R JOIN S?

R

C	
40	T1
60	T2
30	T3
10	T4
20	T5

S

C	
10	T6
60	T7
40	T8
20	T9

Nested-Loop Join (NLJ)

For each r $\in$ R do

  For each s $\in$ S do

    if r.C = s.C then output r,s pair

R

40	T1
60	T2
30	T3
10	T4
20	T5

S

10	T6
60	T7
40	T8
20	T9

- *R* is called the **outer relation** and *S* the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

For each r $\in$ R do
 For each s $\in$ S do
 if r.C = s.C then output r,s pair

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

$$b_R + n_R * b_S \text{ block transfers}$$

- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks

- When neither relation fits in memory, **Block Nested-Loops** algorithm (next slide) should be used instead.

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

for each block B_r **of** r **do begin**
    for each block B_s **of** ***s*** **do begin**
        for each tuple t_r **in** B_r **do begin**
            for each tuple t_s **in** B_s **do begin**
                Check if (t_r, t_s) satisfy the join condition
                if they do, add $t_r \bullet t_s$ to the result.
            end
        end
    end
end

Block Nested-Loop Join (Cont.)

- **Worst case estimate: $b_R + b_R * b_S$ block transfers**
  - Each block in the inner relation *S* is read once for each *block* in the outer relation R.
- **Best case: $b_R + b_S$ block transfers (***if $1+ b_S$ blocks fit in memory)*
- Many Improvements proposed to nested loop and block nested loop algorithms including:
  - **Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)**
  - **Use index on inner relation if available (next slide)**

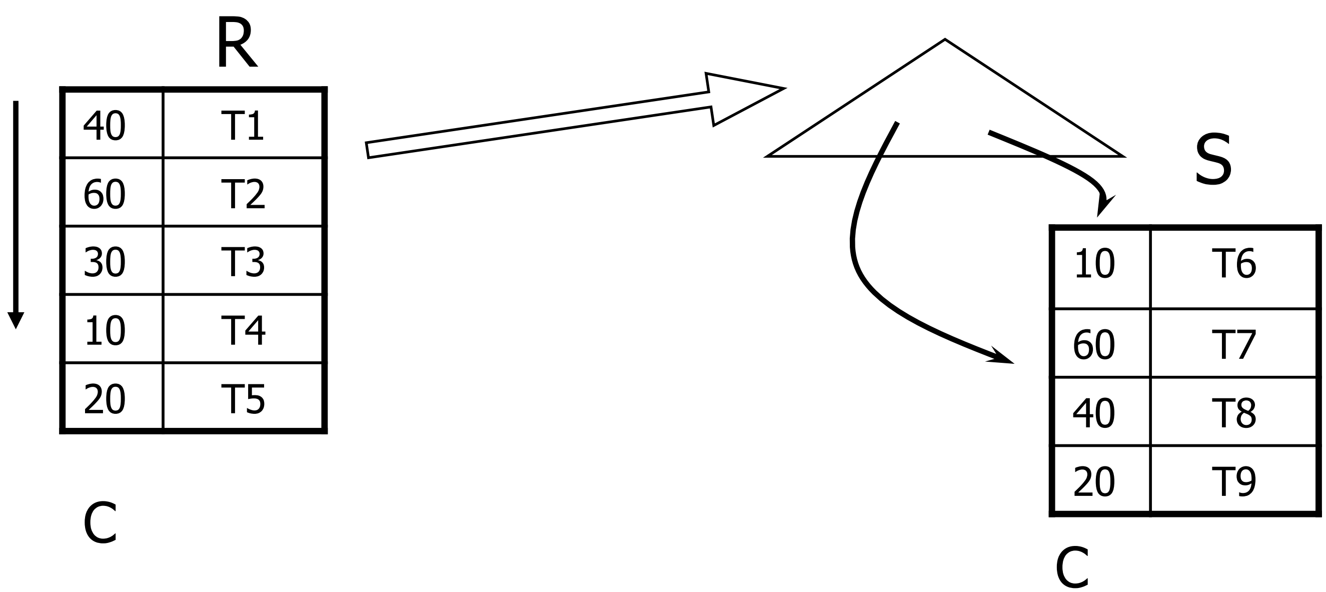
E.g. Compute *student* $\bowtie$ *takes,* with R=*student* as the outer relation, S= takes the inner relation:

- Number of records for *student :* 5,000, and for *takes*: 10,000
- Number of blocks for *student:* 100 and for *takes*: 400

Index Join (IJ)

(1) Create an index for S.C if one is not already there

(2) For each r $\in$ R find matching records in S
(s.C=r.C) and output (r,s)



Example of Nested-Loop Join Costs

- Compute *student* $\bowtie$ *takes,* with *student* as the outer relation.
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400
- Let *takes* have a primary B^+-tree index on the attribute *ID,* which contains 20 entries in each index node. Since *takes* has 10,000 tuples, the height of the tree is 4.
- 4 nodes accessed from the index +1 more access to the actual data
- *student* has 5000 tuples
- Cost of indexed nested loops join
  - 100 + 5000 * 5 = 25,100 block transfers
- Cost of block nested loops join
  - 400*100 + 100 = 40,100 block transfers assuming worst case memory
    - may be significantly less with more memory

Hash Join (HJ)

- Hash function h(v), range 1 → k

<u>Algorithm</u>

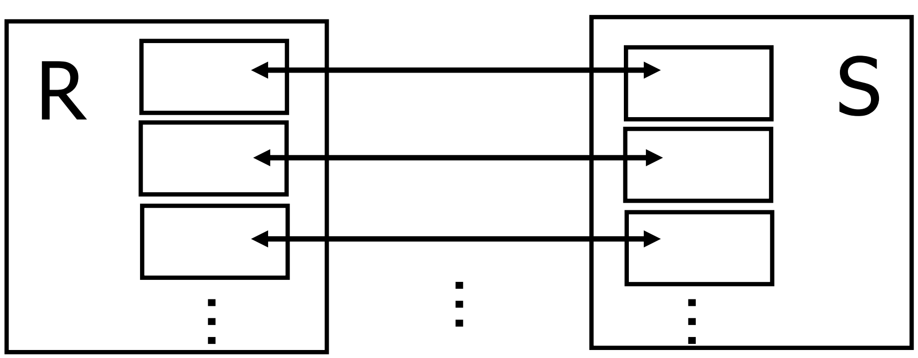
(1) Hashing stage (bucketizing): hash tuples into buckets

- Hash R tuples into G1,…,Gk buckets
- Hash S tuples into H1,…,Hk buckets

(2) Join stage: join tuples in matching buckets

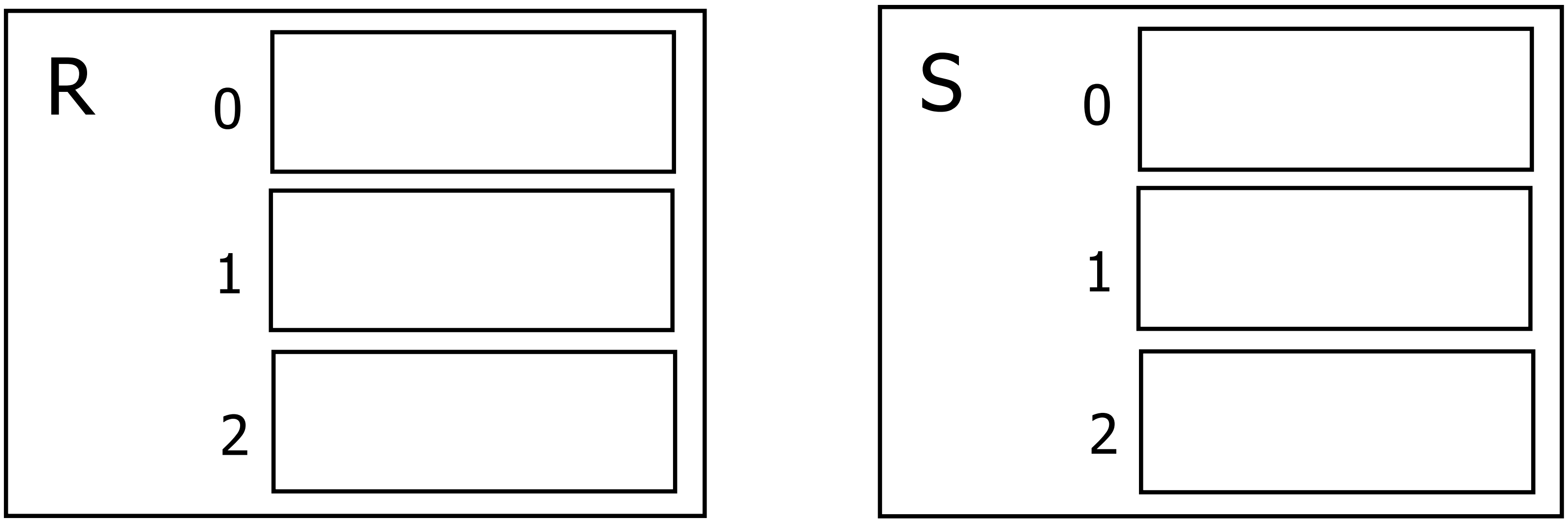
- For i = 1 to k do

  match tuples in Gi, Hi buckets



Hash Join (HJ)

- H(k) = k mod 3

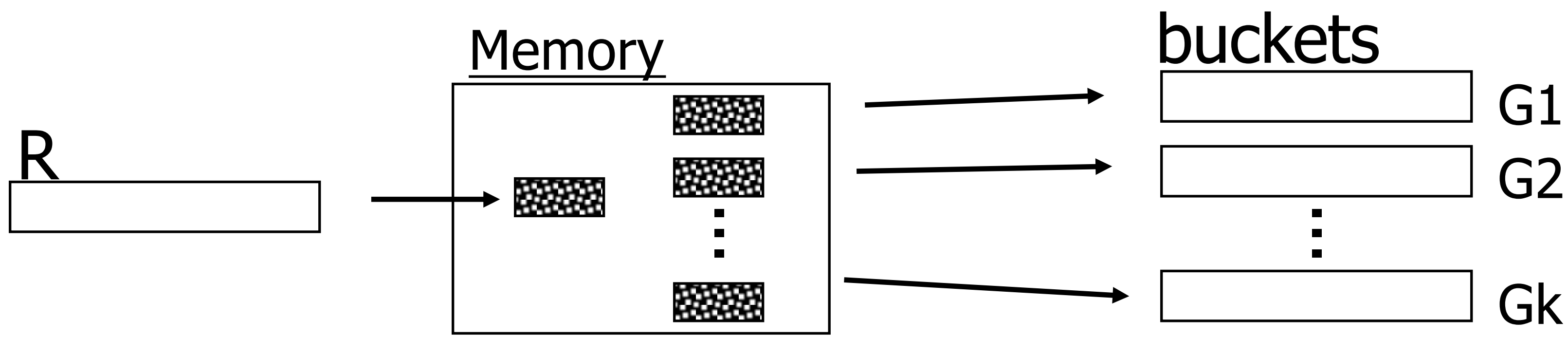


40	T1
60	T2
30	T3
10	T4
20	T5

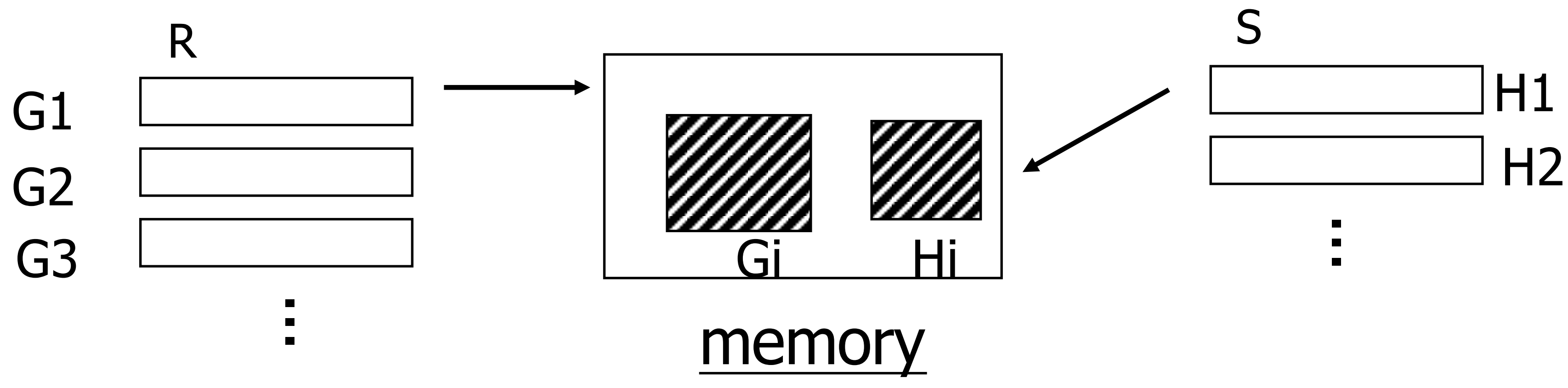
10	T6
60	T7
40	T8
20	T9

Hash Equi Join (HJ) of G and H

- Step (1): Hashing stage

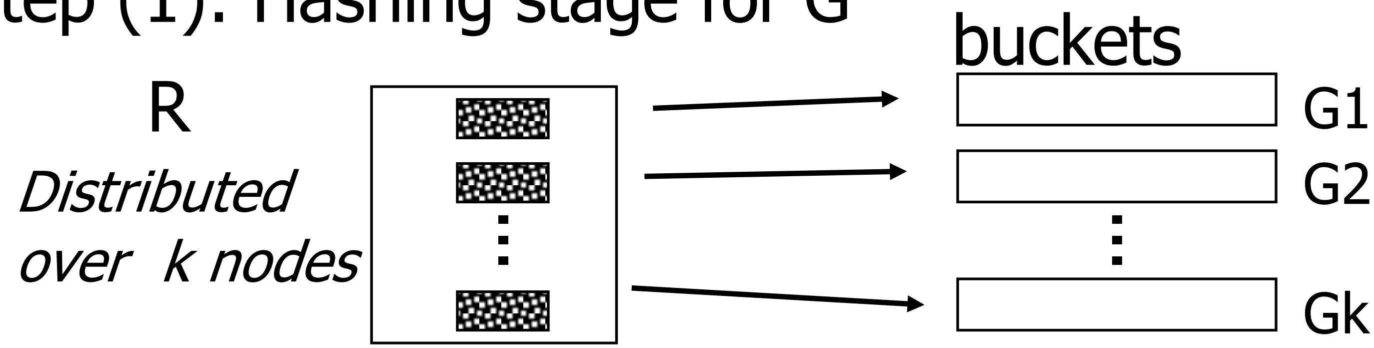


- Step (2): Join stage



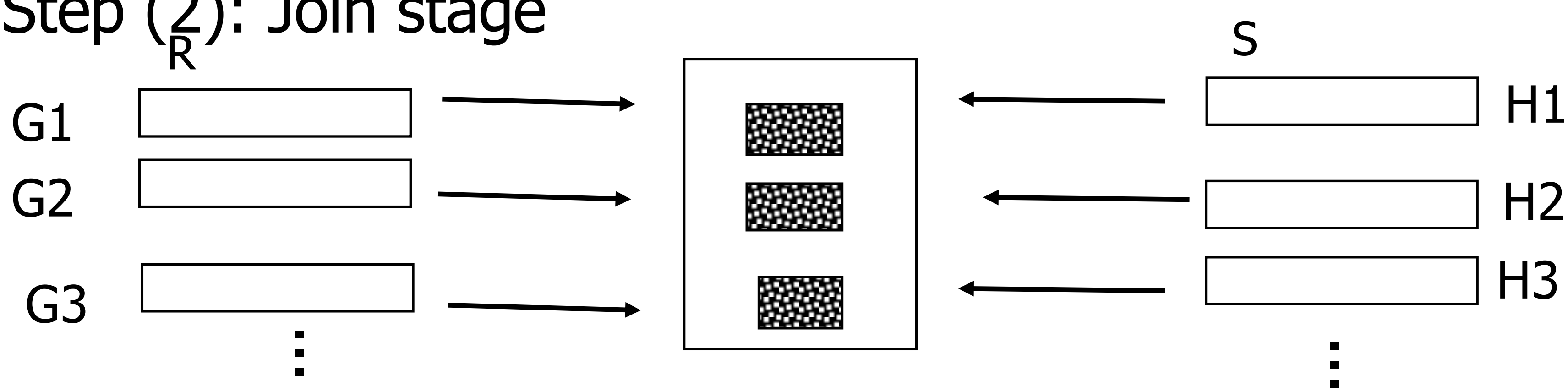
HJ: Critical for Parallel Joins

- Step (1): Hashing stage for G



  –
  – Ditto for H: H1, H2, H3 …

- Step (2): Join stage



Sort-Merge Join (SMJ)

- Sort the relations first and then join

R

10	T4
20	T5
30	T3
40	T1
60	T2

S

10	T6
20	T9
40	T8
60	T7

Quicksort can be used if data fits in memory
External sor is needed otherwise.

Sort-Merge Join (SMJ)

(1) if R and S not sorted, sort them

(2) $i \leftarrow 1$; $j \leftarrow 1$;

While $(i \leq |R|) \wedge (j \leq |S|)$ do

if R[i].C = S[j].C then outputTuples

else if R[i].C > S[j].C then $j \leftarrow j+1$

else if R[i].C < S[j].C then $i \leftarrow i+1$

R

i

10	T4
20	T5
30	T3
40	T1
60	T2

S

j

10	T6
20	T9
40	T8
60	T7

<u>(no incrementr to I or J when we have equality?)</u>

Sort-Merge Join (SMJ)

(1) if R and S not sorted, sort them

(2) $i \leftarrow 1$; $j \leftarrow 1$;

While $(i \leq |R|) \wedge (j \leq |S|)$ do

if R[i].C = S[j].C then {outputTuples;
$j \leftarrow j+1$; $i \leftarrow i+1$ }

else if R[i].C > S[j].C then $j \leftarrow j+1$

else if R[i].C < S[j].C then $i \leftarrow i+1$

R
i

10	T4
20	T5
30	T3
40	T1
60	T2

S
j

10	T6
20	T9
40	T8
60	T7

<u>This only works when the keys do not contain duplicates</u>

External Merge Sort

- *Idea:* ***Divide and conquer:*** sort subfiles and merge into larger sorts

Pass 0 ➔ Only one memory block is needed

Pass I > 0 ➔ Only three memory blocks are needed