Information Systems

Chapter 5:

Query Processing

Rita Schindler | TU Ilmenau, Germany
www.tu-ilmenau.de/dbis

Query Processing

- Basic Steps
- Cost Model
- Join Operations
- Other Operations
- Evaluation of Expressions

Query Processing and Optimization

Task

- Find an optimal (efficient) evaluation plan for a descriptive (SQL) query without accessing data files (except data dictionary tables)

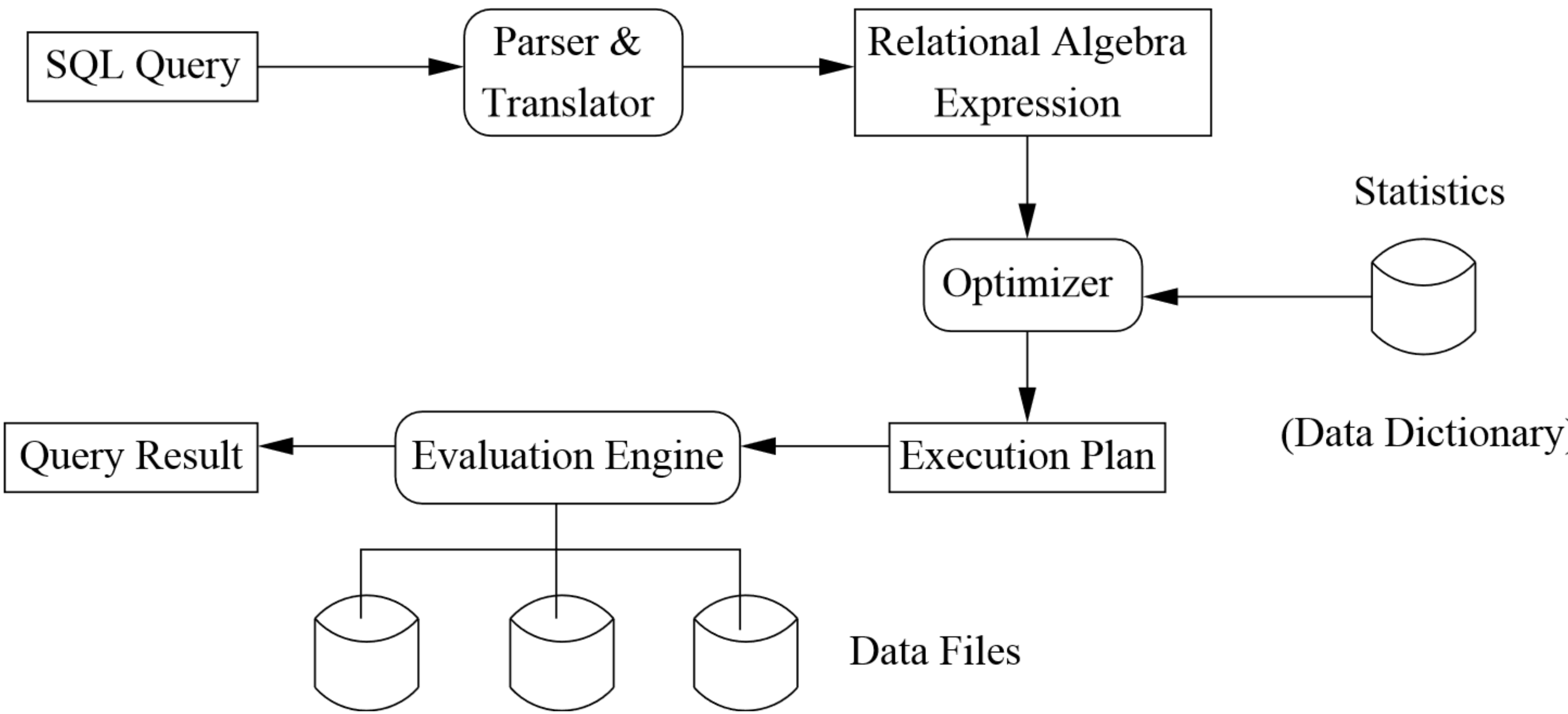
Goal

- Minimize the *evaluation time* for a query, i.e., compute query result as fast as possible

Cost Factors

- Disk accesses, read/write operations, [I/O, page transfer] (CPU time is typically ignored)

Basic Steps in Processing an SQL Query



Basic Steps in Processing an SQL Query (2)

- **Parsing and Translating**
  - Translate the query into its internal form (parse tree).
  - This is then translated into an expression of the relational algebra.
  - Parser checks syntax, validates relations, attributes and access permissions
- **Optimization**
  - Find the "cheapest" evaluation plan for a query
- **Evaluation**
  - The query-execution engine takes a query evaluation (or query execution) plan, executes the plan, and returns the result.

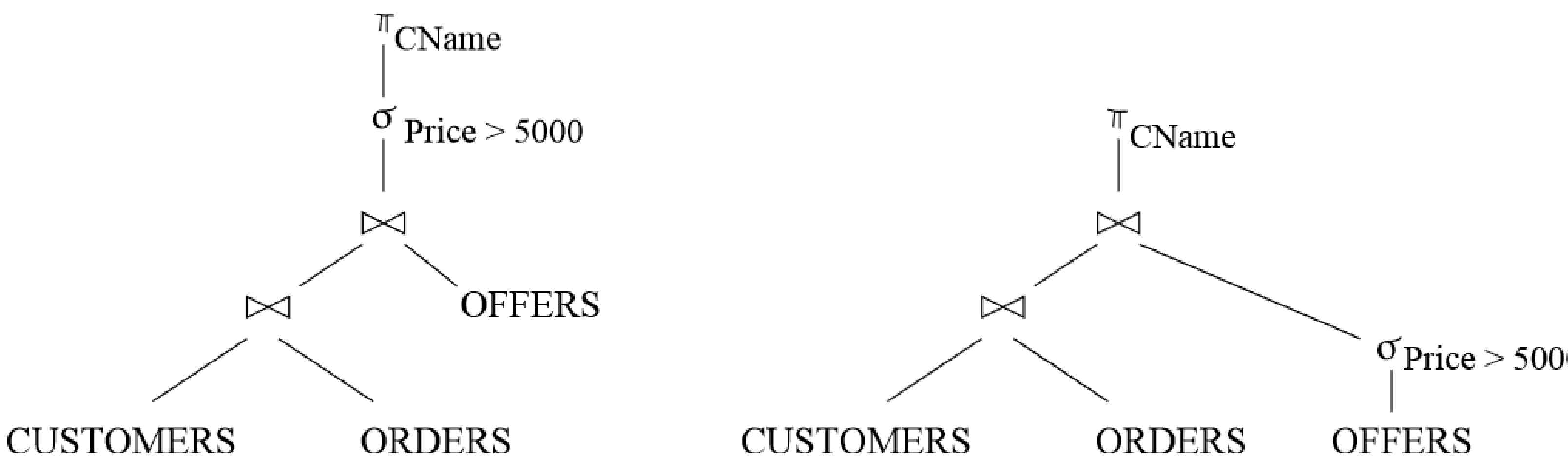
Basic Steps in Processing an SQL Query (3)

- A relational algebra expression may have many equivalent expressions, e.g.,

$$\pi_{CName}(\ \sigma_{Price>5000}\ (\ (\ CUSTOMERS \bowtie ORDERS\ ) \bowtie OFFERS\ )\ )$$

$$\pi_{CName}\ (\ (\ CUSTOMERS \bowtie ORDERS\ ) \bowtie (\ \sigma_{Price>5000}\ OFFERS\ )\ )$$

Representation as **evaluation plan** (query tree):



Non-leaf nodes ≡ operations of relational algebra (with parameters);
Leaf nodes ≡ relations

Basic Steps in Processing an SQL Query (4)

- A relational algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called **evaluation plan** (includes, e.g., whether index is used, algorithm for natural join, . . .)
- Among all semantically equivalent expression, the one with the least costly evaluation plan is chosen.
- Cost estimate of a plan is based on **statistical information** in the catalog (aka data dictionary).

Catalog Information for Cost Estimation

Information about relations and attributes:

- N_R: number of tuples in the relation R
- B_R: number of blocks that contain tuples of the relation R
- S_R: size of a tuple of R
- F_R: blocking factor; number of tuples from R that fit into one block ($F_R = [N_R/B_R]$)
- $V(A, R)$: number of distinct values for attribute A in R
- $SC(A, R)$: selectivity of attribute A ≡ avg. number of tuples of R that satisfy an equality condition on A; $SC(A, R) = N_R/V(A, R)$ for non-key attributes

Information about indexes:

- HT_I: number of levels in index I (B^+-tree).
- LB_I: number of blocks occupied by leaf nodes in index I (first-level blocks).
- Val_I: number of distinct values for the search key.

Measures of Query Cost

- There are many possible ways to estimate cost, e.g., based on disk accesses, CPU time, or communication overhead.
- Disk access is the predominant cost (in terms of time);
  - relatively easy to estimate; therefore, number of block transfers from/to disk is used as measure.
  - Assumption: Each block transfer has the same cost.
- Cost of algorithm (e.g., for join or selection) depends on database buffer size;
  - more memory for DB buffer reduces disk accesses.
  - Thus DB buffer size is a parameter for estimating cost.
- We refer to the cost estimate of algorithm A as cost(A). We do not consider cost of writing output to disk.

Selection Operation

$\sigma_{A=a}(R)$, A attribute in R

- **File Scan** – search algorithms that locate and retrieve records that satisfy a selection condition
- **S1** – Linear search

  If A is primary key then cost$(\mathrm{S}1)$ = $B_R/2$,
  otherwise cost$(\mathrm{S}1)$= B_R.

Selection Operation (2)

$\sigma_{A=a}(R)$, A attribute in R

- **S2** – Binary search,
  i.e., the file ordered based on attribute A (primary index)

$$cost(S2) = \lceil \log_2(B_R) \rceil + \left\lceil \frac{SC(A,R)}{F_R} \right\rceil - 1$$

  - $\lceil \log_2(B_R) \rceil$ ≡ cost to locate the first tuple using binary search
  - Second term ≡ blocks that contain records satisfying the selection.
  - If A is primary key and SC(A, R) = 1, then cost(S2) = $\lceil \log_2(B_R) \rceil$.
  - In case of a uniform distribution of attribute values for A, selection $\sigma_{A=a}(R)$, retrieves $SC(A, R) = N_R / V(A, R)$ tuples.

Selection Operation: Example

- $F_{\text{Employee}} = 10$;
  $V(\text{Deptno}, \text{Employee}) = 50$ (different departments)
- $N_{\text{Employee}} = 10{,}000$ (Relation Employee has 10,000 tuples)
- Assume selection $\sigma_{\text{Deptno}=20}(\text{Employee})$ and Employee is sorted on search key Deptno :

  $\Rightarrow$

  - 10,000/50 = 200 tuples in Employee belong to Deptno 20 (asuming an equal distribution)
  - 200/10 = 20 blocks for these tuples
  - A binary search finding the first block would require $\lceil \log_2(1,000) \rceil = 10$ block accesses (assuming that the header block of the file maintains a list of file blocks)

Total cost of binary search is 10+20 block accesses (versus 1,000 for linear search and Employee not sorted by Deptno)

Selection Operation (3)

Index scan – search algorithms that use an index (based on a B^+-tree); selection condition is on search key of index

- **S3** – Primary index I for A, A primary key, equality A = a
  $cost(S3) = HT_I + 1$ (only 1 tuple satisfies condition)
- **S4** – Primary index I on non-key A, equality A = a
  $$cost(S4) = HT_I + \left\lceil \frac{SC(A,R)}{F_R} \right\rceil$$
- **S5** – Non-primary (non-clustered) index on non-key A, equality A = a
  $cost(S5) = HT_I + SC(A, R)$
  Worst case: each matching record resides in a different block.

Selection Operation: Example (2)

- Assume primary (B^+-tree) index for attribute Deptno
- 200/10=20 blocks accesses are required to read Employee tuples
- If B^+-tree index stores 20 pointers per (inner) node, then the B^+-tree index must have between 3 and 5 leaf nodes and the entire tree has a depth of 2
  $\Rightarrow$ a total of 22 blocks must be read.

Selections Involving Comparisons

Selections of the form $\sigma_{A \le v}(R)$ or $\sigma_{A \ge v}(R)$ are implemented using a file scan or binary search, or by using either a

- **S6** – A primary index on A, or
- **S7** – A secondary index on A (in this case, typically a linear file scan may be cheaper; but this depends on the selectivity of A)

Complex Selections

- General pattern:
  - Conjunction – $\sigma_{\wedge 1 \wedge \ldots \wedge \wedge n}$ (R)
  - Disjunction – $\sigma_{\wedge 1 \vee \ldots \vee \wedge n}$ (R)
  - Negation – $\sigma_{\neg \wedge}$ (R)
- The selectivity of a condition Θ_i is the probability that a tuple in the relation R satisfies Θ_i . If s_i is the number of tuples in R that satisfy Θ_i , then Θ_i's selectivity is estimated as s_i / N_R.

Join Operations

- There are several different algorithms that can be used to implement joins (natural-, equi-, condition-join)
  - Nested-Loop Join
  - Block Nested-Loop Join
  - Index Nested-Loop Join
  - Sort-Merge Join
  - Hash-Join
- Choice of a particular algorithm is based on cost estimate
- For this, join size estimates are required and in particular cost estimates for outer-level operations in a relational algebra expression.

Join Operations: Example

- Assume the query $\mathrm{CUSTOMERS} \bowtie \mathrm{ORDERS}$ (with join attribute only being CName)
- $\mathrm{N_{CUSTOMERS}} = 5{,}000$ tuples
- $\mathrm{F_{CUSTOMERS}} = 20$, i.e., $\mathrm{B_{CUSTOMERS}} = 5{,}000/20 = 250$ blocks
- $\mathrm{N_{ORDERS}} = 10{,}000$ tuples
- $\mathrm{F_{ORDERS}} = 25$, i.e., $\mathrm{B_{ORDERS}} = 400$ blocks
- $\mathrm{V(CName,\ ORDERS)} = 2{,}500$, meaning that in this relation, on average, each customer has four orders
- Also assume that CName in ORDERS is a foreign key on $\mathrm{CUSTOMERS}$

Estimating the Size of Joins

- The Cartesian product $R \times S$ results in $N_R * N_S$ tuples; each tuple requires $S_R + S_S$ bytes.
- If $\text{schema}(R) \cap \text{schema}(S)$ = primary key for R, then a tuple of S will match with at most one tuple from R. Therefore, the number of tuples in $R \bowtie S$ is not greater than N_S
- If $\text{schema}(R) \cap \text{schema}(S)$ = foreign key in S referencing R, then the number of tuples in $R \bowtie S$ is exactly N_S
- Other cases are symmetric.

Estimating the Size of Joins (2)

- In the example query CUSTOMERS ⋈ ORDERS, CName in ORDERS is a foreign key of CUSTOMERS; the result thus has exactly N_{ORDERS} =10,000 tuples
- If schema(R) ∩ schema(S) = { A } is not a key for R or S; assume that every tuple in R produces tuples in R ⋈ S. Then the number of tuples in R ⋈ S is estimated to be:

$$\frac{N_R * N_S}{V(A,S)}$$

- If the reverse is true, the estimate is

$$\frac{N_R * N_S}{V(A,R)}$$

and the lower of the two estimates is probably the more accurate one.

Estimating the Size of Joins (3)

- Size estimates for CUSTOMERS $\bowtie$ ORDERS without using information about foreign keys:
  - V(CName, CUSTOMERS) = 5,000, and V(CName, ORDERS) = 2,500
  - The two estimates are 5,000*10,000/2,500=20,000 and 5,000*10,000/5,000=10,000.
- We choose the lower estimate, which, in this case, is the same as our earlier computation using foreign key information.

Block Nested-Loop Join

- Evaluate the condition join $R \ltimes_C S$

```
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
            for each tuple t_S in B_S do
                check whether pair (t_R, t_S)
                satisfies join condition
                if they do, add t_R ∘ t_S to the result
end end end end
```

- R is called the *outer* and S the *inner* relation of the join.
- Requires no indexes and can be used with any kind of join condition.

Block Nested-Loop Join (2)

- Worst case: db buffer can only hold one block of each relation
  $\Rightarrow \mathrm{B_R} + \mathrm{B_R} * \mathrm{B_S}$ disk accesses.
- Best case: both relations fit into db buffer
  $\Rightarrow \mathrm{B_R} + \mathrm{B_S}$ disk accesses.
- If smaller relation completely fits into db buffer, use that as inner relation. Reduces the cost estimate to $\mathrm{B_R} + \mathrm{B_S}$ disk accesses.
- Some improvements of block nested-loop algorithm
  - If equi-join attribute is the key on inner relation, stop inner loop with first match
  - Use $M - 2$ disk blocks as blocking unit for outer relation, where M = db buffer size in blocks; use remaining two blocks to buffer inner relation and output.
    Reduces number of scans of inner relation greatly.
  - Scan inner loop forward and backward alternately, to make use of blocks remaining in buffer (with LRU replacement strategy)
  - Use index on inner relation, if available.

Index Nested-Loop Join

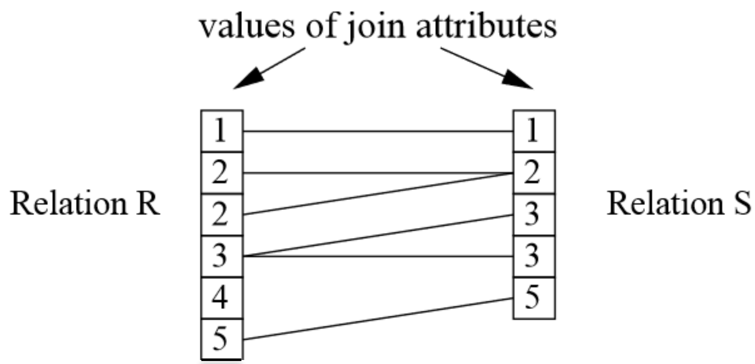
- If an index is available on the inner loop's join attribute and join is an equi-join or natural join, more efficient index lookups can replace file scans.
- It is even possible (reasonable) to construct index just to compute a join.
- For each tuple t_R in the outer relation R, use the index to lookup tuples in S that satisfy join condition with t_R
- Worst case: db buffer has space for only one page of R and one page of the index associated with S:
  - B_R disk accesses to read R, and for each tuple in R, perform index lookup on S.
  - Cost of the join: $B_R + N_R * c$, where c is the cost of a single selection on S using the join condition.
- If indexes are available on both R and S, use the one with the fewer tuples as the outer relation.

Index Nested-Loop Join (2)

- **Example:**
  - Compute $\text{CUSTOMERS} \bowtie \text{ORDERS}$, with CUSTOMERS as the outer relation.
  - Let ORDERS have a primary B+-tree index on the join-attribute CName, which contains 20 entries per index node
  - Since ORDERS has $10{,}000$ tuples, the height of the tree is 4, and one more access is needed to find the actual data records (based on tuple identifier).
  - Since $N_{\text{CUSTOMERS}}$ is $5{,}000$,
    the total cost is $250 + 5000 * 5 = 25{,}250$ disk accesses.
  - This cost is lower than the $100{,}250$ accesses needed for a block nested-loop join.

Sort-Merge Join

- Basic idea: first sort both relations on join attribute (if not already sorted this way)
- Join steps are similar to the merge stage in the external sort-merge algorithm (discussed later)
- Every pair with same value on join attribute must be matched.



- Each tuple needs to be read only once. As a result, each block is read only once. Thus, the number of block accesses is $B_R + B_S$ plus the cost of sorting, if relations are unsorted.
- Can be used only for equi-join and natural join

Sort-Merge Join (2)

- If one relation is sorted and the other has a secondary B^+-tree index on the join attribute, a *hybrid merge-join* is possible. The sorted relation is merged with the leaf node entries of the B^+-tree.
- The result is sorted on the addresses (rids) of the unsorted relation's tuples, and then the addresses can be replaced by the actual tuples efficiently.

Hash-Join

- Only applicable in case of equi-join or natural join
- A hash function is used to partition tuples of both relations into sets that have the same hash value on the join attribute

1. Partitioning Phase: $2 * (\mathrm{B_R} + \mathrm{B_S})$ block accesses
2. Matching Phase: $\mathrm{B_R} + \mathrm{B_S}$ block accesses (under the assumption that one partition of each relation fits into the database buffer)

Cost Estimates for other Operations

Sorting:

- If whole relation fits into db buffer $\rightarrow$ quick-sort
- Or, build index on the relation, and use index to read relation in sorted order.
- Relation that does not fit into db buffer $\rightarrow$ external sort-merge
  1. Phase: Create runs by sorting portions of the relation in db buffer
  2. Phase: Read runs from disk and merge runs in sort order

Cost Estimates for other Operations (2)

Duplicate Elimination:

- Sorting: remove all but one copy of tuples having identical value(s) on projection attribute(s)
- Hashing:
  - partition relation using hash function on projection;
  - then read partitions into buffer and create in-memory hash index;
  - tuple is only inserted into index if not already present

Cost Estimates for other Operations (3)

Set Operations:

- Sorting or hashing
- Hashing:
  - Partition both relations using the same hash function;
  - use in-memory index for partitions R_i
  - $R \cup S$: if tuple in R_i or in S_i, add tuple to result
  - $\cap$: if tuple in R_i and in S_i, $\ldots$
  - $-$: if tuple in R_i and not in S_i, $\ldots$

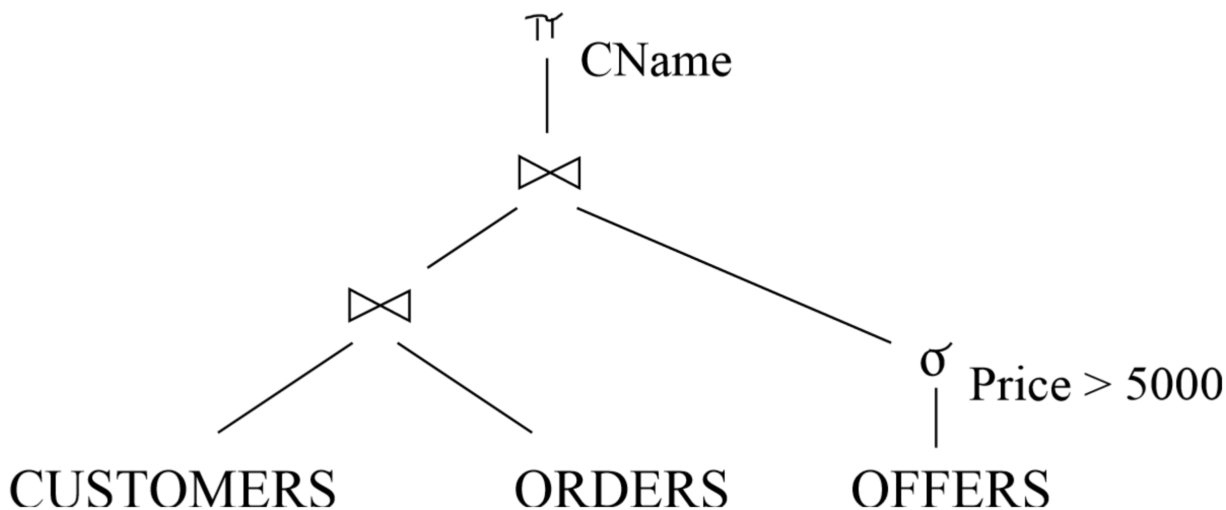
Cost Estimates for other Operations (4)

Grouping:

- Sorting or hashing.
- Hashing: while groups (partitions) are built, compute partial aggregate values (for group attribute A, $V(A,R)$ tuples to store values)

Evaluation of Expressions

- The basic concept is **materialization**: Evaluate one operation at a time, starting at the lowest level. Use intermediate results materialized in temporary relations to evaluate next level operation(s).



- First compute and store $\sigma_{\text{Price}>5000}(\text{OFFERS})$; then compute and store join of CUSTOMERS and ORDERS; finally, join the two materialized relations and project on to CName.

Evaluation of Expressions (2)

- **Pipelining**: evaluate several operations simultaneously, and pass the result (tuple- or block-wise) on to the next operation.
- In the example above, once a tuple from OFFERS satisfying selection condition has been found, pass it on to the join. Similarly, don't store result of (final) join, but pass tuples directly to projection.
- Much cheaper than materialization, because temporary relations are not generated and stored on disk.

Evaluation of Expressions (3)

- Pipelining is not always possible, e.g., for all operations that include sorting (**blocking operation**).
- Pipelining can be executed in either **demand driven** or **producer driven** fashion.

Transformation of Relational Expressions

- Generating a query-evaluation plan for an expression of the relational algebra involves two steps:
  1. generate logically equivalent expressions
  2. annotate these evaluation plans by specific algorithms and access structures to get alternative query plans
- Use **equivalence rules** to transform a relational algebra
- expression into an equivalent one.
- Based on estimated cost, the most cost-effective annotated plan is selected for evaluation. The process is called **cost-based query optimization**.

Equivalence of Expressions

- Result relations generated by two equivalent relational algebra expressions have the same set of attributes and contain the same set of tuples, although their attributes may be ordered differently.

Equivalence Rules (for expr. E, E_1, E_2, cond. F_i)

Applying distribution and commutativity of relational algebra operations

1. $\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$
2. $\sigma_F(E_1\ [\cup, \cap, -]\ E_2) \equiv \sigma_F(E_1)\ [\cup, \cap, -]\ \sigma_F(E_2)$
3. $\sigma_F(E_1 \times E_2) \equiv \sigma_{F0}(\sigma_{F1}(E_1) \times \sigma_{F2}(E_2))$; $F \equiv F0 \wedge F1 \wedge F2$, Fi contains only attributes of E_i, $i = 1, 2$.
4. $\sigma_{A=B}(E_1 \times E_2) \equiv E_1 \underset{A=B}{\bowtie} E_2$
5. $\pi_{\mathbf{A}}(E_1\ [\cup, \cap, -]\ E_2) \equiv \pi_{\mathbf{A}}(E_1)\ [\cup, \cap, -]\ \pi_{\mathbf{A}}(E_2)$
6. $\pi_{\mathbf{A}}(E_1 \times E_2) \equiv \pi_{\mathbf{A1}}(E_1) \times \pi_{\mathbf{A2}}(E_2)$, with $\mathbf{Ai} = \mathbf{A} \cap \{$ attributes in $E_i\}$, $i = 1, 2$.
7. $E_1\ [\cup, \cap]\ E_2 \equiv E_2\ [\cup, \cap]\ E_1$

   $(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$ (the analogous holds for $\cap$)
8. $E_1 \times E_2 \equiv \pi_{\mathbf{A1,A2}}(E_2 \times E_1)$

   $(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$

   $(E_1 \times E_2) \times E_3 \equiv (E_1 \times E_3) \times E_2$
9. $E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$ $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$

Examples

- Selection:
  - Find the name of all customers who have ordered a product for more than $5,000 from a supplier located in Ilmenau.

$$\pi_{\text{CName}}\ (\sigma_{\text{SAddress like '\%Ilmenau\%'} \wedge \text{Price>5000}}$$
$$(\text{CUSTOMERS} \bowtie (\text{ORDERS} \bowtie (\text{OFFERS} \bowtie \text{SUPPLIERS}))))$$

  - Perform selection as early as possible
    (but take existing indexes on relations into account)

$$\pi_{\text{CName}}\ (\text{CUSTOMERS} \bowtie (\text{ORDERS} \bowtie (\sigma_{\text{Price>5000}}\ (\text{OFFERS})$$
$$\bowtie\ (\sigma_{\text{SAddress like '\%Ilmenau\%'}}\ (\text{SUPPLIERS})))))$$

Examples (2)

- Projection:
  - Find the name and account of all customers who have ordered a product 'CD-ROM'

$$\pi_{\text{CName, account}} \left(\text{CUSTOMERS} \bowtie \sigma_{\text{Prodname} = \text{'CD-ROM'}} (\text{ORDERS}) \right)$$

  - Reduce the size of argument relation in join

$$\pi_{\text{CName, account}} \left(\text{CUSTOMERS} \bowtie \pi_{\text{CName}} \left(\sigma_{\text{Prodname} = \text{'CD-ROM'}} (\text{ORDERS}) \right) \right)$$

Projection should not be shifted before selections, because minimizing the number of tuples in general leads to more efficient plans than reducing the size of tuples.

Join Ordering

- For relations R_1, R_2, R_3 applies $(R_1 \bowtie R_2) \bowtie R_3 \equiv R_1 \bowtie (R_2 \bowtie R_3)$
- If $(R_2 \bowtie R_3)$ is quite large and $(R_1 \bowtie R_2)$ is small, we choose $(R_1 \bowtie R_2) \bowtie R_3$ so that a smaller temporary relation is computed and materialized
  - Example: List the name of all customers who have ordered a product from a supplier located in Ilmenau.

$$\pi_{\text{CName}} (\sigma_{\text{SAddress like ‘\%Ilmenau\%‘}} (\text{SUPPLIERS} \bowtie \text{ORDERS} \bowtie \text{CUSTOMERS}))$$

ORDERS $\bowtie$ CUSTOMERS is likely to be a large relation. Because it is likely that only a small fraction of suppliers are from Ilmenau, we compute the join

$$\sigma_{\text{SAddress like ‘\%Ilmenau\%‘}} (\text{SUPPLIERS} \bowtie \text{ORDERS})$$

first.

Summary of Algebraic Optimization Rules

1. Perform selection as early as possible
2. Replace Cartesian Product by join whenever possible
3. Project out useless attributes early.
4. If there are several joins, perform *most restrictive* join first

Evaluation Plan

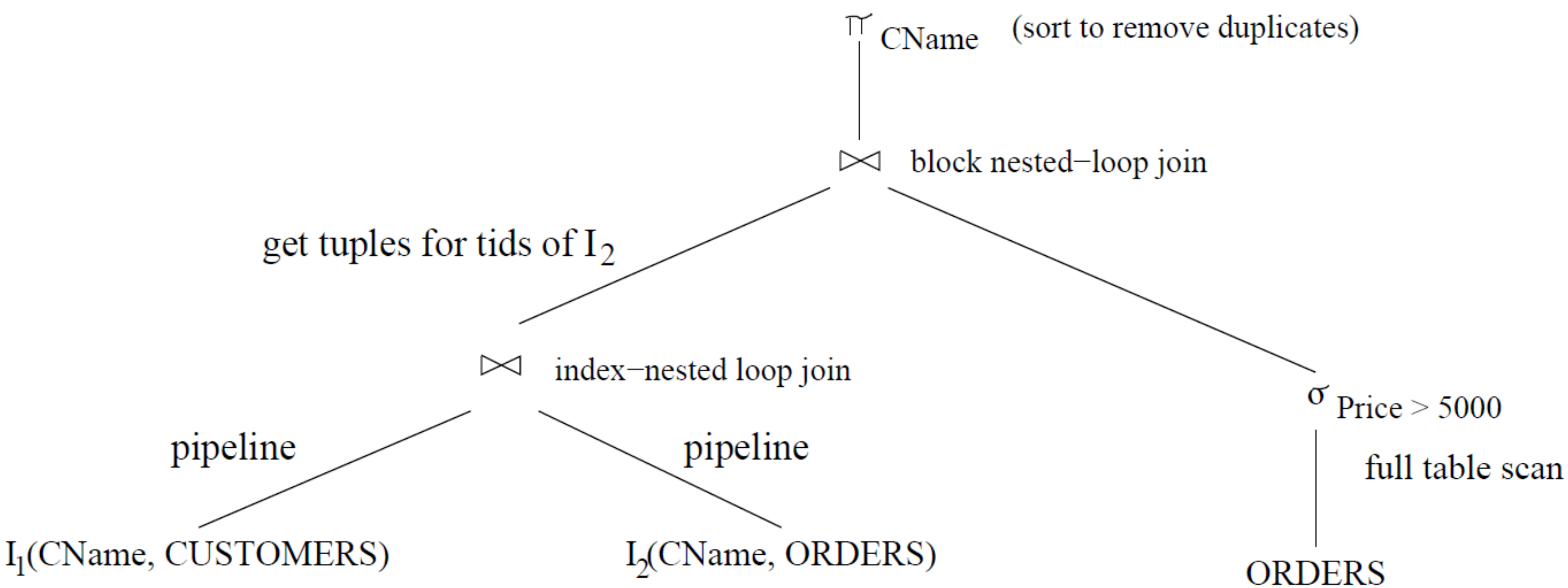
An evaluation plan for a query exactly defines

- what algorithm is used for each operation,
- which access structures are used (tables, indexes, clusters),
- and how the execution of the operations is coordinated.

Example of Annotated Evaluation Plan

- **Query: List the name of all customers who have ordered a product that costs more than $5,000.**

  Assume that for both CUSTOMERS and ORDERS an index on CName exists: I_1(CName, CUSTOMERS), I_2(CName, ORDERS).



Choice of an Evaluation Plan

- Must consider interaction of evaluation techniques when choosing evaluation plan: choosing the algorithm with the least cost for each operation independently may not yield the best overall algorithm.
- Practical query optimizers incorporate elements of the following two optimization approaches:
  - **Cost-based**: enumerate all the plans and choose the best plan in a cost-based fashion.
  - **Rule-based**: Use rules (heuristics) to choose plan.
- Remarks on cost-based optimization:
  - Finding a join order for $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$:

    There are $(2(n-1))!/(n-1)!$ different join orders;

    For example, for $n=7$, the number is 665280.

    $\rightarrow$ use of dynamic programming techniques

Choice of an Evaluation Plan (2)

- Heuristic (or rule-based) optimization transforms a given query tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces number of tuples)
  - Perform projection early (reduces number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.

Summary

- Steps of query processing: translate a SQL query into an execution plan
- Cost estimation based on statistics and cost formulas
- Physical query operators: scans, join implementations, etc.
- Evaluation of query expressions
- Query optimization: search strategy for finding the best (cheapest) plan

Example (1)

Customer (<u>Cname</u> , Caddr , Account)
Order (<u>Cname , Article</u> , Amount)

SELECT Customer.Cname , Account PROJ
FROM Customer , Order
WHERE Customer.Cname = Order.Cname
AND Article = 'Coffee' SEL

- Customer: 100 tuple ; 5 tuple / page
- Order: 10 000 tuple ; 10 tuple / page
- 50 orders related to coffee
- tuple (Customer.Cname , Account) 50 tuple / page
  $\Rightarrow$ result of SELECT … $\rightarrow$ 1 page
- tuple (Customer $\times$ Order) 3 rows / page
- Buffer: size = 1 (for each relation)