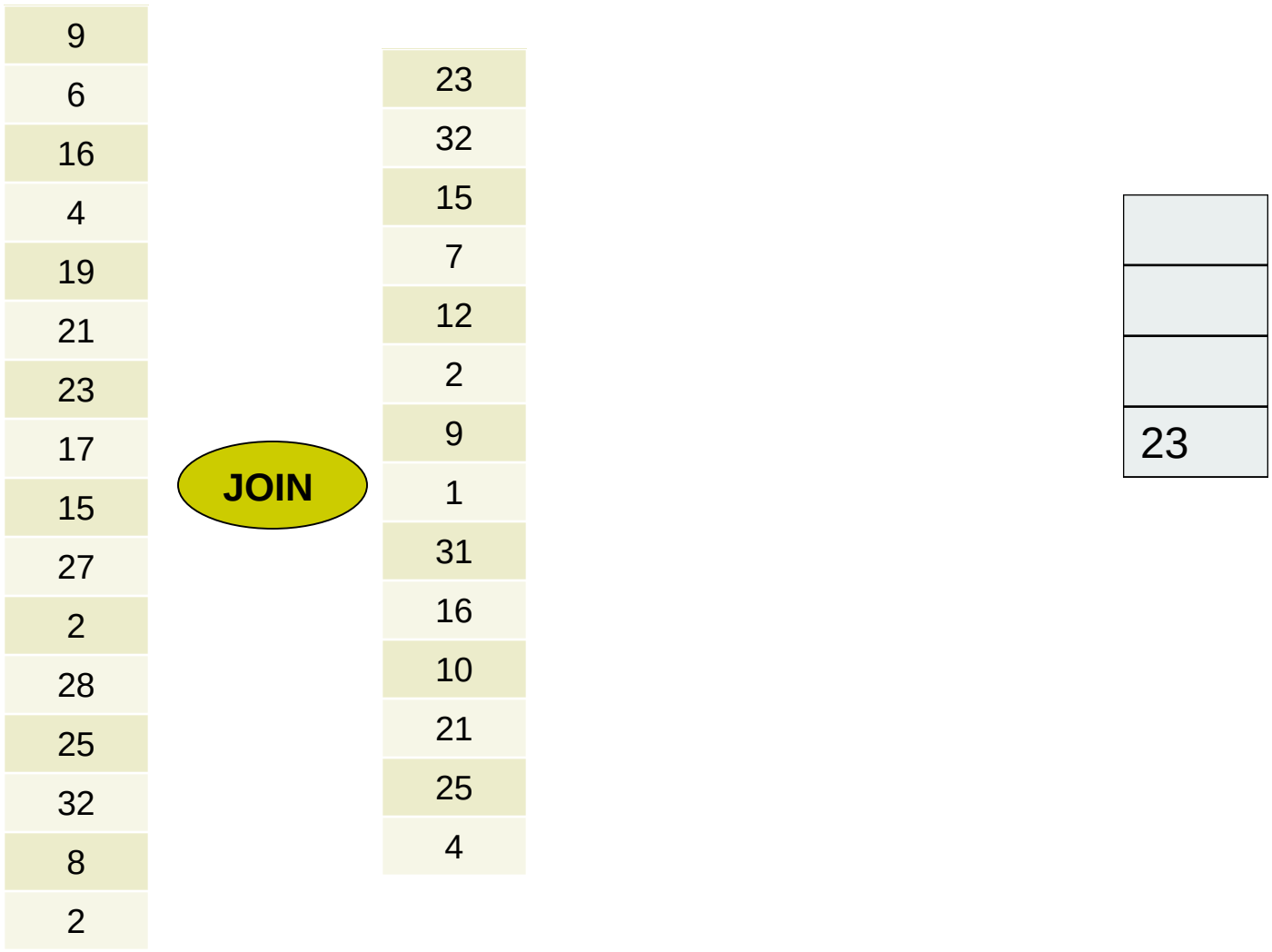
4 hash buckets, each can hold one block of values (one value in in this example).

Note: in this example, in order to fit on the slide, each block can only hold one tuple, so:
$b_R = n_R = 14$.

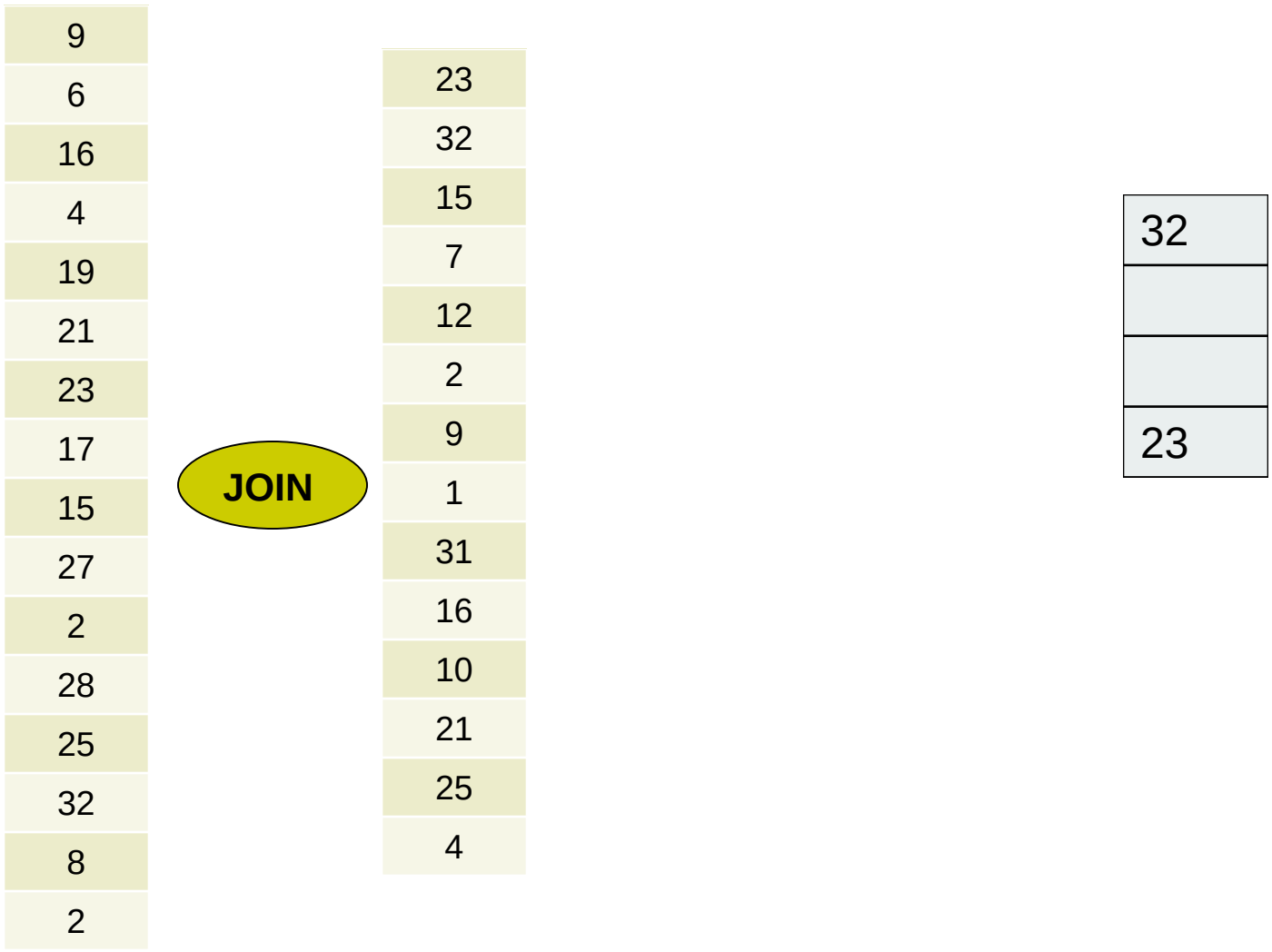
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4

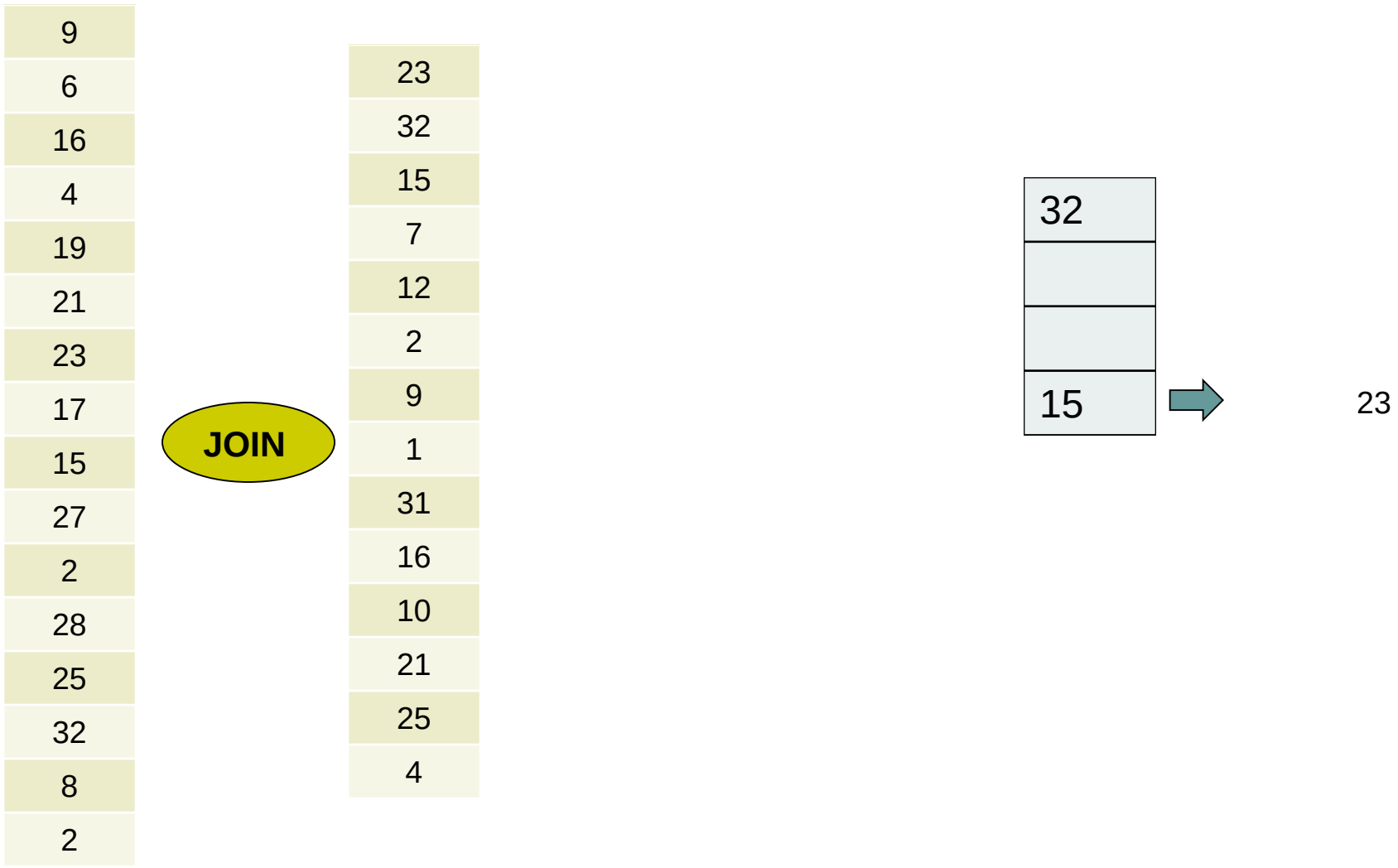
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
23

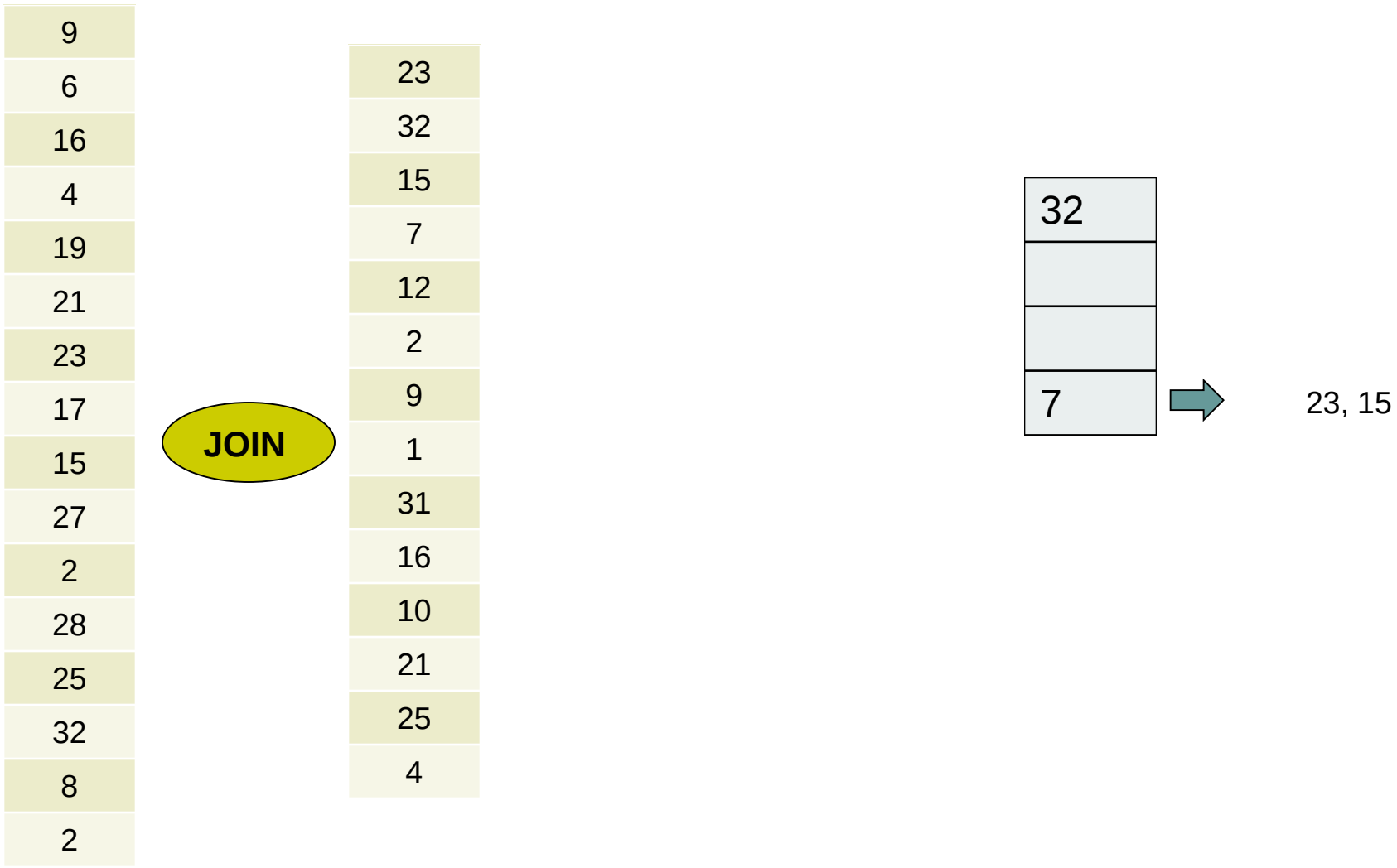
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
32
23

Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
32
15
23

Hash Join

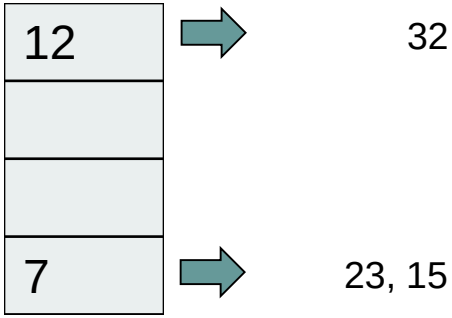

9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
32
7
23, 15

Hash Join

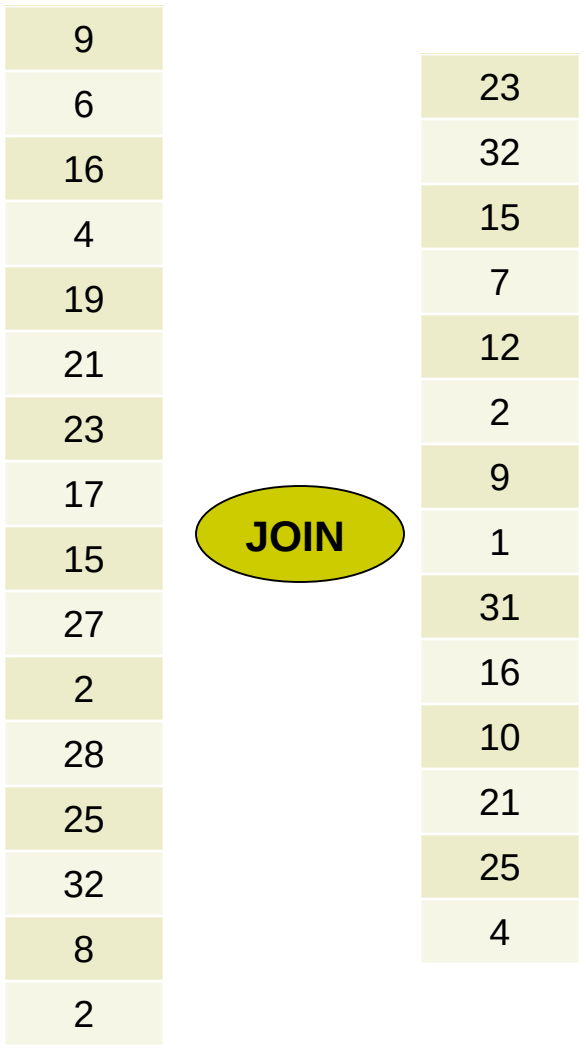
9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2

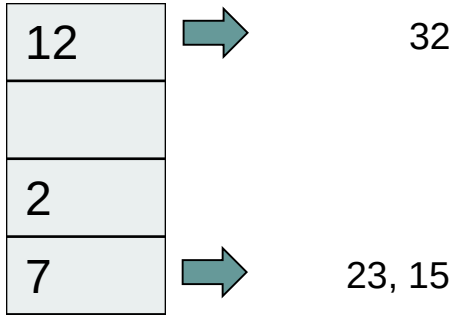
JOIN

23
32
15
7
12
2
9
1
31
16
10
21
25
4

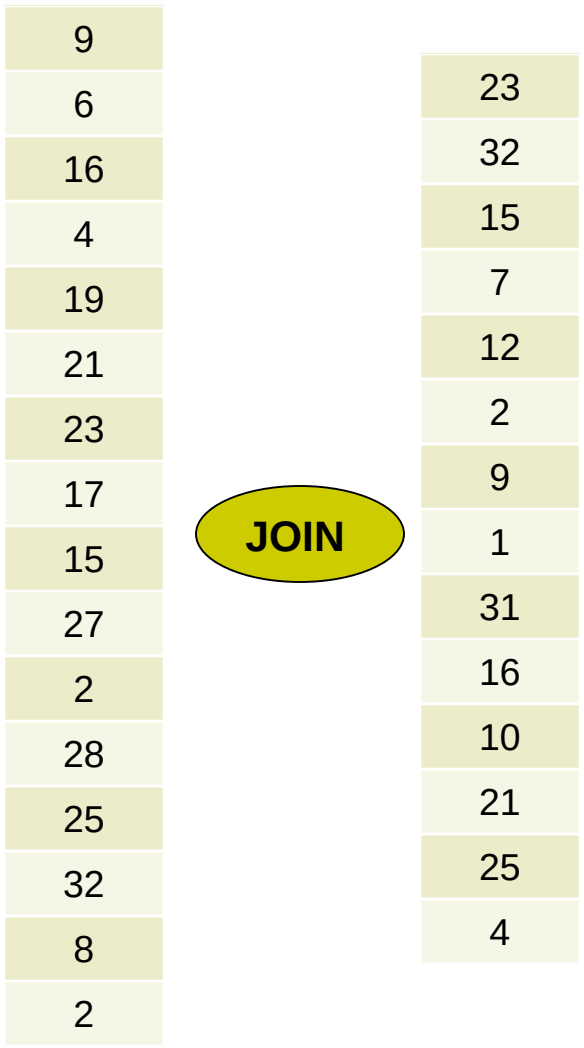


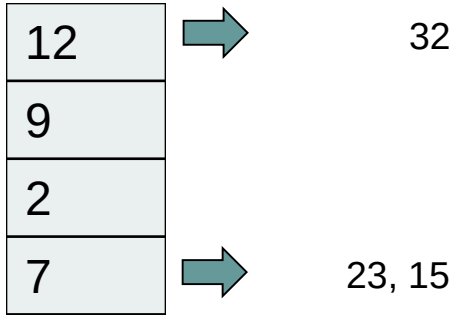
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4

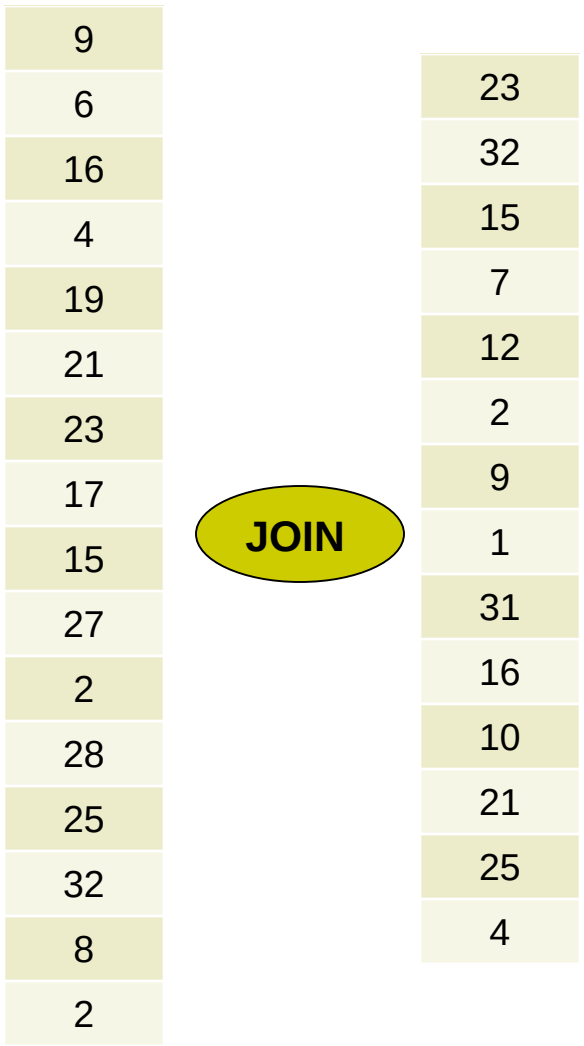

12
2
7
32
23, 15

Hash Join

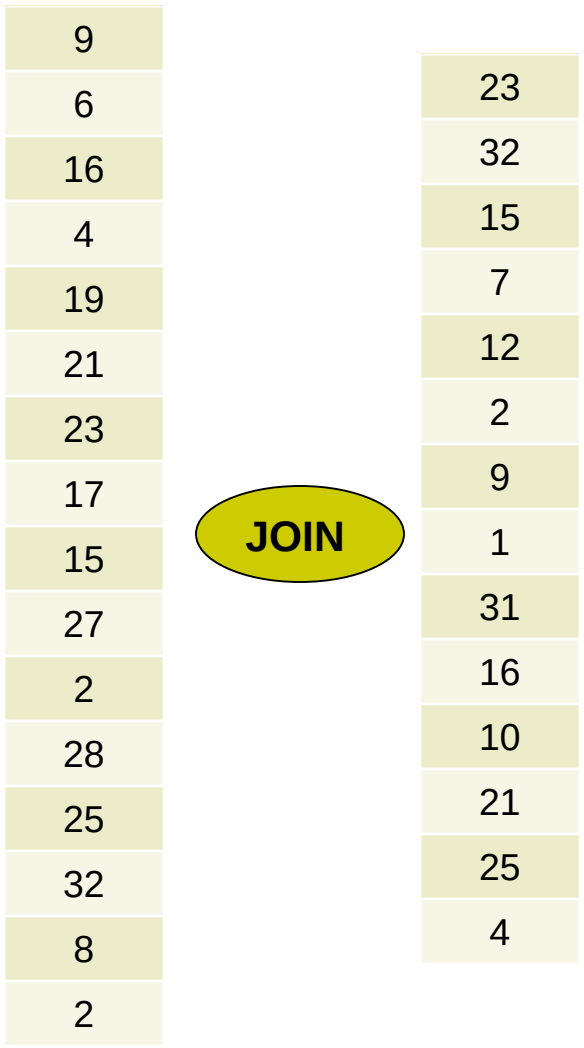

9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4

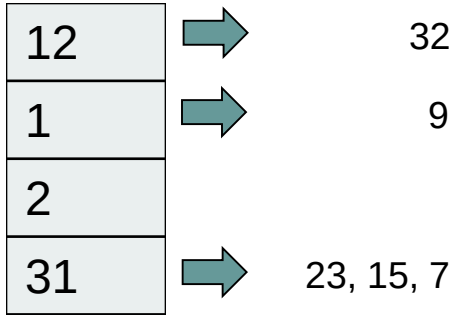

12
9
2
7
32
23, 15

Hash Join

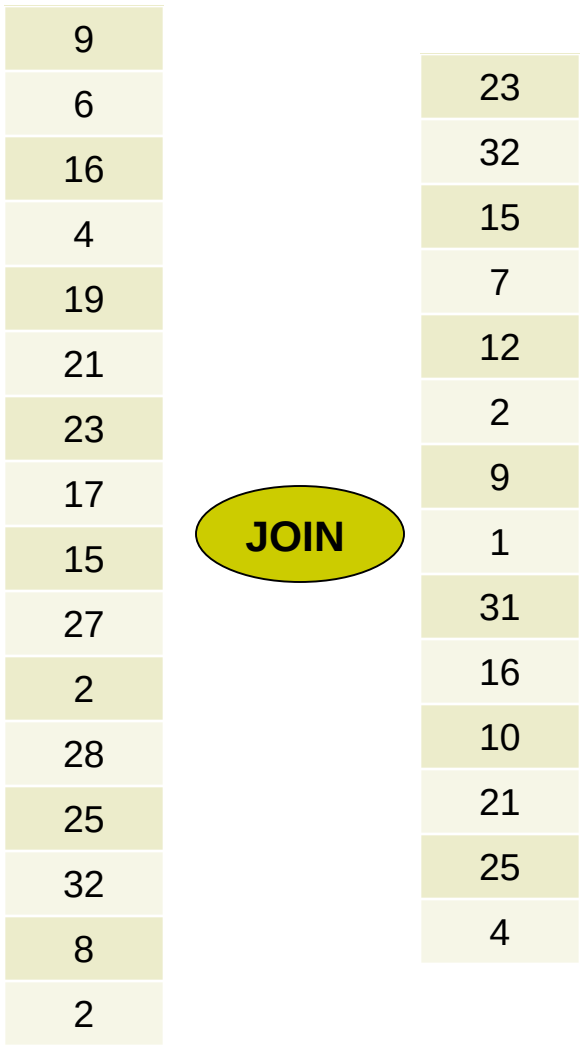

9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
12
1
2
7
32
9
23, 15

Hash Join

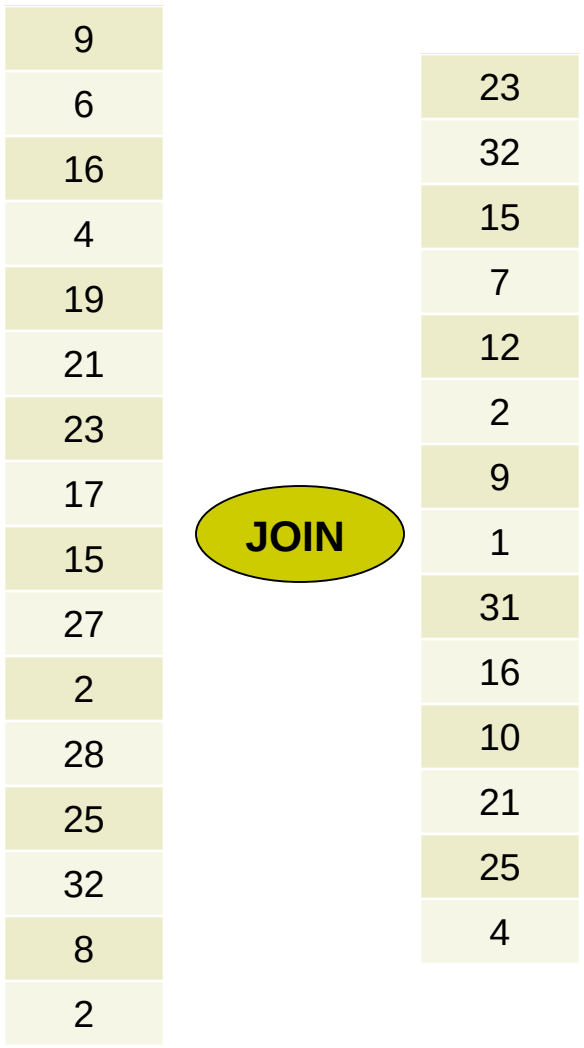

9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4

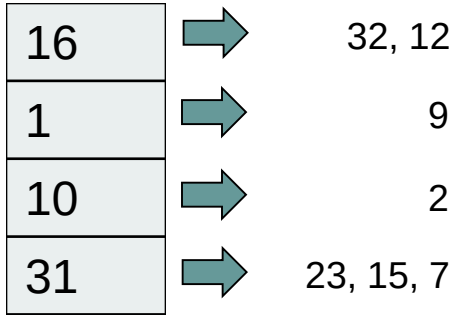

12
1
2
31
32
9
23, 15, 7

Hash Join

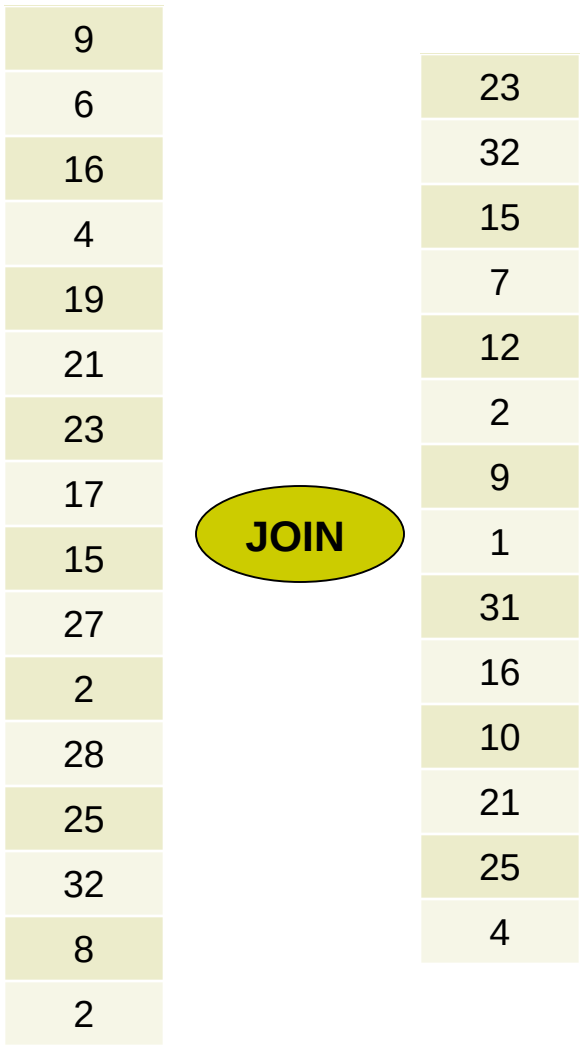

9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
16
1
2
31
32, 12
9
23, 15, 7

Hash Join

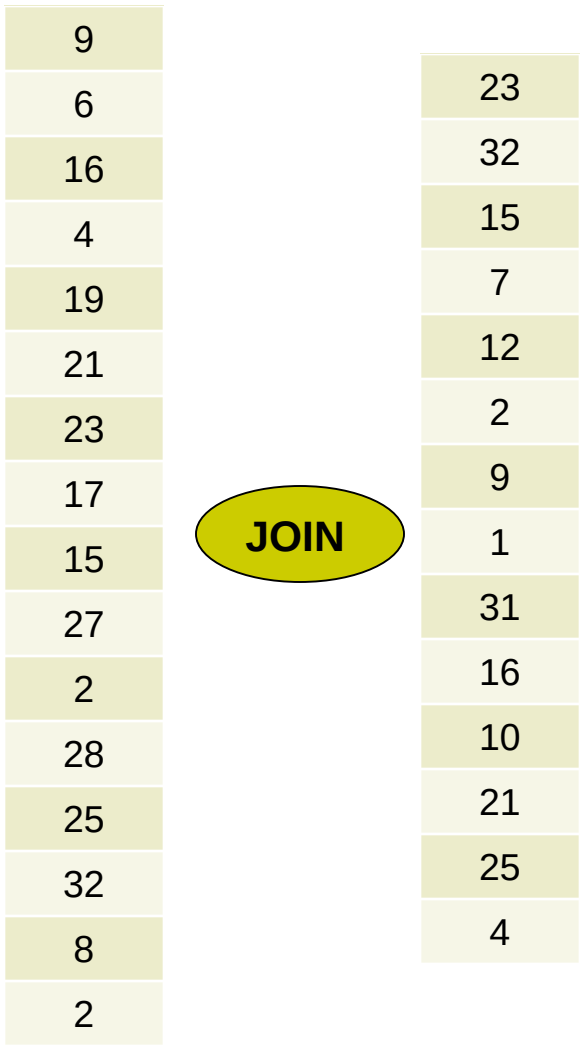

9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4


16
1
10
31
32, 12
9
2
23, 15, 7

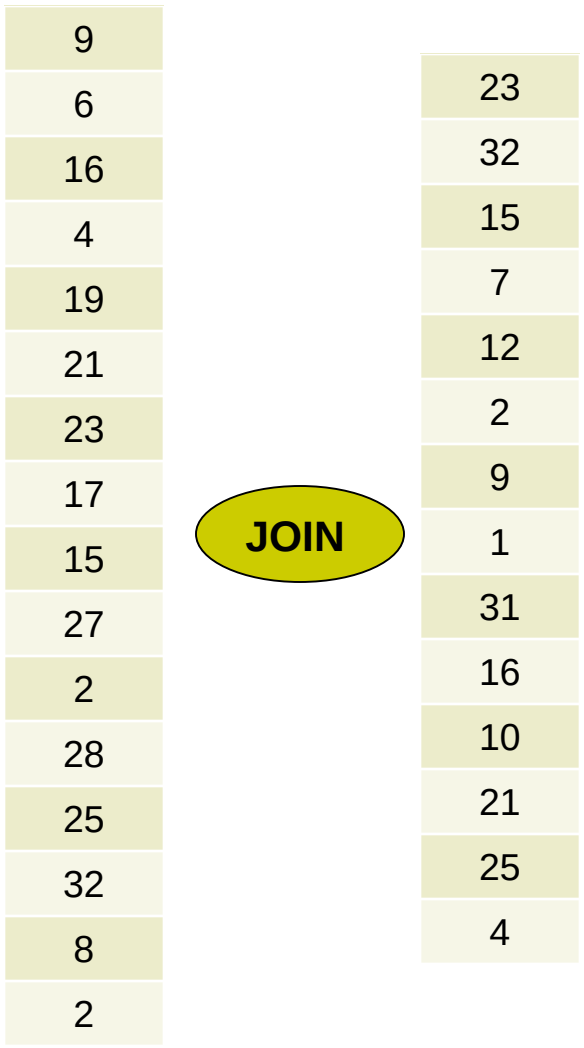
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
16
21
10
31
32, 12
9, 1
2
23, 15, 7

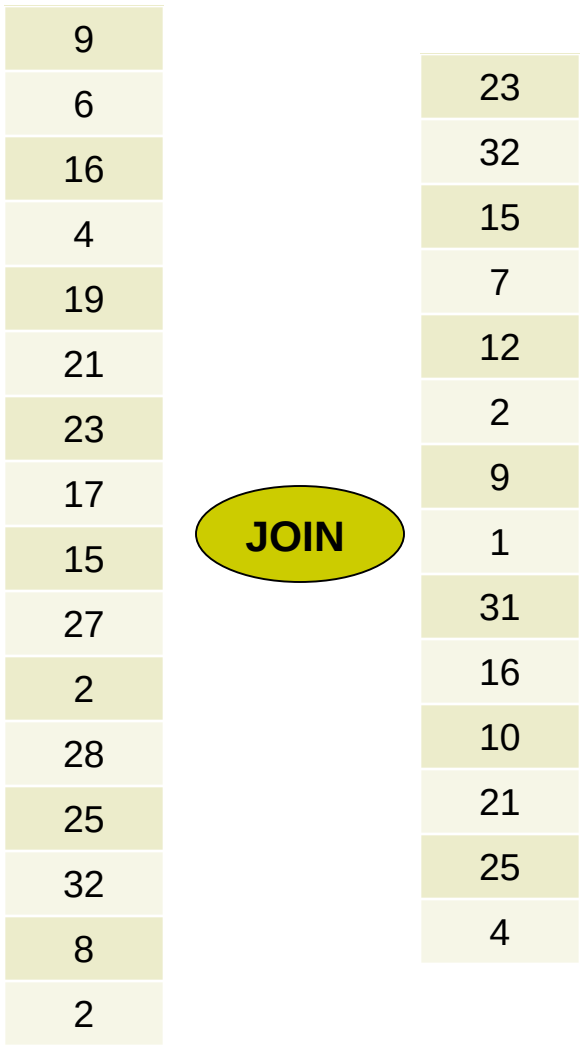
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
16
25
10
31
32, 12
9, 1, 21
2
23, 15, 7

Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
4
25
10
31
32, 12, 16
9, 1, 21
2
23, 15, 7

Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
32, 12, 16, 4
9, 1, 21, 25
2,10
23, 15, 7,31

Hash Join

9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2

JOIN

23
32
15
7
12
2
9
1
31
16
10
21
25
4

- 32, 12, 16, 4
- 9, 1, 21, 25
- 2,10
- 23, 15, 7,31

Hash Join

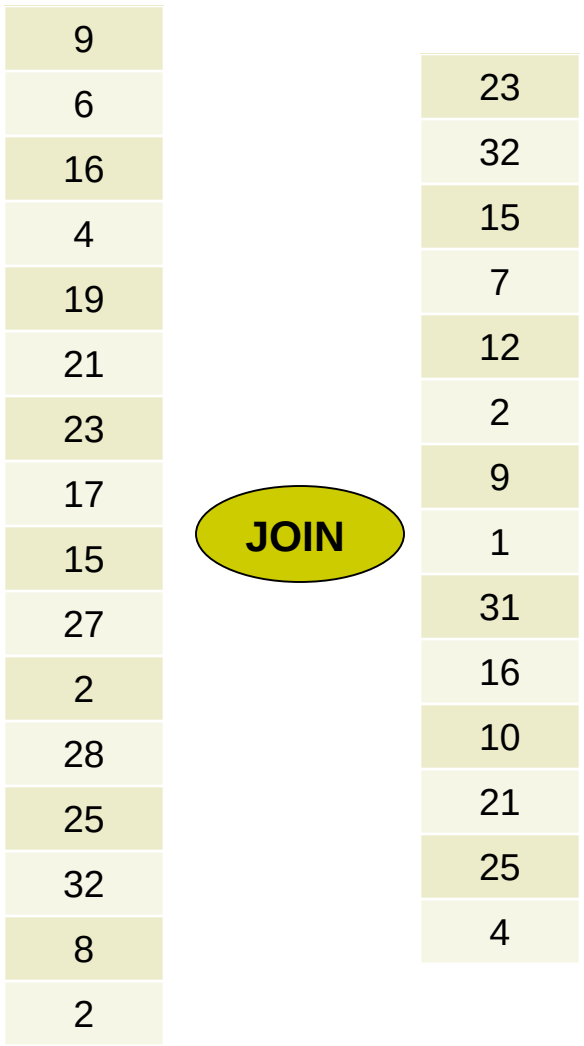
9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2

JOIN

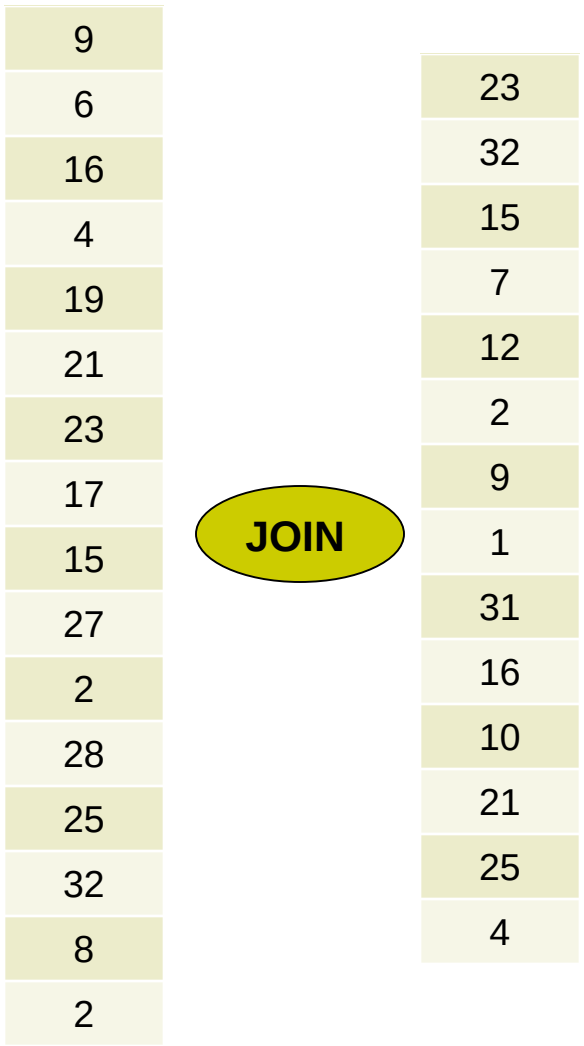
23
32
15
7
12
2
9
1
31
16
10
21
25
4

- 32, 12, 16, 4
- 9, 1, 21, 25
- 2,10
- 23, 15, 7,31

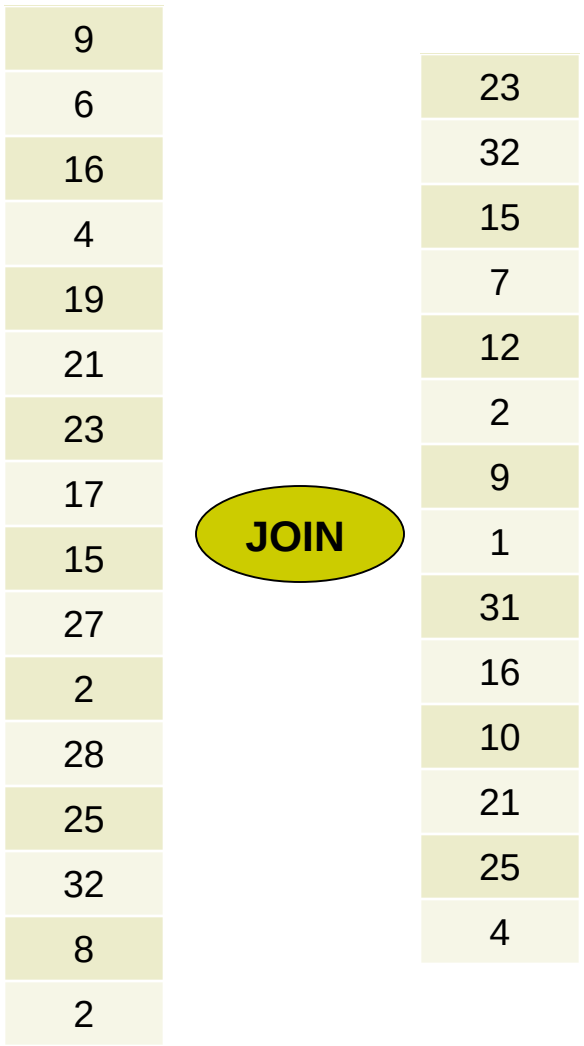
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
9
32, 12, 16, 4
9, 1, 21, 25
2,10
23, 15, 7,31

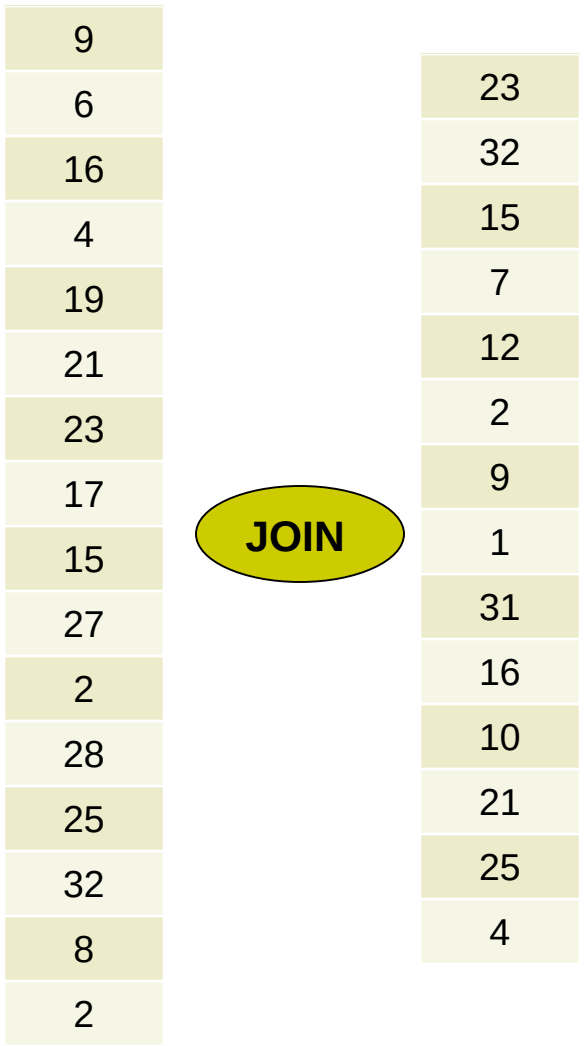
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
9
6
32, 12, 16, 4
9, 1, 21, 25
2,10
23, 15, 7,31

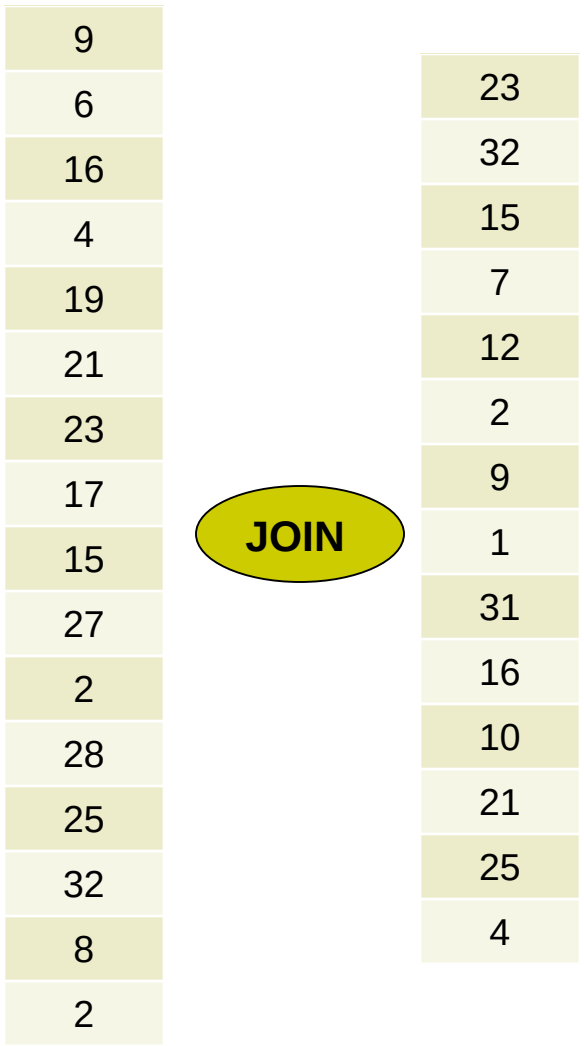
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
16
9
6
32, 12, 16, 4
9, 1, 21, 25
2,10
23, 15, 7,31

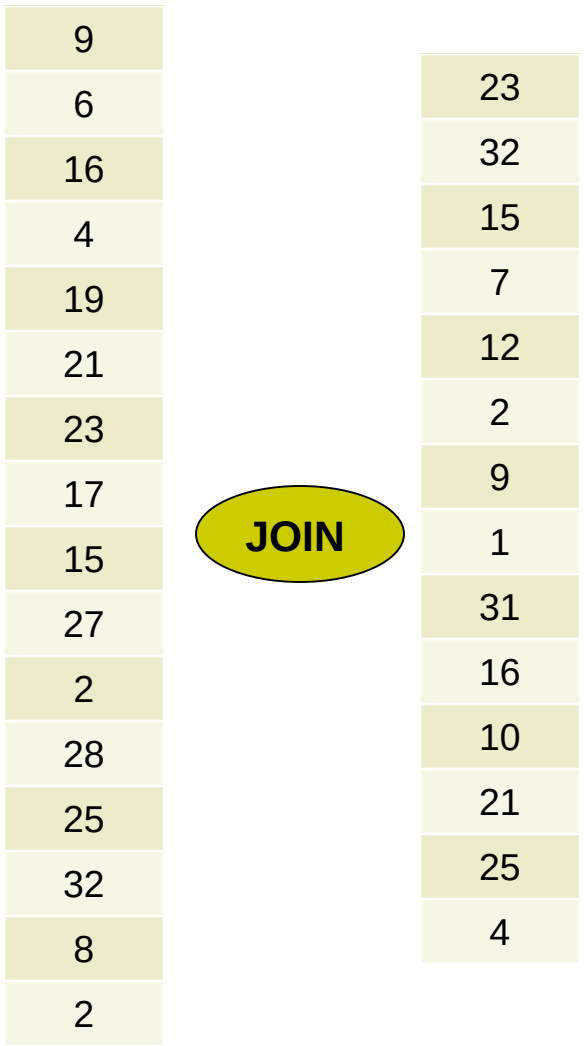
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
4
9
6
16
32, 12, 16, 4
9, 1, 21, 25
2,10
23, 15, 7,31

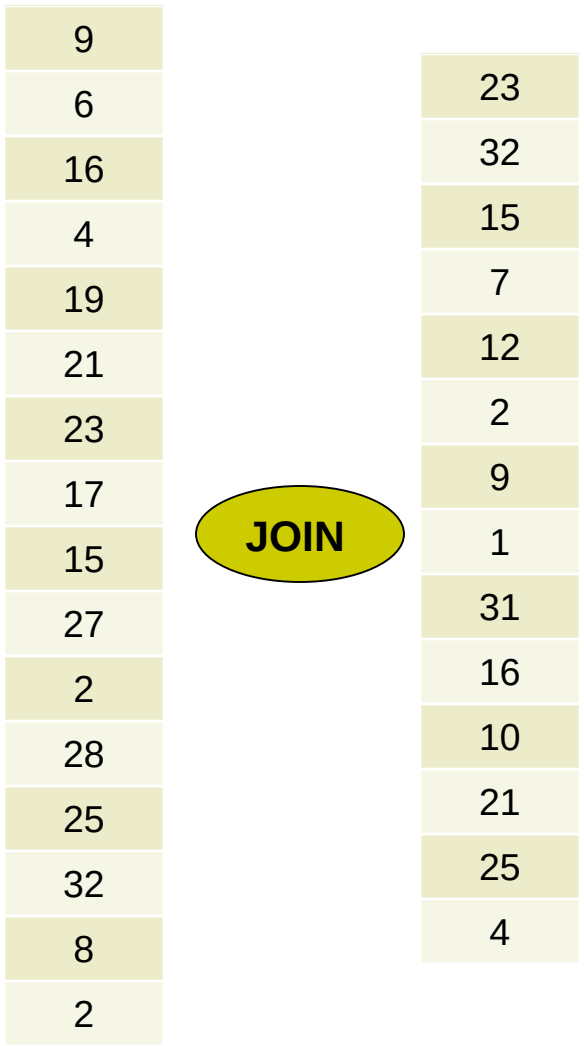
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
4
9
6
19
16
32, 12, 16, 4
9, 1, 21, 25
2,10
23, 15, 7,31

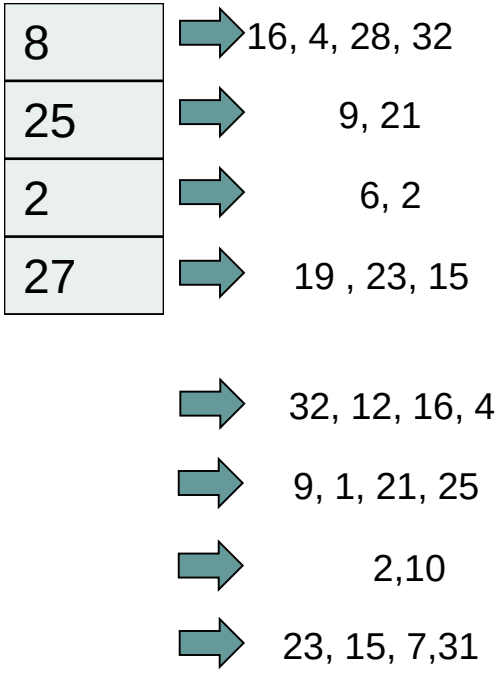
Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
4
21
6
19
16
9
32, 12, 16, 4
9, 1, 21, 25
2,10
23, 15, 7,31

Hash Join


9
6
16
4
19
21
23
17
15
27
2
28
25
32
8
2
JOIN
23
32
15
7
12
2
9
1
31
16
10
21
25
4
8
25
2
27
16, 4, 28, 32
9, 21
6, 2
19 , 23, 15
32, 12, 16, 4
9, 1, 21, 25
2,10
23, 15, 7,31

Hash Join



Hash Join

16
4
28
32
8

9
21
25

6
2
2

19
23
15
27

32, 12, 16, 4

9, 1, 21, 25

2,10

23, 15, 7,31

Hash Join

16
4
28
32
8

9
21
25

6
2
2

19
23
15
27

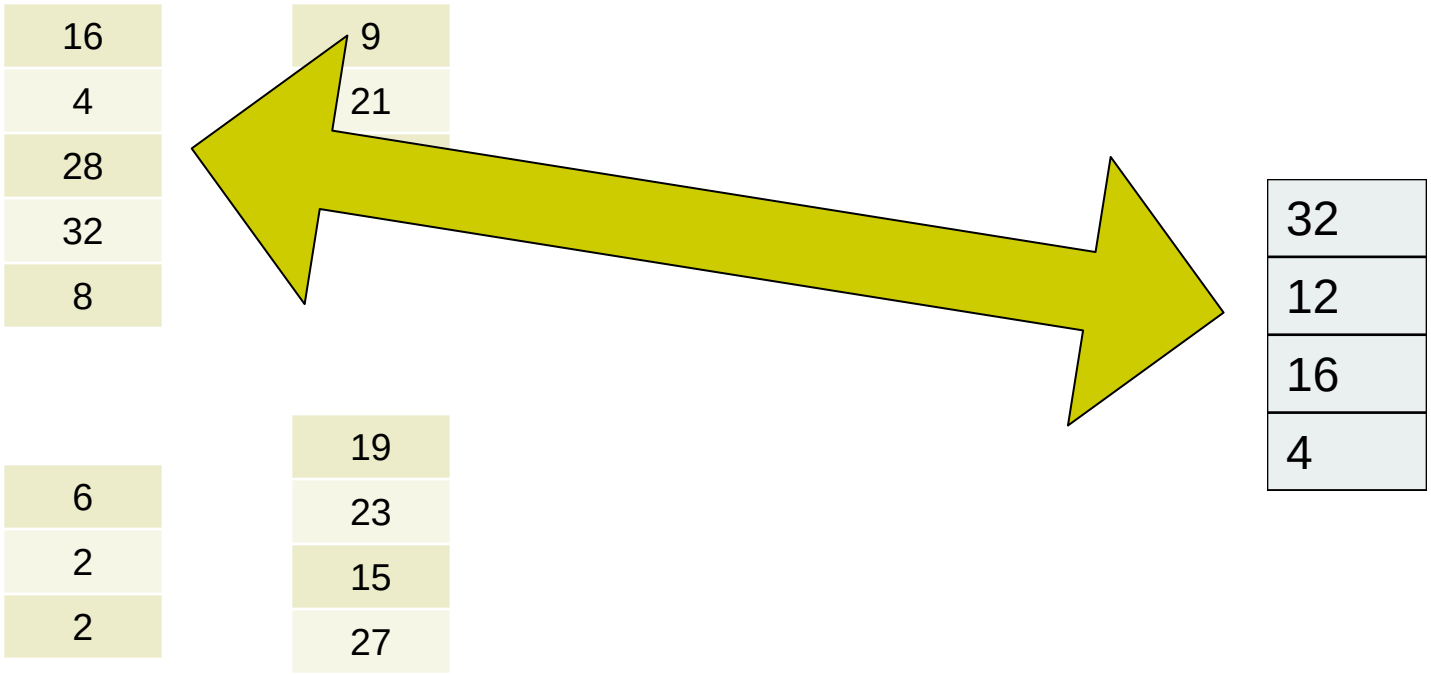
32
12
16
4


9, 1, 21, 25


2,10

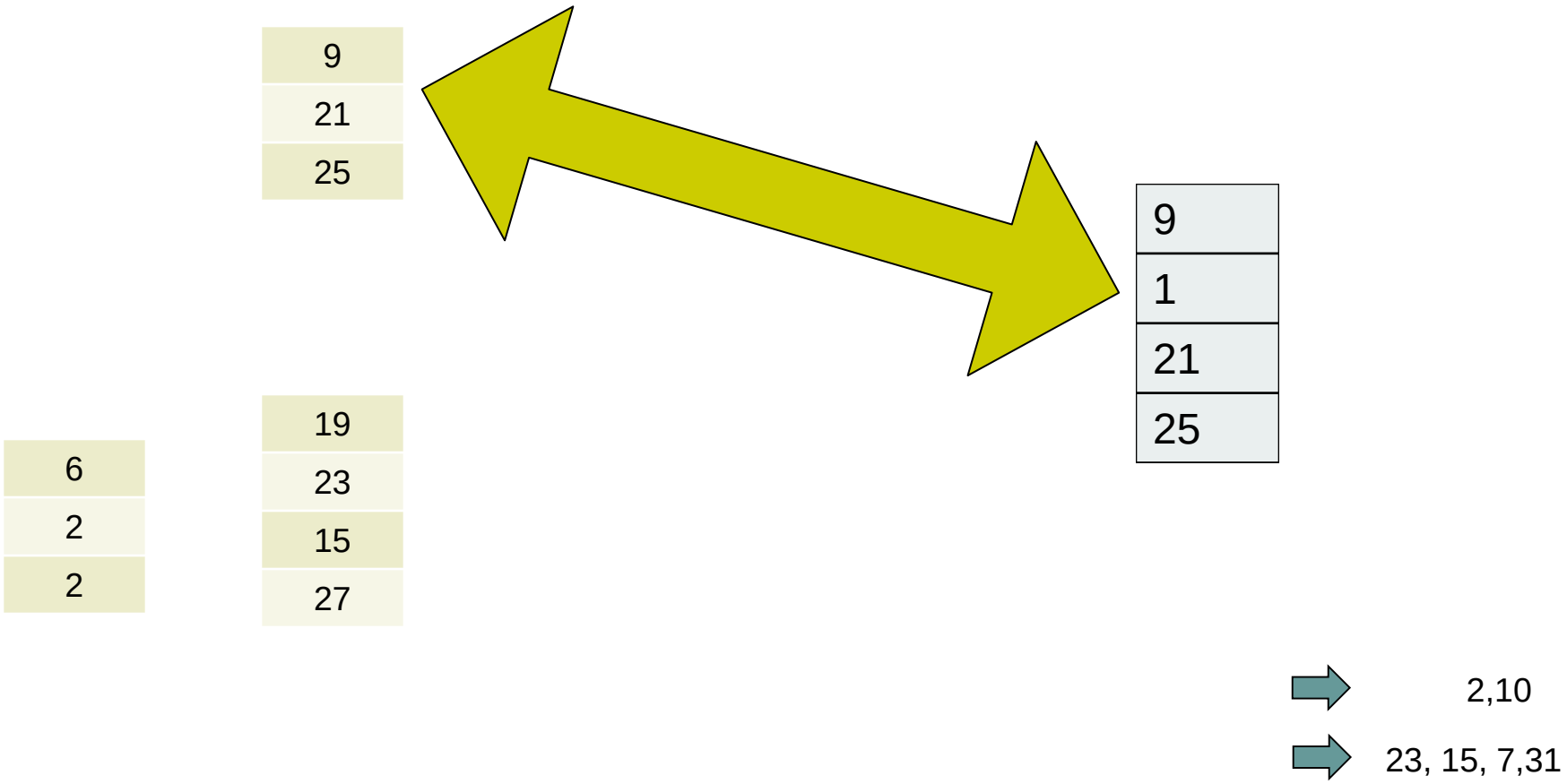

23, 15, 7,31

Hash Join



Hash Join

Hash Join



Hash Join

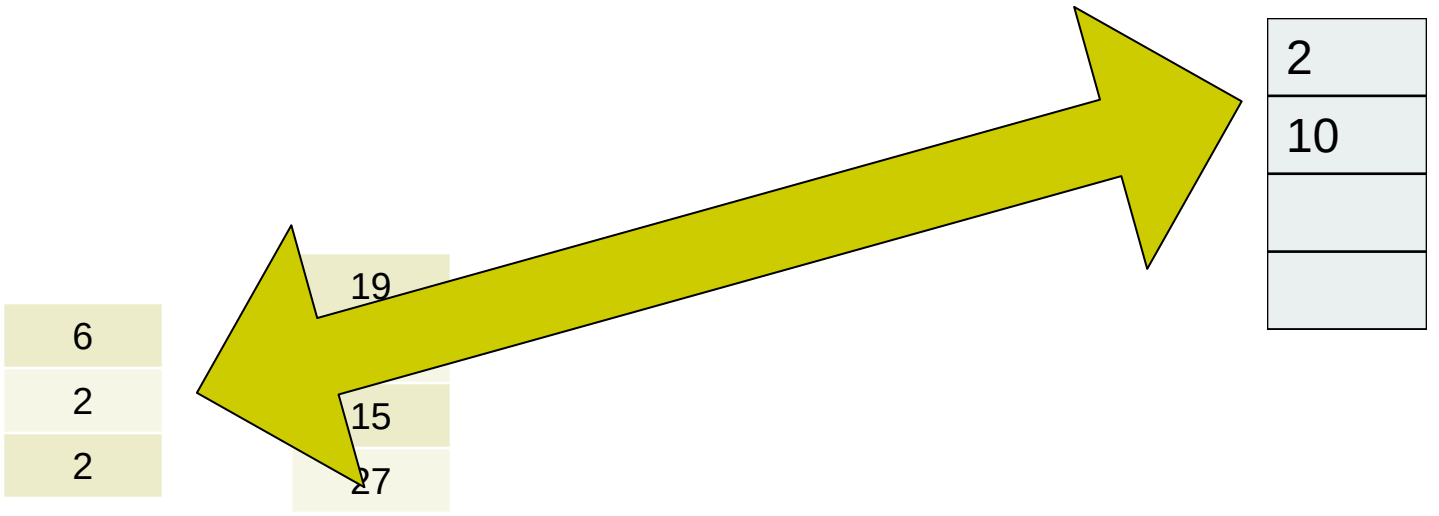
6
2
2

19
23
15
27

2,10

23, 15, 7,31

Hash Join



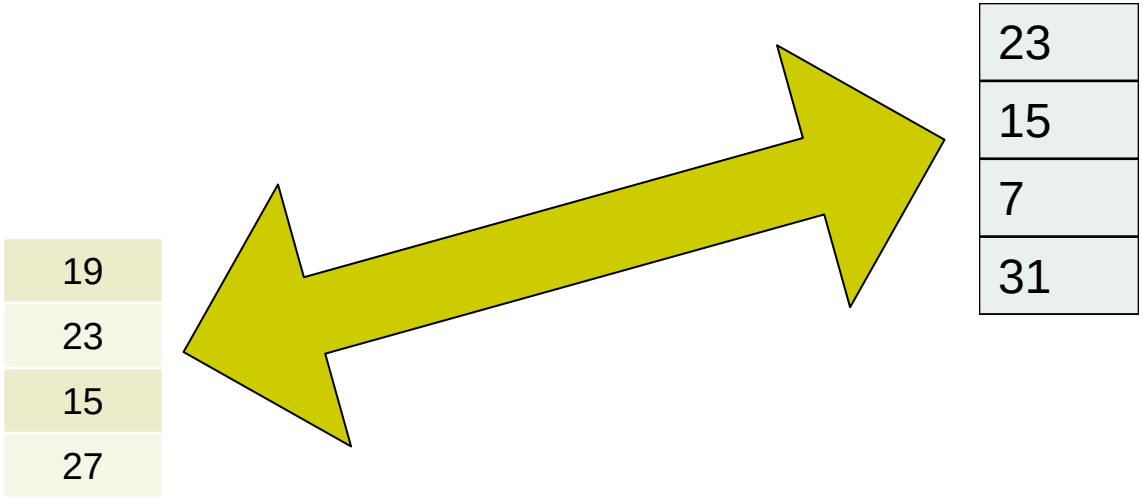
23, 15, 7,31

Hash Join

19
23
15
27

23, 15, 7,31

Hash Join



If there are sqrt(b_R) hash buckets, what is avg size of each partition of R?

A. 1

B. sqrt(b_R)

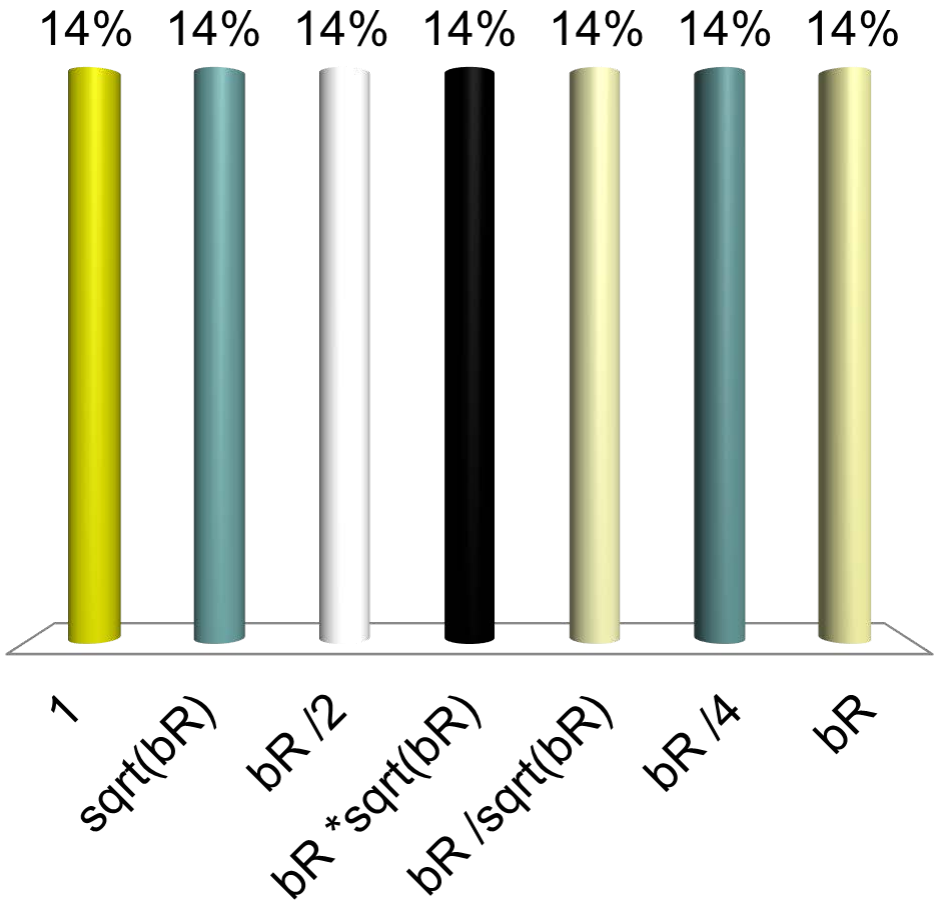
C. b_R /2

D. b_R *sqrt(b_R)

E. b_R /sqrt(b_R)

F. b_R /4

G. b_R



Hash Join

- First phase requires sqrt(b_R) blocks of memory
- Second phase requires sqrt(b_R) blocks of memory only if EVERY bucket has average number of tuples
  - Very unlikely
    - In practice, allocate sqrt(F* b_R) blocks of memory, F is fudge factor, usually around 1.2
- So if M is >= sqrt(F * b_R), both phases work (i.e. don't need to spill to disk)

Hash Join Cost

- Every block of bigger table (S) gets read, then written, then read
  - Total of 3 * b_S
- Every block of smaller table (R) gets read, then written, then read
  - Total of 3 * b_R
- Seeks hard to calculate for first set of reads/writes. Assume worst case: 2 * (b_S + b_R)
- One seek for each S partition and one for each R partition for last set of reads