	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	**BD**	**0.5BD**	**BD**	**2D**	**Search +D**
(2) Sorted	**BD**	**Dlog 2B**	**Dlog 2 B + # matches**	**Search + BD**	**Search +BD**
(3) Clustered	**1.5BD**	**Dlog F 1.5B**	**Dlog F 1.5B + # matches**	**Search + D**	**Search +D**
(4) Unclustered Tree index	**BD(R+0.15)**	**D(1 + log F 0.15B)**	**Dlog F 0.15B + # matches**	**D(3 + log F 0.15B)**	**Search + 2D**
(5) Unclustered Hash index	**BD(R+0.125)**	**2D**	**BD**	**4D**	**Search + 2D**

average of the I/O cost only

Join Operations

Adapted from Database System Concepts

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Examples use the following information
  - Number of records of *customer*: 10,000 *depositor*: 5000
  - Number of blocks of *customer*: 400 *depositor*: 100

Nested-Loop Join

- To compute the theta join $r \bowtie_{\theta} s$

for each tuple t_r **in** r **do begin**
    for each tuple t_s **in** s **do begin**
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add $t_r \bullet t_s$ to the result.
    end
end

r is called the **outer relation** and *s* the **inner relation** of the join.

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  $$n_r * b_s + b_r$$
  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
- Assuming worst case memory availability cost estimate is
  - with *depositor* as outer relation:
    - 5000 $*$ 400 + 100 = 2,000,100 block transfers,
  - with *customer* as the outer relation
    - 10000 $*$ 100 + 400 = 1,000,400 block transfers
- If smaller relation (*depositor)* fits entirely in memory, the cost estimate will be 500 block transfers.

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 • t_s to the result.
            end
        end
    end
end
```

Block Nested-Loop Join (Cont.)

- Worst case estimate: $b_r * b_s + b_r$ block transfers

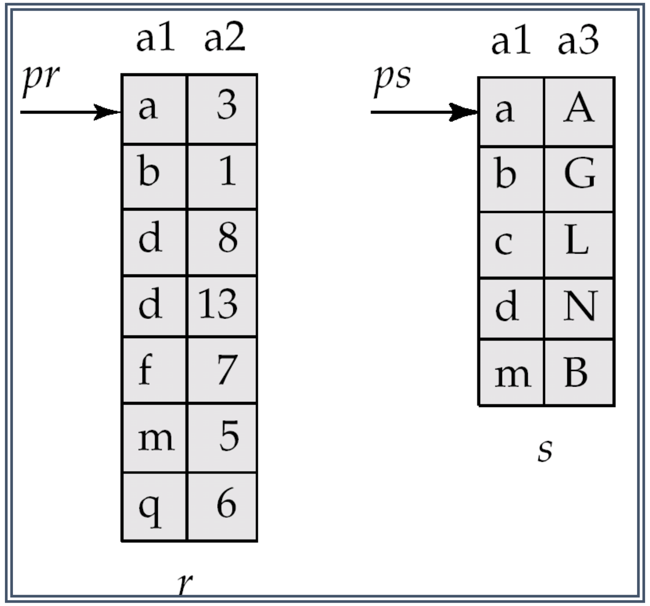
- Best case: $b_r + b_s$ block transfers
- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use M — 2 disk blocks as blocking unit for outer relations, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r*.*
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $b_r\, t_T + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them



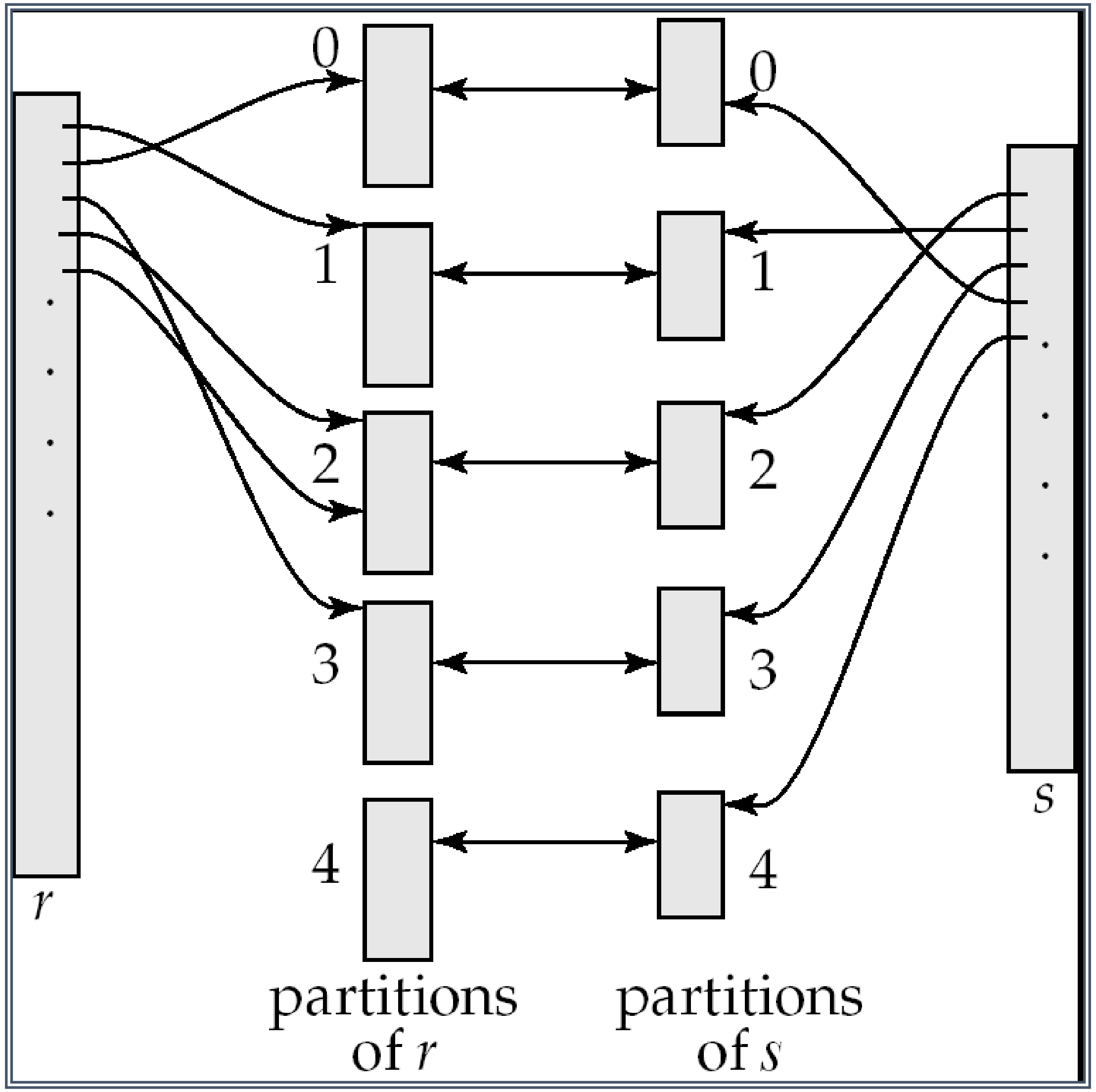
Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:
  $b_r + b_s$ block transfers + the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B^+-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B^+-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
  - Intuition: partitions fit in memory
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - r_0, r_1, . . ., r_n denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r [JoinAttrs])$.
  - r_0,, r_1. . ., r_n denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s [JoinAttrs])$.

Hash-Join (Cont.)



Hash-Join (Cont.)

- *r* tuples in r_i need only to be compared with *s* tuples in s_i Need not be compared with *s* tuples in any other partition, since:
  - an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value *i*, the *r* tuple has to be in r_i and the *s* tuple in s_i.

Hash-Join Algorithm

The hash-join of *r* and *s* is computed as follows.

1. **Partition** the relation *s* using hashing function *h*.
    1. When partitioning a relation, one block of memory is reserved as the output buffer for each partition, and one block for input
    2. If extra memory is available, allocate b_b blocks as buffer for input and each output
2. Partition *r* similarly.
3. … next slide ..

Hash Join (Cont.)

Hash Join Algorithm (cont)

3. For each partition *i:*
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute.
      - This hash index uses a different hash function than the earlier one *h.*

   (b) Read the tuples in r_i from the disk one by one.
      - For each tuple t_r probe the in-memory hash index to find all matching tuples t_s in s_i
        - For each matching tuple t_s in s_i
          - output the concatenation of the attributes of t_r and t_s

Relation *s* is called the **build input** and
r is called the **probe input.**

Hash-Join algorithm (Cont.)

- The value *n* and the hash function *h* is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "fudge factor", typically around 1.2
  - The probe relation partitions s_i need not fit in memory
- **Recursive partitioning** required if number of partitions *n* is greater than number of pages *M* of memory.
  - instead of partitioning *n* ways, use $M - 1$ partitions for s
  - Further partition the $M - 1$ partitions using a different hash function
  - Use same partitioning method on *r*
  - Rarely required: e.g., recursive partitioning not needed for relations of 1GB or less with memory size of 2MB, with block size of 4KB.

Handling of Overflows

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition s_i if s_i does not fit in memory. Reasons could be
  - Many tuples in s with same value for join attributes
  - Bad hash function
- **Overflow resolution** can be done in build phase
  - Partition s_i is further partitioned using different hash function.
  - Partition r_i must be similarly partitioned.
- **Overflow avoidance** performs partitioning carefully to avoid overflows during build phase
  - E.g. partition build relation into many partitions, then combine them
- Both approaches fail with large numbers of duplicates
  - Fallback option: use block nested loops join on overflowed partitions

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is

  $3(b_r + b_s) + 4 * n_h$ block transfers +

  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
- If recursive partitioning required:
  - number of passes required for partitioning build relation

    s is $\lceil log_{M-1}(b_s) - 1 \rceil$
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:

    $2(b_r + b_s \lceil log_{M-1}(b_s) - 1 \rceil + b_r + b_s$ block transfers +

    $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \lceil log_{M-1}(b_s) - 1 \rceil$ seeks
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash-Join $customer \bowtie depositor$

- Assume that memory size is 20 blocks
- $b_{depositor}$= 100 and $b_{customer}$ = 400.
- *depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition *customer* into five partitions,each of size 80. This is also done in one pass.
- Therefore total cost, ignoring cost of writing partially filled blocks:
  - 3(100 + 400) = 1500 block transfers + $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil)$ = 336 seeks

13.3 Let relations $r_1(A, B, C)$ and $r_2(C, D, E)$ have the following properties: r_1 has 20,000 tuples, r_2 has 45,000 tuples, 25 tuples of r_1 fit on one block, and 30 tuples of r_2 fit on one block. Estimate the number of block transfers and seeks required, using each of the following join strategies for $r_1 \bowtie r_2$:

a. Nested-loop join

b. Block nested-loop join

c. Merge join