Database Systems
Lecture16 – Chapter 15: Query Processing

SUNG KYUN KWAN UNIVERSITY

Beomseok Nam (남범석)

bnam@skku.edu

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of *student*: 5,000
  - Number of records of *takes*: 10,000
  - Number of blocks of *student*: 100
  - Number of blocks of *takes*: 400

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: $b_r\,(t_T + t_S) + n_r * c$
  - 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 Nested-Loop Join Costs

- Compute *student* ⋈ *takes,* with *student* as the outer relation.
- 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, and one more access is needed to find the actual data
- *student* has 5000 tuples
- Cost of block nested loops join
  - 400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested loops join
  - 100 + 5000 * 5 = 25,100 block transfers and seeks.
  - # of seeks increased

Sort-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
   1. Join step is similar to the merge stage of the sort-merge algorithm.
   2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
   3. Detailed algorithm in book

pr

a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

r

ps

a1	*a3*
a	A
b	G
c	L
d	N
m	B

s

Sort-Merge-Join (Cont.)

- Can be used 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:

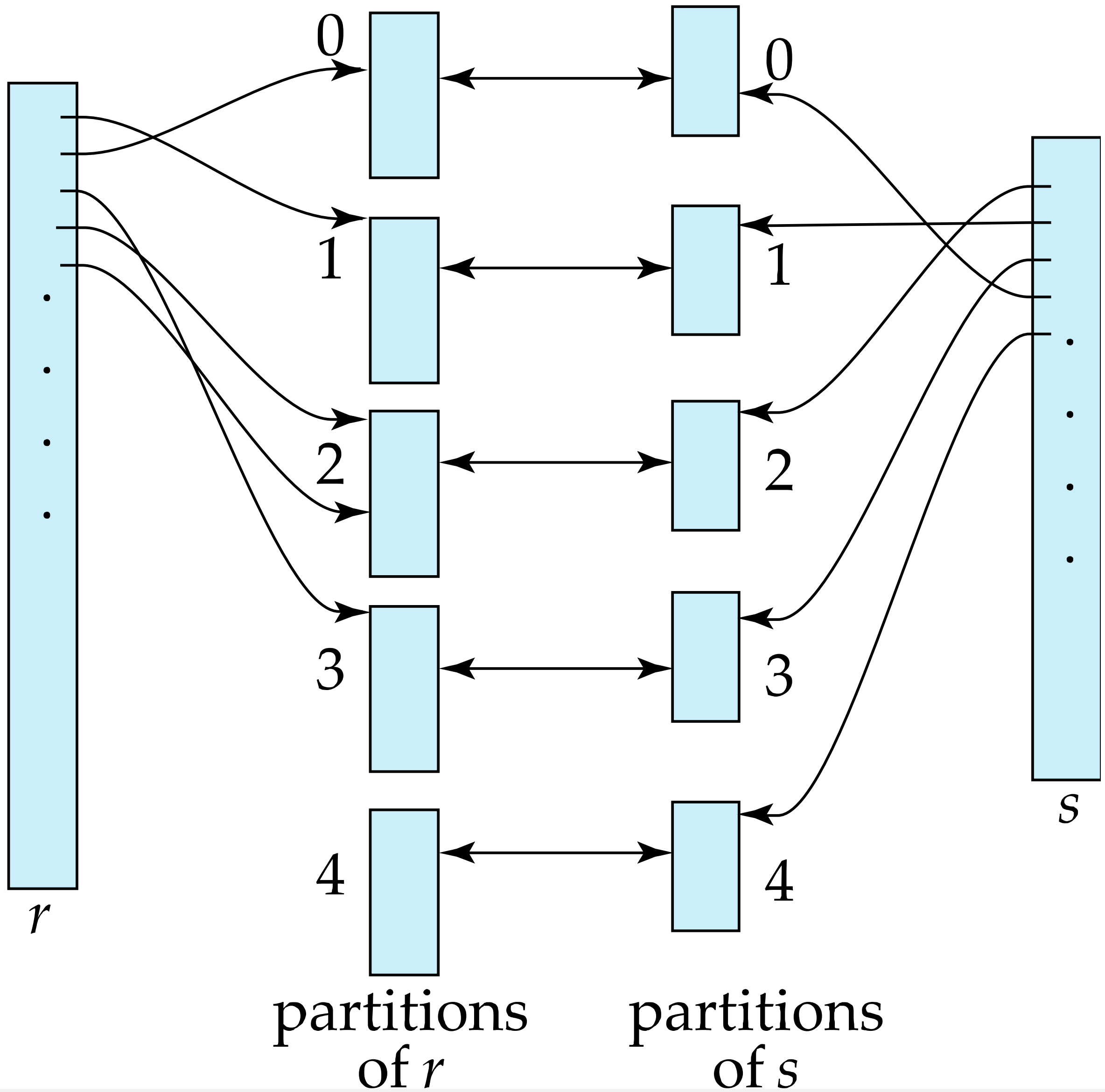
  $b_r + b_s$ block transfers

  - \+ the cost of sorting if relations are unsorted.

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function h is used to partition tuples of both relations
- h maps *JoinAttrs* values to {0, 1, ..., n}, where *JoinAttrs* denotes the common attributes of r and s used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of r tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1, \ldots, s_n$ 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*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *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 locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

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

Cost of Hash-Join

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

  $$3(b_r + b_s) + \alpha \text{ block transfers} + 2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \text{ seeks}$$

  - b_b : # of blocks allocated for input/output buffer

  - Partitioning : $2(b_r + b_s)$
  - build and probe phase: $(b_r + b_s)$
  - α is an overhead for partially filled blocks

- 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

- *instructor* ⋈ *teaches*

- Assume that memory size is 20 blocks
- $b_{instructor}$= 100 and $b_{teaches}$ = 400.
- *instructor* 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 *teaches* 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
    - b_b : # of blocks allocated for input = 3
      # of blocks allocated for output = 5

Complex Joins

- Join with a conjunctive condition:

$$r \bowtie_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simplest joins $r \bowtie_{\theta i} s$
    - final result comprises those tuples in the intermediate result that satisfy the remaining conditions

$$\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$

- Join with a disjunctive condition

$$r \bowtie_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta i} s$:

$$(r \bowtie_{\theta 1} s) \cup (r \bowtie_{\theta 2} s) \cup \ldots \cup (r \bowtie_{\theta n} s)$$

Other Operations

- **Duplicate elimination (distinct)** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - Hashing is similar – duplicates will come into the same bucket.

- **Projection:**
  - perform projection on each tuple and eliminate duplicate records by the method described above.

Other Operations : Set Operations

- **Set operations** ($\cup$, $\cap$ and —): can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - $r \cup s$:
          1. Add tuples in s_i to the hash index if they are not already in it.
          2. At end of s_i, add the tuples in the hash index to the result.
        - $r \cap s$:
          1. output tuples in s_i to the result if they are already there in the hash index
        - $r - s$:
          1. for each tuple in s_i, if it is there in the hash index, delete it from the index.
          2. At end of s_i add remaining tuples in the hash index to the result.

Other Operations : Outer Join

- **Outer join** can be computed either as
  - A join followed by addition of null-padded non-participating tuples.
- Modifying merge join to compute r ⟕ s
  - During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls.
  - Right outer-join and full outer-join can be computed similarly.
- Modifying hash join to compute r ⟕ s
  - If r is probe relation, output non-matching r tuples padded with nulls
  - If r is build relation, when probing keep track of which r tuples matched s tuples. At end of s_i output non-matched r tuples padded with nulls

Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - *E.g.) **select** dept_name, **avg**(salary)*
    ***from** instructor*
    ***group** by dept_name*

  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - As the groups are being constructed, apply the aggregation operations on the fly.
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the aggregates
    - For avg, keep sum and count, and divide sum by count at the end

Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an expression containing multiple operations
  - **Materialization**: generate results of an expression and reuse
  - **Pipelining**: evaluate several operations simultaneously

Materialization

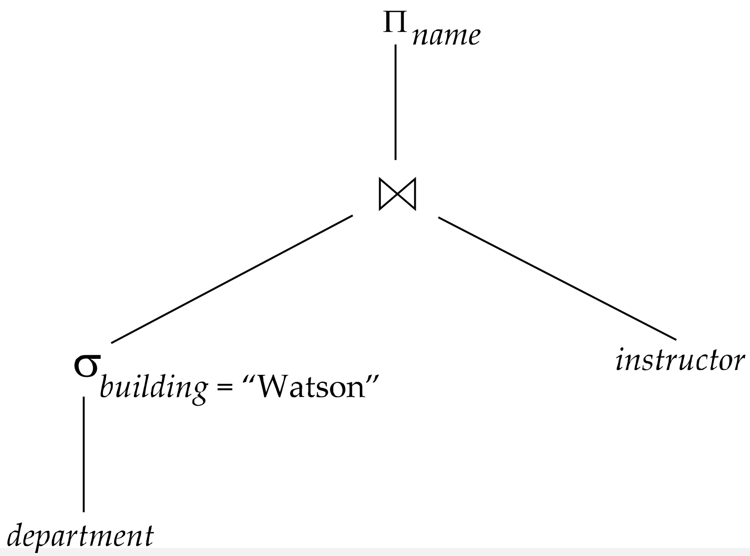
- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.

  ***Materialize: store a temporary relation to disk.*

- E.g., in figure below, compute and store

- $$\sigma_{building="\text{Watson}"}(department)$$

  then compute its join with *instructor,* and finally compute the projection on *name.*



Materialization (Cont.)

- Materialized evaluation is always applicable
  - Cost of writing results to disk and reading them back can be quite high

- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- **Pipelined evaluation :** evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{building = "\text{Watson}"}(department)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.

- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

Pipelining (Cont.)

- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain state so it knows what to return next
- In **producer-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining