

Database Development and Design (CPT201)

Lecture 4c: Query Evaluation - Join

Dr. Wei Wang
Department of Computing

Learning Outcomes

- Algorithms for evaluating join operators
- Algorithms for evaluating other expressions

Natural-Join Operation

Notation: $r \bowtie s$

- Let r and s be relations on schemas R and S respectively.
Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
 - Consider each pair of tuples t_r from r and t_s from s .
 - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
 - t has the same value as t_r on r
 - t has the same value as t_s on s
- Example:

$R = (A, B, C, D)$

$S = (E, B, D)$

- Result schema = (A, B, C, D, E)
- $r \bowtie s$ is defined as:

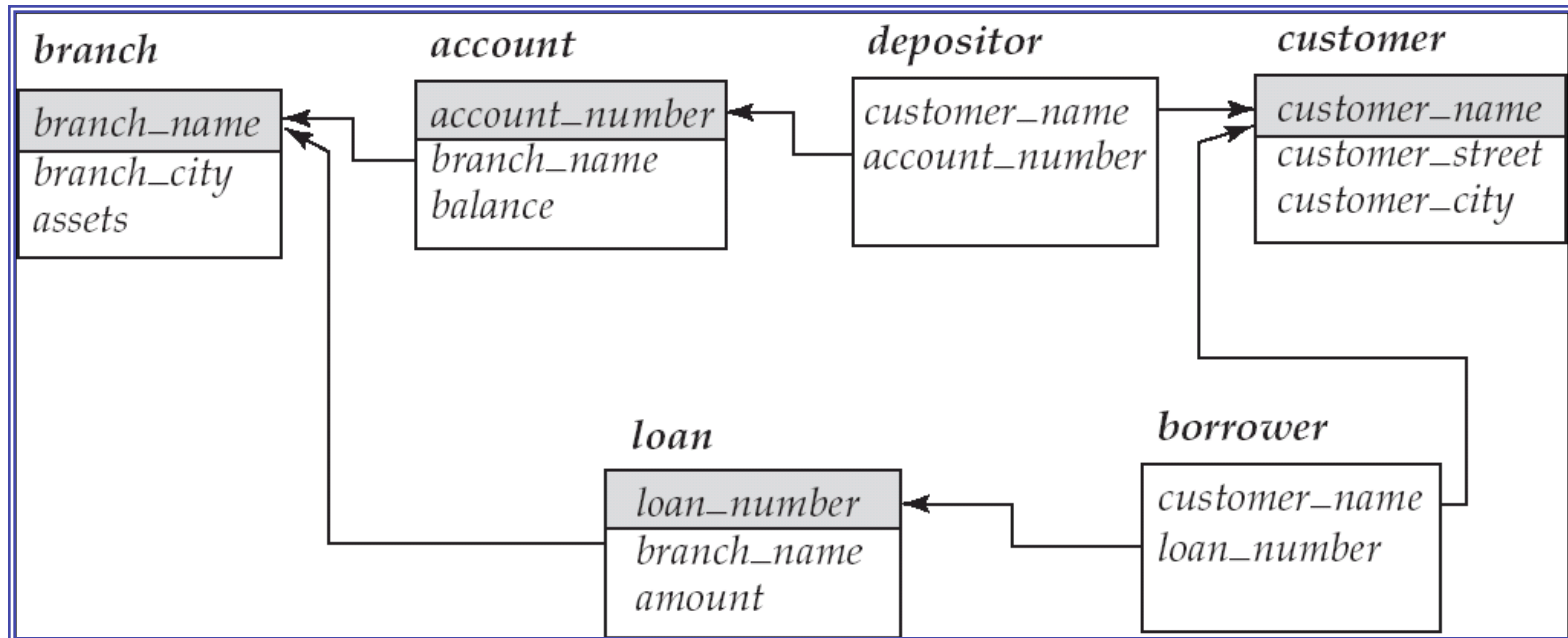
$$\Pi_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$



Join Operation

- Several different algorithms to implement joins exist (not counting with the ones involving parallelism)
 - Nested-loop join
 - Block nested-loop join
 - Indexed nested-loop join
 - Merge-join
 - Hash-join
- As for selection, the choice is based on cost estimate.

Banking Example



- Examples in next slides use the following information:
 - Number of records of *customer*: 10,000
 - Number of blocks of *customer*: 400
 - Number of records of *depositor*: 5,000
 - Number of blocks of *depositor*: 100

Nested-Loop Join

- The simplest join algorithms, that can be used **independently** of everything (like the linear search for selection)
- 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 $t_r \cdot t_s$ to the result.
 end
end
- r is called the **outer relation** and s the **inner relation** of the join.
- Quite expensive in general, since it requires to examine every pair of tuples in the two relations.

Nested-Loop Join cont'd

- In the worst case, if there is enough memory **only** to hold one block of each relation, n_r is the number of tuples in relation r , the estimated cost is
 $n_r * b_s + b_r$ block transfers, plus
 $n_r + b_r$ seeks
- 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
- But in general, it is much better to have the **smaller** relation as the **outer** relation
- The choice of the inner and outer relation strongly depends on the estimate of the size of each relation.

Nested-Loop Join Cost in Example

- Assuming **worst case** memory availability cost estimate is
 - with *depositor* as outer relation:
 - $5,000 * 400 + 100 = 2,000,100$ block transfers,
 - $5,000 + 100 = 5,100$ seeks
 - with *customer* as the outer relation
 - $10,000 * 100 + 400 = 1,000,400$ block transfers and 10,400 seeks
- If smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 block transfers and 2 seeks
- Instead of iterating over records, one could iterate over blocks. This way, instead of $n_r * b_s + b_r$ we would have $b_r * b_s + b_r$ block transfers
- This is the basis of the block nested-loops algorithm.

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  $t_r \cdot t_s$  to the result.
```

end

end

end

end

Block Nested-Loop Join Cost

- Worst case estimate: $b_r * b_s + b_r$ block transfers and $2 * b_r$ seeks
 - Each block in the inner relation s is **read once** for each *block* in the outer relation (instead of once for each tuple in the outer relation).
- Best case (when smaller relation fits into memory): $b_r + b_s$ block transfers plus 2 seeks.
- Some improvements to nested loop and block nested loop algorithms can be made:
 - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer, reduce the number of disk access
 - Use index on inner relation (if available) to quickly get the tuples which match the tuple of the outer relation.

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
 - In some cases, it pays to construct an index just to compute a join.
- For each tuple t_r in the outer relation r , use the index on s 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: $b_r + n_r * c$ block transfers and seeks
 - Where c is the cost of traversing index and fetching all matching s tuples for one tuple in r
 - c can be estimated as cost of a single selection on s using the join condition (usually quite low, when compared to the join)
- 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

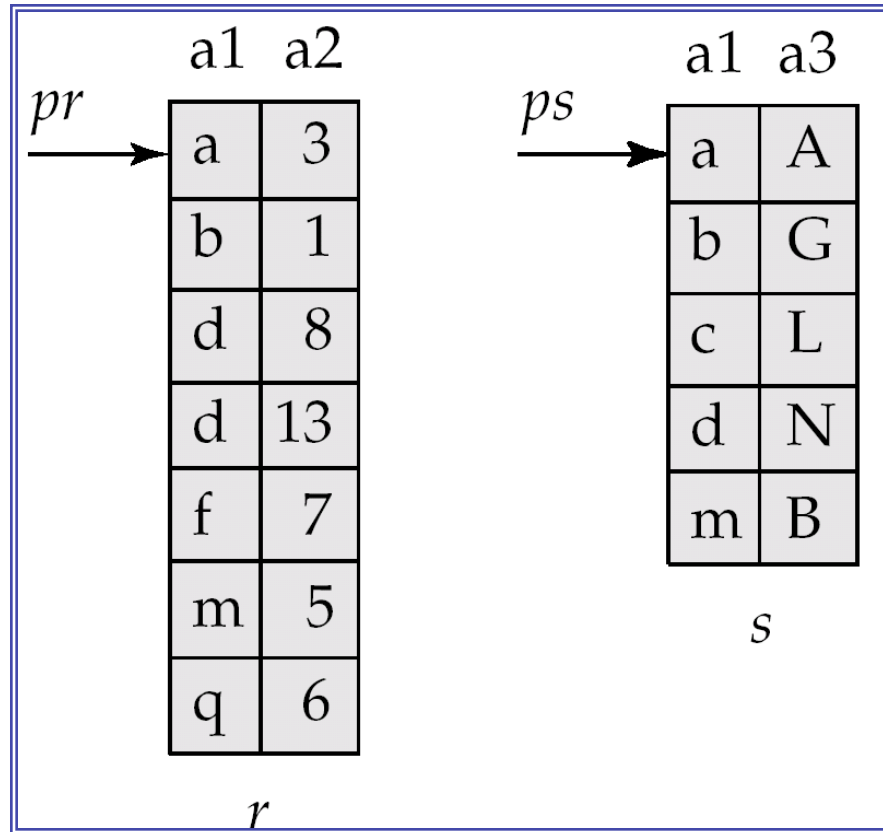
- Compute *depositor* ⋈ *customer*, with *depositor* as the **outer** relation.
- Let *customer* have a primary B⁺-tree index on the join attribute *customer-name*, with $n=20$.
- Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *depositor* has 5,000 tuples
- **Nested loop join**: 2,000,100 block transfers and 5,100 seeks
- Cost of **block nested loops join**
 - $400 \times 100 + 100 = 40,100$ block transfers + $2 \times 100 = 200$ seeks
- Cost of **indexed nested loops join**
 - $100 + 5,000 \times (4+1) = 25,100$ block transfers and seeks.
 - The number of block transfers is less than that for block nested loops join
 - But number of seeks is much larger
 - In this case using the index **doesn't pay** (this is specially so because the relations are small)

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
Join step is similar to the merge stage of the sort-merge algorithm.
2. Merge-join algorithm
 1. Initialise two pointers point to r and s
 2. While not done
 1. the pointers to r and s move through the relation.
 2. A group of tuples of inner relation s with the same value on the join attributes is read into S_s .
 3. Do join on tuple pointed by p_r and tuples in S_s ;
 3. End while

Merge-Join cont'd

Read pseudocode in the textbook!



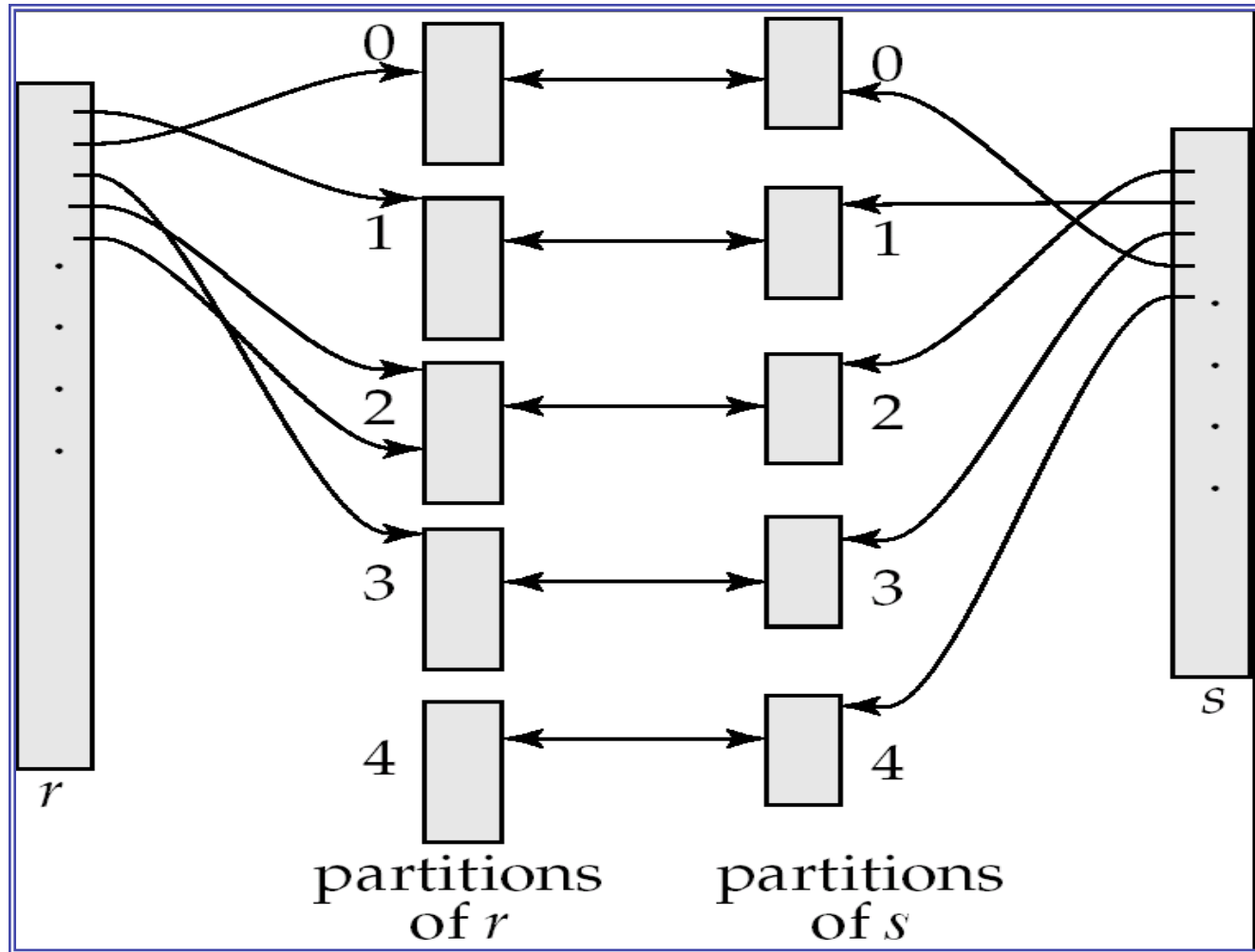
Merge-Join cont'd

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming that all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is (where b_b is the number of blocks in allocated in memory for each relation):
 - $b_r + b_s$ block transfers +
 $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks
 - **Plus** the cost of sorting if relations are unsorted.
 - Since seeks are much more expensive than data transfer, it makes sense to allocate multiple buffer blocks to each relation, provided extra memory is available.

Hash-Join

- Also only applicable for equi-joins and natural joins.
- A hash function h is used to partition tuples of both relations
- h maps $JoinAttrs$ values to $\{0, 1, \dots, n\}$, where $JoinAttrs$ denotes the common attributes of r and s used in the natural join.
 - r_0, r_1, \dots, 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, \dots, s_n denotes partitions of s tuples
 - Each tuple $t_s \in s$ is put in partition s_i , where $i = h(t_s[JoinAttrs])$.
- General idea:
 - Partition the relations according to this
 - Then perform the join on each partition r_i and s_i
 - There is no need to compute the join between different partitions 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 cont'd



Hash-Join Algorithm

1. Partition the relation s using hashing function h .
When partitioning a relation, some blocks of memory (b_b) are 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 h for partitioning.
 - (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**.



Hash-Join algorithm cont'd

- The number of partitions n for 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, to avoid overflows
 - The probe relation partitions r_i need not fit in memory

Cost of Hash-Join

- The cost of hash join is $3(b_r + b_s) + 4 * n_h$ block transfers, and $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2 * n_h$ seeks
 - each of the n_h partitions could have a partially filled block that has to be written and read back
 - The build and probe phases require only one seek for each of the n_h partitions of each relation, since each partition can be read sequentially.
- If the entire build input can be kept in main memory (then no partitioning is required), Cost estimate goes down to $b_r + b_s$ and 2 seeks.

Cost of Hash-Join in Example

- For the running example, assume that memory size is 20 blocks $b_{\text{ depositor}} = 100$ and $b_{\text{ customer}} = 400$.
- **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.
- Assuming 3 blocks are allocated for the input buffer and each output buffer
- Therefore total cost, ignoring cost of writing partially filled blocks:
$$3(100 + 400) = 1,500 \text{ block transfers} +$$
$$2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) + 2*5 = 344 \text{ seeks}$$
- We had up to here:
 - 40,100 block transfers plus 200 seeks (for block nested loop)
 - 25,100 block transfers and seeks (for index nested loop).

Other Operations: Duplicate Elimination

- **Duplicate elimination** 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.
 - *Optimisation:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
 - Hashing is similar - duplicates will come into the same bucket.
- **Projection:**
 - perform projection on each tuple;
 - followed by duplicate elimination.

Other Operations: Aggregation

- **Aggregation** can be implemented similarly to duplicate elimination.
 - 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.
 - *Optimisation*: combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
 - 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

Other Operations: Set Operations

- **Set operations** (\cup , \cap and $-$): can either use variant of merge-join after sorting, or variant of hash-join.
- 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 in it.
 2. At the end, 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 in the hash index
 - $r - s$:
 1. for each tuple in s_i , if it is in the hash index, delete it from the index.
 2. At the end, add remaining tuples in the hash index to the result.

End of Lecture

■ Summary

- Join
 - Nested-Loop Join
 - Block-Nested-Loop Join
 - Indexed-Nested-Loop Join
 - Sorted-Merge-Join
 - Hash Join
- Other Operations

■ Reading

- 6th edition, Chapters 12.5 and 12.6
- 7th edition, Chapters 15.5 and 15.6