

Part 4: Join Algorithms

Joins

- ❖ Three main kinds (our focus)
 - ❖ cross product (or cross joins)
 - ❖ equality joins
 - ❖ inequality joins
- ❖ Several special joins
 - ❖ outer joins
 - ❖ joins to handle special situations
 - ❖ correlated subqueries, exist construct, etc.

The Nested Loop Join

- ❖ The fallback algorithm, analogous to scanning the table for WHERE condition
- ❖ Must have “tool” in the join algorithm tool box

Block Nested Loop

- ❖ Let's say we are joining relation R and C: $R \bowtie S$
- ❖ For Each block, B_R of R
 - ❖ read the block B_R
 - ❖ For Each block, B_S of S
 - ❖ read the block B_S
 - ❖ for each row r in B_R
 - ❖ for each row s in B_S
 - ❖ if r joins with s then put $r \bowtie s$ in output buffer
 - ❖ if (output buffer full) then flush it

R \bowtie S

R

[28,	7,	1,	13]
[7,	25,	24,	17]
[19,	6,	22,	4]
[8,	4,	20,	16]
[18,	19,	11,	29]

S

[26,	20,	8,	8]
[3,	22,	1,	23]
[15,	12,	17,	13]
[10,	11,	5,	1]
[0,	21,	15,	16]
[24,	14,	14,	16]
[3,	17,	11,	17]
[7,	10,	29,	3]

R ⋈ S

R

[28, 7, 1, 13]
[7, 25, 24, 17]
[19, 6, 22, 4]
[8, 4, 20, 16]
[18, 19, 11, 29]

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

❖ In Memory

❖ Bring a block of R

[28, 7, 1, 13]

❖ In Memory

❖ Bring a block of S

[26, 20, 8, 8]

❖ Join the two blocks of rows

❖ No matches (in this case)

❖ Bring next block of S

[3, 22, 1, 23]

❖ Join the two blocks of rows

❖ [1,1] is a match, output

❖ Bring next block of S

[15, 12, 17, 13]

❖ Join the two blocks of rows

❖ [13,13] is a match, output

❖ Bring next block of S

❖ Output rows

[(1,1)]

[(1,1),
(13,13)]

R ⋈ S

R

❖ In Memory

❖ Bring a block of R

[28, 7, 1, 13]
[7, 25, 24, 17]
[19, 6, 22, 4]
[8, 4, 20, 16]
[18, 19, 11, 29]

[28, 7, 1, 13]

❖ In Memory

❖ Bring blocks of S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]

[10, 11, 5, 1]

❖ Output rows

[(1,1)]

[(1,1), (13,13)]

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

[(1,1), (13,13),
(1,1), (7,7)]

[7, 25, 24, 17]

R ⋈ S

R

❖ In Memory

[28, 7, 1, 13]
[7, 25, 24, 17]
[19, 6, 22, 4]
[8, 4, 20, 16]
[18, 19, 11, 29]

❖ Bring next block of R

[7, 25, 24, 17]

❖ In Memory

❖ Bring blocks of S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]

❖ Output rows

[(1,1),(13,13),(1,1),(7,7)]
[(17,17)]

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]

[3, 17, 11, 17]

[7, 10, 29, 3]

[(1,1),(13,13),(1,1),(7,7)]
[(17,17),(24,24)]

[(1,1),(13,13),(1,1),(7,7)]
[(17,17),(24,24),
(17,17),(17,17)]

[(1,1),(13,13),(1,1),(7,7)]
[(17,17),(24,24),(17,17),
(17,17)]

[(7,7)]

I/O costs?

- ❖ How many times did we read relation R?
- ❖ How many times did we read relation S?

R ⋈ S

R	❖ In Memory ❖ Bring a block of R	❖ In Memory ❖ Bring blocks of S	❖ Output rows
[28, 7, 1, 13]	[28, 7, 1, 13]	[26, 20, 8, 8]	
[7, 25, 24, 17]	[7, 25, 24, 17]	[3, 22, 1, 23]	[(1,1)]
[19, 6, 22, 4]		[15, 12, 17, 13]	[(1,1), (13,13), (17,17)]
[8, 4, 20, 16]		[10, 11, 5, 1]	[(1,1), (13,13), (17,17), (1,1)]
[18, 19, 11, 29]			
		[0, 21, 15, 16]	[(1,1), (13,13), (17,17), (1,1)]
		[24, 14, 14, 16]	[(24,24)]
		[3, 17, 11, 17]	[(1,1), (13,13), (17,17), (1,1)]
			[(24,24), (17,17), (17,17)]
		[7, 10, 29, 3]	[(1,1), (13,13), (17,17), (1,1)]
			[(24,24), (17,17), (17,17), (7,7)]
			[(7,7)]
S			
[26, 20, 8, 8]			
[3, 22, 1, 23]			
[15, 12, 17, 13]			
[10, 11, 5, 1]			
[0, 21, 15, 16]			
[24, 14, 14, 16]			
[3, 17, 11, 17]			
[7, 10, 29, 3]			

Block Nested Loop - Cost

- ❖ Let $|R|$, $|S|$ be number of pages in R , S respectively
- ❖ If fraction f of R fits in memory, we need to fetch S , $1/f$ times
 - ❖ if all of R fits in memory then we only need to read both just once!
- ❖ Which relation should be outer?
- ❖ Still a lot of CPU cost

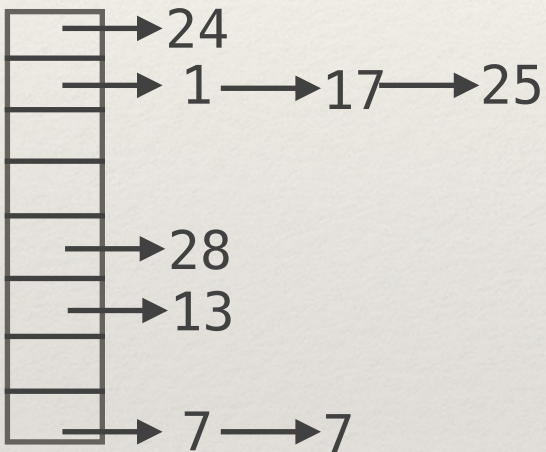
Equality=Inner=Natural Join

- ❖ Most joins (e.g. PK-FK) are equality joins
- ❖ Several algorithms to handle the equality join

Hash Joins

- ❖ In block nested loop join, we could build a hash table on the outer table for fast equality search
- ❖ Still same amount of I/O = $|R| + |S|/f$
- ❖ But lot less CPU processing
- ❖ Can we do better?

R ⋈ S using Hash Join

R	❖ In Memory ❖ Bring a block of R	❖ In Memory ❖ Bring blocks of S	❖ Output rows
[28, 7, 1, 13]	[28, 7, 1, 13]	[26, 20, 8, 8]	
[7, 25, 24, 17]	[7, 25, 24, 17]	[3, 22, 1, 23]	[(1,1)]
[19, 6, 22, 4]		[15, 12, 17, 13]	[(1,1), (13,13), (17,17)]
[8, 4, 20, 16]		[10, 11, 5, 1]	[(1,1), (13,13), (17,17), (1,1)]
[18, 19, 11, 29]		[0, 21, 15, 16]	[(1,1), (13,13), (17,17), (1,1)]
		[24, 14, 14, 16]	[(24,24)]
		[3, 17, 11, 17]	[(1,1), (13,13), (17,17), (1,1)]
			[(24,24), (17,17), (17,17)]
		[7, 10, 29, 3]	[(1,1), (13,13), (17,17), (1,1)]
			[(24,24), (17,17), (17,17), (7,7)]
			[(7,7)]
S			
[26, 20, 8, 8]			
[3, 22, 1, 23]			
[15, 12, 17, 13]			
[10, 11, 5, 1]			
[0, 21, 15, 16]			
[24, 14, 14, 16]			
[3, 17, 11, 17]			
[7, 10, 29, 3]			

Hash Join: Grace

- ❖ We can partition both R and S using hashing so that each partition of R would fit in memory
- ❖ Two I/O (read+write) of R & S for partitioning
- ❖ One scan of R & S for joining
- ❖ Make partition size small enough to minimize memory overflow issues

Hash Partitioning

S

[26, 20, 8, 8]

[3, 22, 1, 23]

[15, 12, 17, 13]

[10, 11, 5, 1]

[0, 21, 15, 16]

[24, 14, 14, 16]

[3, 17, 11, 17]

[7, 10, 29, 3]

B0: []

B1: []

B2: []

B3: []

B4: []

B5: []

B6: []

B7: []

Hash Partitioning

S

[26, 20, 8, 8]

[3, 22, 1, 23]

[15, 12, 17, 13]

[10, 11, 5, 1]

[0, 21, 15, 16]

[24, 14, 14, 16]

[3, 17, 11, 17]

[7, 10, 29, 3]

B0: []

B1: []

B2: [26]

B3: []

B4: []

B5: []

B6: []

B7: []

Hash Partitioning

S

[26, 20, 8, 8]

[3, 22, 1, 23]

[15, 12, 17, 13]

[10, 11, 5, 1]

[0, 21, 15, 16]

[24, 14, 14, 16]

[3, 17, 11, 17]

[7, 10, 29, 3]

B0: []

B1: []

B2: [26]

B3: []

B4: [20]

B5: []

B6: []

B7: []

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [8]
B1: []
B2: [26]
B3: []
B4: [20]
B5: []
B6: []
B7: []

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [8, 8]
B1: []
B2: [26]
B3: []
B4: [20]
B5: []
B6: []
B7: []

Hash Partitioning

S	mod 8
[26, 20, 8, 8]	B0: [8, 8]
[3, 22, 1, 23]	B1: []
[15, 12, 17, 13]	B2: [26]
[10, 11, 5, 1]	B3: [3]
[0, 21, 15, 16]	B4: [20]
[24, 14, 14, 16]	B5: []
[3, 17, 11, 17]	B6: []
[7, 10, 29, 3]	B7: []

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [8, 8]
B1: []
B2: [26]
B3: [3]
B4: [20]
B5: []
B6: [22]
B7: []

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, **16**]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [8, 8, 0, 16]
B1: [1, 17, 1]
B2: [26, 10]
B3: [3, 11]
B4: [20, 12]
B5: [13, 5, 21]
B6: [22]
B7: [23, 15, 15]

- ❖ Assuming page size = 4
- ❖ B0 is full when we process 16
- ❖ B0 is written to disk

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [24]
B1: [1, 17, 1]
B2: [26, 10]
B3: [3, 11]
B4: [20, 12]
B5: [13, 5, 21]
B6: [22]
B7: [23, 15, 15]

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [24]
B1: [1, 17, 1]
B2: [26, 10]
B3: [3, 11]
B4: [20, 12]
B5: [13, 5, 21]
B6: [22, 14]
B7: [23, 15, 15]

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [24]
B1: [1, 17, 1]
B2: [26, 10]
B3: [3, 11]
B4: [20, 12]
B5: [13, 5, 21]
B6: [22, 14, 14]
B7: [23, 15, 15]

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [24, 16]
B1: [1, 17, 1]
B2: [26, 10]
B3: [3, 11]
B4: [20, 12]
B5: [13, 5, 21]
B6: [22, 14, 14]
B7: [23, 15, 15]

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [24, 16]
B1: [1, 17, 1]
B2: [26, 10]
B3: [3, 11, 3]
B4: [20, 12]
B5: [13, 5, 21]
B6: [22, 14, 14]
B7: [23, 15, 15]

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, **17**, 11, 17]
[7, 10, 29, 3]

B0: [24, 16]
B1: [1, 17, 1, 17]
B2: [26, 10]
B3: [3, 11, 3]
B4: [20, 12]
B5: [13, 5, 21]
B6: [22, 14, 14]
B7: [23, 15, 15]

- ❖ Assuming page size = 4
- ❖ B1 is full when we process 17
- ❖ B1 is written to disk

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, **17**, 11, 17]
[7, 10, 29, 3]

B0: [24, 16]
B1: []
B2: [26, 10]
B3: [3, 11, 3, 11]
B4: [20, 12]
B5: [13, 5, 21]
B6: [22, 14, 14]
B7: [23, 15, 15]

- ❖ Assuming page size = 4
- ❖ B3 is full when we process 11
- ❖ B3 is written to disk

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, **11**, 17]
[7, 10, 29, 3]

B0: [24, 16]
B1: []
B2: [26, 10]
B3: []
B4: [20, 12]
B5: [13, 5, 21]
B6: [22, 14, 14]
B7: [23, 15, 15]

- ❖ Assuming page size = 4
- ❖ B3 is full when we process 11
- ❖ B3 is written to disk

Hash Partitioning

S

[26, 20, 8, 8]

[3, 22, 1, 23]

[15, 12, 17, 13]

[10, 11, 5, 1]

[0, 21, 15, 16]

[24, 14, 14, 16]

[3, 17, 11, 17]

[7, 10, 29, 3]

B0: [24, 16]

B1: [17]

B2: [26, 10]

B3: []

B4: [20, 12]

B5: [13, 5, 21]

B6: [22, 14, 14]

B7: [23, 15, 15]

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[**7**, 10, 29, 3]

B0: [24, 16]
B1: [17]
B2: [26, 10]
B3: []
B4: [20, 12]
B5: [13, 5, 21]
B6: [22, 14, 14]
B7: [23, 15, 15, 7]

- ❖ Assuming page size = 4
- ❖ B7 is full when we process 7
- ❖ B7 is written to disk

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [24, 16]
B1: [17]
B2: [26, 10, 10]
B3: []
B4: [20, 12]
B5: [13, 5, 21]
B6: [22, 14, 14]
B7: []

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, **29**, 3]

B0: [24, 16]
B1: [17]
B2: [26, 10, 10]
B3: []
B4: [20, 12]
B5: [13, 5, 21, 29]
B6: [22, 14, 14]
B7: []

- ❖ Assuming page size = 4
- ❖ B5 is full when we process 29
- ❖ B5 is written to disk

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, **29**, 3]

B0: [24, 16]
B1: [17]
B2: [26, 10, 10]
B3: []
B4: [20, 12]
B5: []
B6: [22, 14, 14]
B7: []

- ❖ Assuming page size = 4
- ❖ B5 is full when we process 29
- ❖ B5 is written to disk

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [24, 16]
B1: [17]
B2: [26, 10, 10]
B3: [3]
B4: [20, 12]
B5: []
B6: [22, 14, 14]
B7: []

Hash Partitioning

S

[26, 20, 8, 8]
[3, 22, 1, 23]
[15, 12, 17, 13]
[10, 11, 5, 1]
[0, 21, 15, 16]
[24, 14, 14, 16]
[3, 17, 11, 17]
[7, 10, 29, 3]

B0: [24, 16] + [8, 8, 0, 16]
B1: [17] + [1, 17, 1, 17]
B2: [26, 10, 10]
B3: [3] + [3, 11, 3, 11]
B4: [20, 12]
B5: [] + [13, 5, 21, 29]
B6: [22, 14, 14]
B7: [] + [23, 15, 15, 7]

R ⋈ S using Grace Hash Join

R Partitioned R

[28, 7, 1, 13]	P0: [24, 8, 16]
[7, 25, 24, 17]	P1: [1, 25, 17]
[19, 6, 22, 4]	P2: [18]
[8, 4, 20, 16]	P3: [19, 19, 11]
[18, 19, 11, 29]	P4: [28, 4, 4, 20]
	P5: [13, 29]
	P6: [6, 22]
	P7: [7, 7]

S Partitioned S

[26, 20, 8, 8]	P0: [8, 8, 0, 16, 24, 16]
[3, 22, 1, 23]	P1: [1, 17, 1, 17, 17]
[15, 12, 17, 13]	P2: [26, 10, 10]
[10, 11, 5, 1]	P3: [3, 11, 3, 11, 3]
[0, 21, 15, 16]	P4: [20, 12]
[24, 14, 14, 16]	P5: [13, 5, 21, 29]
[3, 17, 11, 17]	P6: [22, 14, 14]
[7, 10, 29, 3]	P7: [23, 15, 15, 7]

R ⋈ S using Grace Hash Join

R: P0

P0: [24, 8, 16]

R: P0 hash table



P0 Output:

$[(8, 8), (8, 8), (16, 16), (24, 24), (16, 16)]$

S: P0

P0: [8, 8, 0, 16, 24, 16]

This algorithm avoids rescanning the entire S relation by first partitioning both R and S via a hash function, and writing these partitions out to disk. The algorithm then loads pairs of partitions into memory, builds a hash table for the smaller partitioned relation, and probes the other relation for matches with the current hash table. Because the partitions were formed by hashing on the join key, it must be the case that any join output tuples must belong to the same partition.

It is possible that one or more of the partitions still does not fit into the available memory, in which case the algorithm is recursively applied: an additional orthogonal hash function is chosen to hash the large partition into sub-partitions, which are then processed as before. Since this is expensive, the algorithm tries to reduce the chance that it will occur by forming the smallest partitions possible during the initial partitioning phase.

Better Alternative?

- ❖ Hash Partitioning requires
 - ❖ 1 page for input
 - ❖ 1 page each for output buckets
 - ❖ considerably less than a large table
- ❖ Hash Join (1 Pass) requires
 - ❖ enough memory for the entire R relation
- ❖ So likely to have memory available when we decide to do the Grace join approach

Hash Join: Hybrid Hash

- ❖ If we ignore the memory requirements for hash partitioning, we can keep the first partition of R (Partition 0), in memory
- ❖ Only have to write out and read the remaining partitions
- ❖ Flexible: if we only need one partition, becomes like the ideal case where R fits in memory

Hash Joins Summary

- ❖ Efficient but
 - ❖ Need a lot of memory to run well
 - ❖ Can't really exploit the B+ tree indexes

(Sort)-Merge Join

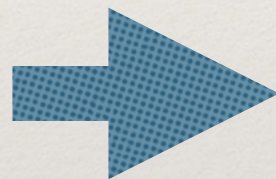
- ❖ Sort R and S on the join column(s)
 - ❖ Hopefully, one or both are already sorted!
- ❖ Scan R and S finding the matches, output matching rows
- ❖ Sort costs are $O(|R| \log |R|)$, usually 4 I/Os (2 pass)
 - ❖ Followed by Reading of the sorted R and S (join-merge)
 - ❖ Can combine the merge phase of the sort-pass with the “join-merge” phase
- ❖ Memory Efficient (compared to Hash Joins)
- ❖ One or both relations can be already be sorted because of the nature of PK/FK relationships and how PI is generally chosen

R \bowtie S via Sort Merge - Sort

R

```
[ 28,  7,  1, 13]
[  7, 25, 24, 17]
[ 19,  6, 22,  4]
[  8,  4, 20, 16]
[ 18, 19, 11, 29]
```

```
[  1,  4,  4,  6]
[  7,  7,  8, 11]
[ 13, 16, 17, 18]
[ 19, 19, 20, 22]
[ 24, 25, 28, 29]
```



S

```
[ 26, 20,  8,  8]
[  3, 22,  1, 23]
[ 15, 12, 17, 13]
[ 10, 11,  5,  1]
[  0, 21, 15, 16]
[ 24, 14, 14, 16]
[  3, 17, 11, 17]
[  7, 10, 29,  3]
```

```
[  0,  1,  1,  3]
[  3,  3,  5,  7]
[  8,  8, 10, 10]
[ 11, 11, 12, 13]
[ 14, 14, 15, 15]
[ 16, 16, 17, 17]
[ 17, 20, 21, 22]
[ 23, 24, 26, 29]
```


R ⋈ S via Sort Merge - Sort

R	→	1	S	0	←
		4		1	
		4		1	
		6		3	
		7		3	
		7		3	
		8		5	
		11		7	
		13		8	
		16		8	
		17		10	
		18		10	
		19		11	
		19		11	
		20		12	
		22		13	
		24		14	
		25		14	
		28		15	
		29		15	
				...	

- ❖ No match
- ❖ Increment the smaller of two

match output

R \bowtie S via Sort Merge - Sort

R	→ 1	S	0
	4		1 ←
	4		1
	6		3
	7		3
	7		3
	8		5
	11		7
	13		8
	16		8
	17		10
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

- ❖ match, output (1,1)
- ❖ Increment S

Output:
(1,1)

R ⋈ S via Sort Merge - Sort

R	→ 1	S	0
	4		1
	4		1 ←
	6		3
	7		3
	7		3
	8		5
	11		7
	13		8
	16		8
	17		10
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

- ❖ match, output (1,1)
- ❖ Increment S

Output:
(1,1)
(1,1)

R ⋈ S via Sort Merge - Sort

R	→	1	S	0
		4		1
		4		1
		6		3 ←
		7		3
		7		3
		8		5
		11		7
		13		8
		16		8
		17		10
		18		10
		19		11
		19		11
		20		12
		22		13
		24		14
		25		14
		28		15
		29		15
				...

- ❖ no match
- ❖ Increment smaller of two

Output:
(1,1)
(1,1)

R ⋈ S via Sort Merge - Sort

R	→	1	S	0
		4		1
		4		1
		6		3 ←
		7		3
		7		3
		8		5
		11		7
		13		8
		16		8
		17		10
		18		10
		19		11
		19		11
		20		12
		22		13
		24		14
		25		14
		28		15
		29		15
				...

- ❖ no match
- ❖ Increment smaller of two
- ❖ Have to consider tied values in R as a group

Output:
(1, 1)
(1, 1)

R ⋈ S via Sort Merge - Sort

R	→	1	S	0
		4		1
		4		1
		6		3
		7		3
		7		3
		8		5 ←
		11		7
		13		8
		16		8
		17		10
		18		10
		19		11
		19		11
		20		12
		22		13
		24		14
		25		14
		28		15
		29		15
				...

- ❖ no match
- ❖ Increment smaller of two
 - ❖ optimize to next higher value

Output:
(1,1)
(1,1)

R ⋈ S via Sort Merge - Sort

R	1	S	0
	4		1
	4		1
→	6		3
	7		3
	7		3
	8		5 ←
	11		7
	13		8
	16		8
	17		10
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

- ❖ no match
- ❖ Increment smaller of two
 - ❖ tied values are treated as a group in R

Output:
(1,1)
(1,1)

R \bowtie S via Sort Merge - Sort

R	1	S	0
	4		1
	4		1
→	6		3
	7		3
	7		3
	8		5
	11		7 ←
	13		8
	16		8
	17		10
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

- ❖ no match
- ❖ Increment smaller of two

Output:
(1,1)
(1,1)

R ⋈ S via Sort Merge - Sort

R	1	S	0
	4		1
	4		1
	6		3
→	7		3
	7		3
	8		5
	11		7 ←
	13		8
	16		8
	17		10
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

- ❖ match, output (7,7), (7,7)
- ❖ have to iterate through all tied values in R
- ❖ increment S

Output:

(1,1)

(1,1)

(7,7)

(7,7)

R \bowtie S via Sort Merge - Sort

R	1	S	0
	4		1
	4		1
	6		3
→	7		3
	7		3
	8		5
	11		7
	13		8 ←
	16		8
	17		10
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

❖ mismatch

❖ increment smaller value

Output:

(1,1)

(1,1)

(7,7)

(7,7)

R \bowtie S via Sort Merge - Sort

R	1	S	0
	4		1
	4		1
	6		3
	7		3
	7		3
→	8		5
	11		7
	13		8 ←
	16		8
	17		10
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

- ❖ match output (8,8)
- ❖ increment S

Output:
(1,1)
(1,1)
(7,7)
(7,7)
(8,8)

R ⋈ S via Sort Merge - Sort

R	1	S	0
	4		1
	4		1
	6		3
	7		3
	7		3
→	8		5
	11		7
	13		8
	16		8 ←
	17		10
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

❖ match, output (8,8)

❖ increment S

Output:

(1,1)

(1,1)

(7,7)

(7,7)

(8,8)

(8,8)

R \bowtie S via Sort Merge - Sort

R	1	S	0
	4		1
	4		1
	6		3
	7		3
	7		3
→	8		5
	11		7
	13		8
	16		8
	17		10 ←
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

- ❖ mismatch
- ❖ increment smaller of two

Output:

(1,1)

(1,1)

(7,7)

(7,7)

(8,8)

(8,8)

R \bowtie S via Sort Merge - Sort

R	1	S	0
	4		1
	4		1
	6		3
	7		3
	7		3
	8		5
→	11		7
	13		8
	16		8
	17		10 ←
	18		10
	19		11
	19		11
	20		12
	22		13
	24		14
	25		14
	28		15
	29		15
			...

❖ mismatch

❖ increment smaller of two

Output:

(1,1)

(1,1)

(7,7)

(7,7)

(8,8)

(8,8)

Sort-Merge Join

- ❖ Some special handling required to handle ties
- ❖ Otherwise a simple scan over the two sorted inputs
- ❖ Think “Block nested loop” being performed on tied values
- ❖ Multiple scans of the tied rows of S may be required in really bad cases if the tied R rows don't fit in memory