Comp 5311 Database Management Systems

12. Join Algorithms

Join Processing

- Several different algorithms to implement joins
- Choice based on cost estimate. We only take into account the I/O operations (reads and writes of pages)
- Terminology:
  - *r, s* relations to be joined
  - n_r, n_s number of records in *r, s*
  - b_r, b_s number of pages in *r, s*
  - *M* available memory in pages
- Examples assume equijoins on the following tables
  - Number of records of *customer*: 10,000 *depositor*: 5000
  - Number of pages of *customer*: 400 *depositor*: 100
  - The join attribute is the *customer-name*, which is the key of customer.

Block Nested-Loop Join

- We wish to compute *r* JOIN *s*
- *r* is called the **outer relation** and *s* the **inner relation** of the join.
- Block nested loop join requires no indices and can be used with any kind of join condition.

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**
        if (t_r, t_s) satisfies the join condition
          add (t_r, t_s) to the result.

Block Nested-Loop Join (Cont.)

- Worst case estimate: $b_r * b_s + b_r$ page accesses.
  - Each page in the inner relation *s* is read once for each *page* in the outer relation
- Best case: $b_r + b_s$ block accesses.
- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use $M - 2$ disk pages as blocking unit for outer relations, where M = memory size in pages; use remaining two pages to buffer inner relation and output
  - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$
- Optimizations:
  - If equi-join attribute forms a key or inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the pages remaining in buffer (with LRU replacement)
  - Use main-memory hash table for the outer relation (to decrease CPU cost)

Example of Block Nested-Loop Join Costs

$b_{depositor}$= 100, $b_{customer}$=400

- Compute *depositor JOIN customer,* with *depositor* as the outer relation.
- Worst case cost of block nested-loop
  - 100*400 + 100 = **40,100** page accesses
  - How many main memory pages you need to apply block nested-loop?
- Best case cost of block nested-loop join
  - 100 + 400 = **500** page accesses
  - How many main memory pages you need to achieve this cost?
- Cost of block nested loops join with 52 main memory pages
  - 2*400 + 100 = **900** page accesses

Indexed Nested-Loop Join

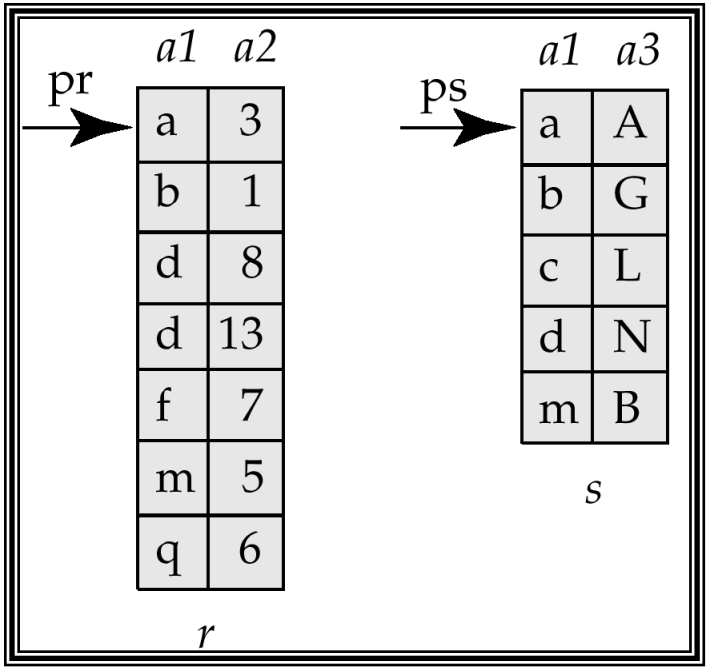
- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
  - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t_r.
- Cost of the join: $b_r + n_r * c$
  - Where c is the cost of traversing index and fetching all matching s tuples for one tuple or r
  - c can be estimated as cost of a single selection on s using the join condition.
- If indices are available on join attributes of both r and s, use the relation with fewer tuples as the outer relation.

Example of Indexed Nested-Loop Join Costs

- Compute *depositor JOIN customer,* with *depositor* as the outer relation.
- Let *customer* have a primary B^+-tree index with 4 levels on the join attribute *customer-name (*which is the primary key of *customer)*.
- Number of pages $b_{depositor} = 100$
- Number of records $n_{depositor} = 5000$
- Cost of indexed nested loops join
  - 100 + 5000 * 5 = **25,100** disk accesses.
- CPU cost likely to be less than that for block nested loops join
- Indexed Nested-Loop is the best algorithm if there are selective conditions on the outer relation

Merge-Join

- Sort both relations on their join attribute (if not already sorted on the join attributes).
- Merge the sorted relations to join them
  - Join step is similar to the merge stage of the sort-merge algorithm.
  - Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched



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 number of page accesses for merge-join is

  $b_r + b_s$ + the cost of sorting if relations are unsorted.

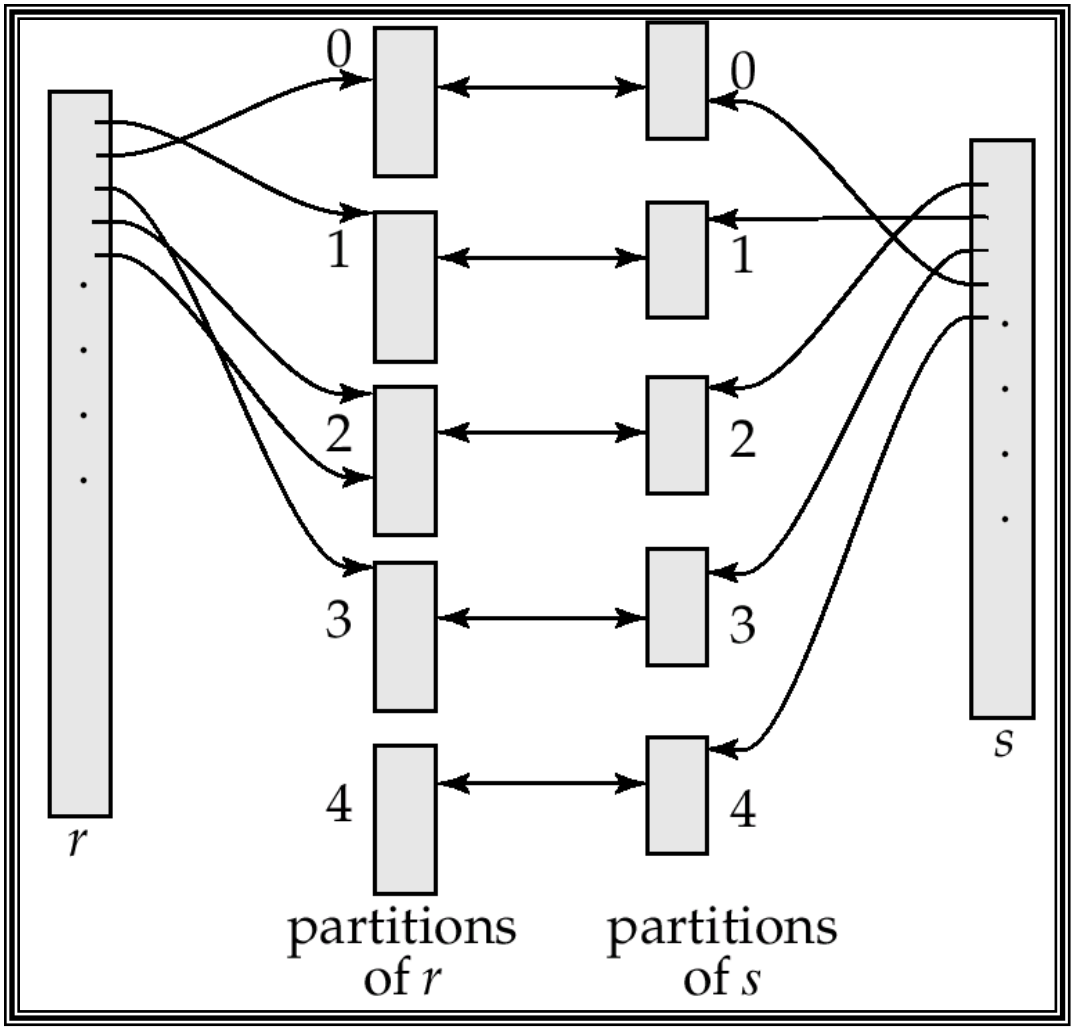
Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations into *n* buckets (i.e., a hash file organization)
- *h* maps *JoinAttrs* values to {0, 1, ..., *n-1*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.

  $r_0, r_1, \ldots, r_{n-1}$ denote partitions of *r* tuples
  - Each tuple $t_r \in r$ is in partition r_i where $i = h(t_r[JoinAttrs])$.

  $s_0, s_1, \ldots, s_{n-1}$ denote partitions of *s* tuples
  - Each tuple $t_s \in s$ is in partition s_i, where $i = h(t_s[JoinAttrs])$.

Hash-Join (Cont.)



Hash-Join (Cont.)

- *r* tuples in bucket/partition 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

1. Partition the relation r using hashing function h. When partitioning a relation, one page of memory is reserved as the output buffer for each partition.
2. Partition s similarly.
3. For each i:
   - Load bucket r_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. Relation r is called the **build input**.
   - Read the tuples in bucket s_i from the disk page by page. For each tuple t_s locate each matching tuple t_r in r_i using the in-memory hash index. Relation s is called the **probe input.**

Hash-Join algorithm (Cont.)

- The number of buckets n is such that each bucket of the build input r should fit in the available main memory pages M. Assuming each bucket has the same size:

  $$M \geq \lceil b_r / n \rceil$$

- Also $M \geq$ n+1 because for each bucket we should have one buffer page (plus one page for input buffer)
- In order to satisfy these conditions: $M > sqrt(b_r)$
  - The probe relation partitions need not fit in memory
- **Recursive partitioning** required if number of partitions n is greater than number of pages M of memory.
  - Rarely necessary: e.g., recursive partitioning not needed for relations of 1GB or less with memory size of 2MB, with page size of 4KB.

Example of Hash-Join between *customer and depositor*

- Assume that memory size is M=25 pages
- $b_{depositor}$= 100 and $b_{customer}$ = 400.
- *depositor* is the build input.
  - Partition *depositor* into 5 buckets, each of size 20 pages. This partitioning can be done in one pass.
- *customer* is the probe input.
  - Partition *customer* into 5 buckets, each of size 80 pages. This is also done in one pass.
- Read each bucket in turn of the build input in memory, and probe against records of the corresponding probe bucket.
- Therefore total cost: 3(100 + 400) = 1500 page transfers
  - ignores cost of writing partially filled pages

Hybrid hash-join

- If the memory is large enough we can keep one or more buckets of one file in memory at all times. Lets say that we have 10 buckets and that each bucket is 90 pages. If we have 100 main memory pages, when we partition the build input *r* we keep the entire first bucket in memory and allocate 9 pages for the remaining buckets and 1 for reading the file page by page.
- When we read the probe input *s* , we use again 10 buckets and the same hash function. If a record falls in the first bucket, we produce immediately results since we have the first bucket of *r* (90 pages) in memory.
- In this way we avoid writing and reading back the first buckets of both *r* and *s*.
- If we have more memory, we can keep more buckets.
- It is better to partition the smallest file (i.e., the build input) first since it has smaller buckets and we may be able to keep more in memory.