

Algorithms for Query Processing and Optimization

Outline

Introduction to Query Processing

Translating SQL Queries into Relational Algebra

Algorithms for External Sorting

Algorithms for SELECT and JOIN Operations

Using Heuristics in Query Optimization

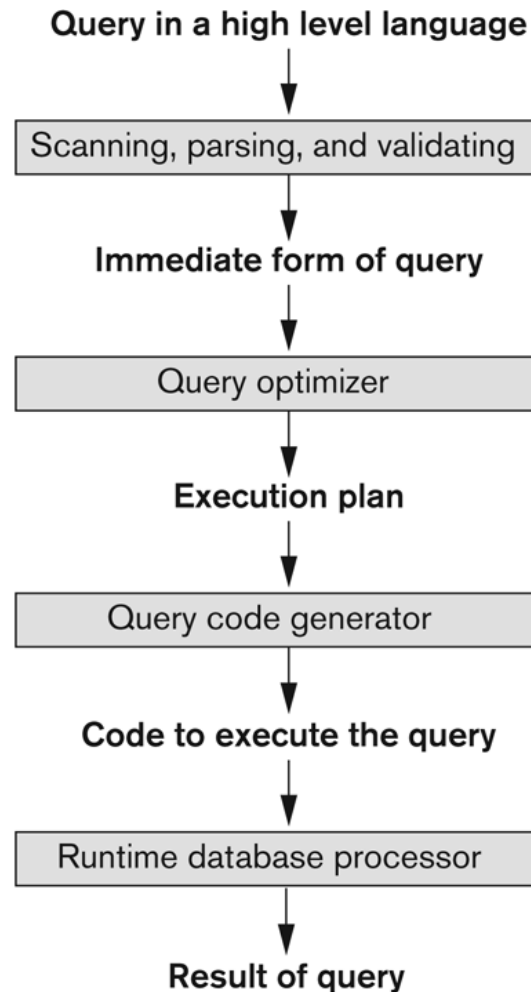
Using Selectivity and Cost Estimates in Query Optimization

Semantic Query Optimization

Introduction to Query Processing

- **Query optimization:**
 - The process of choosing a suitable execution strategy for processing a query.
- **Two internal representations of a query:**
 - **Query Tree**
 - **Query Graph**

Introduction to Query Processing



Code can be:

Executed directly (interpreted mode)
Stored and executed later whenever needed (compiled mode)

Figure 15.1

Typical steps when processing a high-level query.

Translating SQL Queries into Relational Algebra

- **Query block:**
 - The basic unit that can be translated into the algebraic operators and optimized.
- A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clause if these are part of the block.
- **Nested queries** within a query are identified as separate query blocks.
- Aggregate operators in SQL must be included in the extended algebra.

Translating SQL Queries into Relational Algebra

```
SELECT      LNAME, FNAME
FROM        EMPLOYEE
WHERE       SALARY > ( SELECT      MAX (SALARY)
                        FROM        EMPLOYEE
                        WHERE       DNO = 5);
```

```
SELECT      LNAME, FNAME
FROM        EMPLOYEE
WHERE       SALARY > C
```

$\pi_{\text{LNAME, FNAME}} (\sigma_{\text{SALARY} > C} (\text{EMPLOYEE}))$

```
SELECT      MAX (SALARY)
FROM        EMPLOYEE
WHERE       DNO = 5
```

$\mathcal{F}_{\text{MAX SALARY}} (\sigma_{\text{DNO}=5} (\text{EMPLOYEE}))$

Algorithms for External Sorting

- **External sorting:**

- Refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.

- **Sort-Merge strategy:**

- Starts by sorting small subfiles (**runs**) of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn.
- Sorting phase: $n_R = (b/n_B)$
- Merging phase: $d_M = \min(n_B - 1, n_R)$; $n_P = (\log_{d_M}(n_R))$
- n_R : number of initial runs; b : number of file blocks;
- n_B : available buffer space; d_M : degree of merging;
- n_P : number of passes.

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation
- Examples:
 - (OP1): $\sigma_{(SSN='123456789')}$ (EMPLOYEE)
 - (OP2): $\sigma_{(DNUMBER>5)}$ (DEPARTMENT)
 - (OP3): $\sigma_{(DNO=5)}$ (EMPLOYEE)
 - (OP4): $\sigma_{(DNO=5 \text{ AND } SALARY>30000 \text{ AND } GENDER=F)}$ (EMPLOYEE)
 - (OP5): $\sigma_{(ESSN=123456789 \text{ AND } PNO=10)}$ (WORKS_ON)

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation (contd.):
- Search Methods for Simple Selection:
 - **S1 Linear search** (brute force):
 - Retrieve every record in the file, and test whether its attribute values satisfy the selection condition.
 - **S2 Binary search:**
 - If the selection condition involves an equality comparison (eg:OP1) on a key attribute on which the file is ordered, binary search (which is more efficient than linear search) can be used.
 - **S3 Using a primary index or hash key to retrieve a single record:**
 - If the selection condition involves an equality comparison (eg:OP1) on a key attribute with a primary index (or a hash key), use the primary index (or the hash key) to **retrieve the record**.

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation (contd.):
- Search Methods for Simple Selection:
 - **S4 Using a primary index to retrieve multiple records:**
 - If the comparison condition is $>$, \geq , $<$, or \leq on a key field with a primary index (eg: OP2), use the index to find the record satisfying the corresponding equality condition, then **retrieve all subsequent records** in the (ordered) file.
 - **S5 Using a clustering index to retrieve multiple records:**
 - If the selection condition involves an equality comparison on a non-key attribute (ordered but having duplicate values) with a clustering index (eg: OP3), use the clustering index to **retrieve all the records** satisfying the selection condition.
 - **S6 Using a secondary (B+-tree) index:**
 - On an equality comparison, this search method can be used to retrieve a **single record** if the indexing field has unique values (is a key) or to retrieve multiple records if the indexing field is not a key.
 - In addition, it can be used to retrieve records on conditions involving $>$, \geq , $<$, or \leq . (FOR RANGE QUERIES)

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation:
 - Join (EQUIJOIN, NATURAL JOIN)
 - two-way join: a join on two files
 - e.g. $R \bowtie_{A=B} S$
 - multi-way joins: joins involving more than two files.
 - e.g. $R \bowtie_{A=B} S \bowtie_{C=D} T$
- Examples
 - (OP6): $\text{EMPLOYEE} \bowtie_{\text{DNO}=\text{DNUMBER}} \text{DEPARTMENT}$
 - (OP7): $\text{DEPARTMENT} \bowtie_{\text{MGRSSN}=\text{SSN}} \text{EMPLOYEE}$

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):
- Methods for implementing joins:
 - **J1 Nested-loop join** (brute force):
 - For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition $t[A] = s[B]$.
 - **J2 Single-loop join** (Using an access structure to retrieve the matching records):
 - If an index (or hash key) exists for one of the two join attributes — say, B of S — retrieve each record t in R , one at a time, and then use the access structure to retrieve directly all matching records s from S that satisfy $s[B] = t[A]$.

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):
- Methods for implementing joins:
 - **J3 Sort-merge join:**
 - If the records of R and S are *physically sorted (ordered)* by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible.
 - Both files are scanned in order of the join attributes, matching the records that have the same values for A and B.
 - In this method, the records of each file are scanned only once each for matching with the other file—unless both A and B are non-key attributes, in which case the method needs to be modified slightly.

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):
- Methods for implementing joins:
 - **J4 Hash-join:**
 - The records of files R and S are both hashed to the *same hash file*, using the *same hashing function* on the join attributes A of R and B of S as hash keys.
 - A single pass through the file with fewer records (say, R) hashes its records to the hash file buckets.
 - A single pass through the other file (S) then hashes each of its records to the appropriate bucket, where the record is combined with all matching records from R.

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):
- Other types of JOIN algorithms
- Partition hash join
 - Partitioning phase:
 - Each file (R and S) is first partitioned into M partitions using a partitioning hash function on the join attributes:
 - $R_1, R_2, R_3, \dots, R_m$ and $S_1, S_2, S_3, \dots, S_m$
 - Minimum number of in-memory buffers needed for the partitioning phase: $M+1$.
 - A disk sub-file is created per partition to store the tuples for that partition.
 - Joining or probing phase:
 - Involves M iterations, one per partitioned file.
 - Iteration i involves joining partitions R_i and S_i .

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):
- Partitioned Hash Join Procedure:
 - Assume R_i is smaller than S_i .
 1. Copy records from R_i into memory buffers.
 2. Read all blocks from S_i , one at a time and each record from S_i is used to *probe* for a matching record(s) from partition S_i .
 3. Write matching record from R_i after joining to the record from S_i into the result file.

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):
- Cost analysis of partition hash join:
 1. Reading and writing each record from R and S during the partitioning phase:
 $(b_R + b_S), (b_R + b_S)$
 2. Reading each record during the joining phase:
 $(b_R + b_S)$
 3. Writing the result of join:
 b_{RES}
- Total Cost:
 1. $3 * (b_R + b_S) + b_{RES}$

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):
- **Hybrid hash join:**
 - Same as partitioned hash join except:
 - Joining phase of one of the partitions is included during the partitioning phase.
 - **Partitioning phase:**
 - Allocate buffers for smaller relation- one block for each of the M-1 partitions, remaining blocks to partition 1.
 - Repeat for the larger relation in the pass through S.)
 - **Joining phase:**
 - M-1 iterations are needed for the partitions R2 , R3 , R4 ,Rm and S2 , S3 , S4 ,Sm. R1 and S1 are joined during the partitioning of S1, and results of joining R1 and S1 are already written to the disk by the end of partitioning phase.

Using Heuristics in Query Optimization

- Process for heuristics optimization
 1. The parser of a high-level query generates an initial internal representation;
 2. Apply heuristics rules to optimize the internal representation.
 3. A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.
- The main heuristic is to apply first the operations that reduce the size of intermediate results.
 - E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Using Heuristics in Query Optimization

- **Query tree:**

- A tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as **leaf nodes** of the **tree**, and represents the relational algebra operations as internal nodes.

- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.

- **Query graph:**

- A graph data structure that corresponds to a relational calculus expression. It does *not* indicate an order on which operations to perform first. There is only a *single* graph corresponding to each query.

Using Heuristics in Query Optimization

- Example:

- For every project located in 'Stafford', retrieve the project number, the controlling department number and the department manager's last name, address and birthdate.

- Relation algebra:


$$\bowtie_{\begin{matrix} PLOCATION='STAFFORD' \\ DNUM=DNUMBER \\ MGRSSN=SSN \end{matrix}} ((PROJECT) \bowtie (DEPARTMENT)) \bowtie (EMPLOYEE)$$

- SQL query:

```
Q2: SELECT      P.NUMBER,P.DNUM,E.LNAME,
      E.ADDRESS, E.BDATE
FROM    PROJECT AS P,DEPARTMENT AS D,      EMPLOYEE AS E
WHERE   P.DNUM=D.DNUMBER AND      D.MGRSSN=E.SSN AND
      P.PLOCATION='STAFFORD';
```

Using Heuristics in Query Optimization

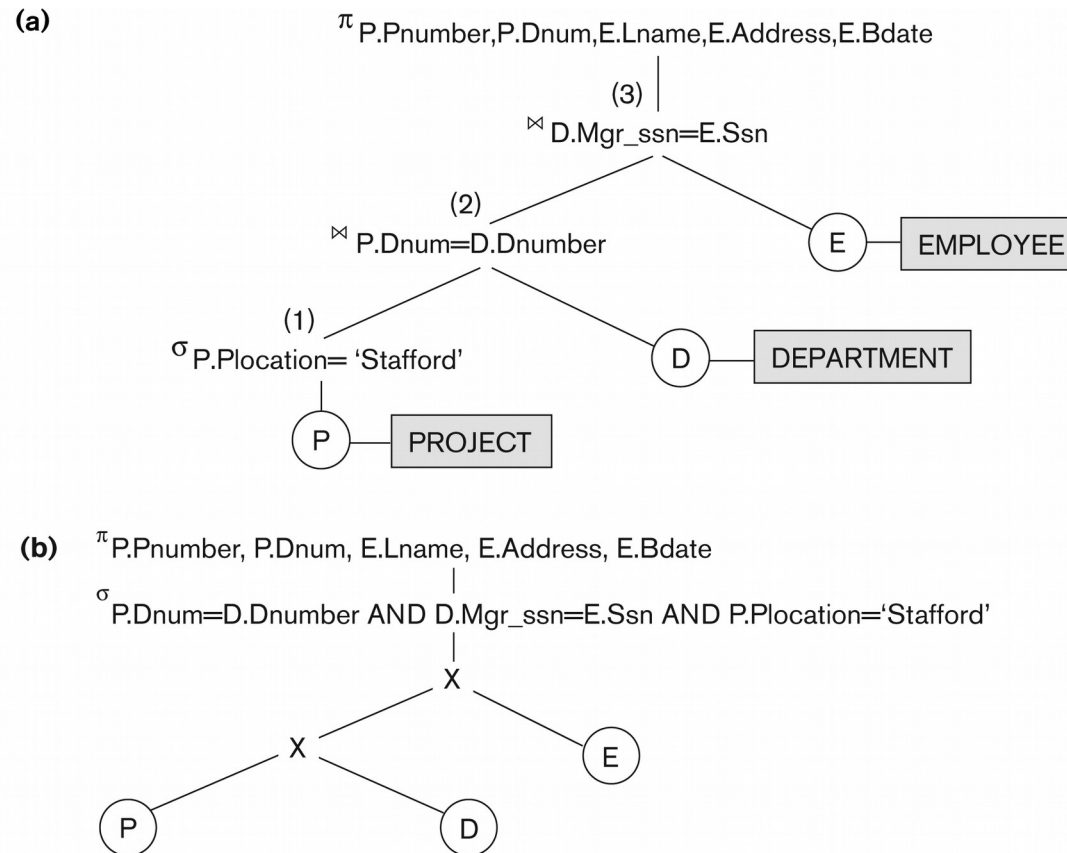


Figure 15.4

Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

Using Heuristics in Query Optimization

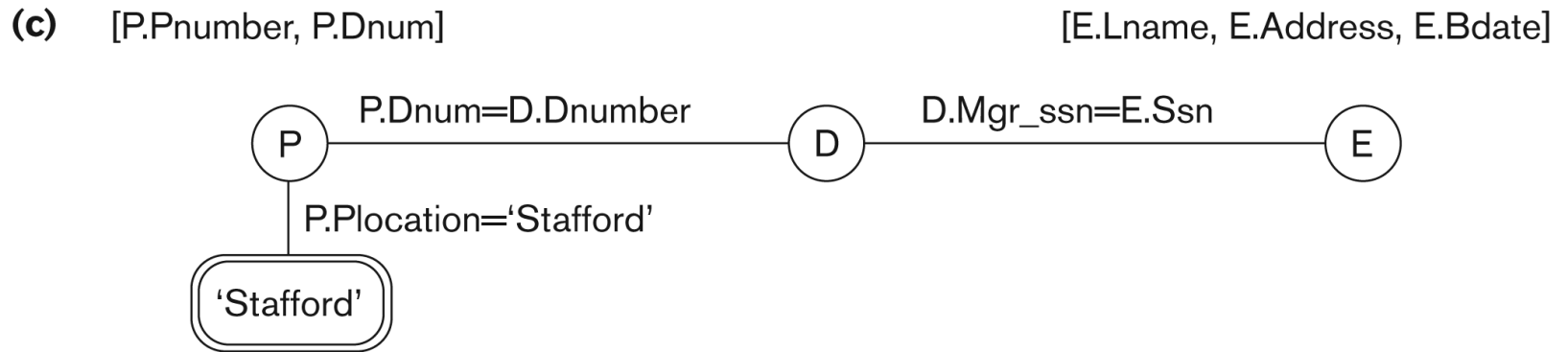


Figure 15.4

Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

Using Heuristics in Query Optimization

- Heuristic Optimization of Query Trees:
 - The same query could correspond to many different relational algebra expressions — and hence many different query trees.
 - The task of heuristic optimization of query trees is to find a **final query tree** that is efficient to execute.

- Example:

```
Q: SELECT      LNAME
    FROM        EMPLOYEE, WORKS_ON, PROJECT
    WHERE       PNAME = 'AQUARIUS' AND
                PNMUBER=PNO AND ESSN=SSN
                AND BDATE > '1957-12-31';
```


Using Heuristics in Query Optimization

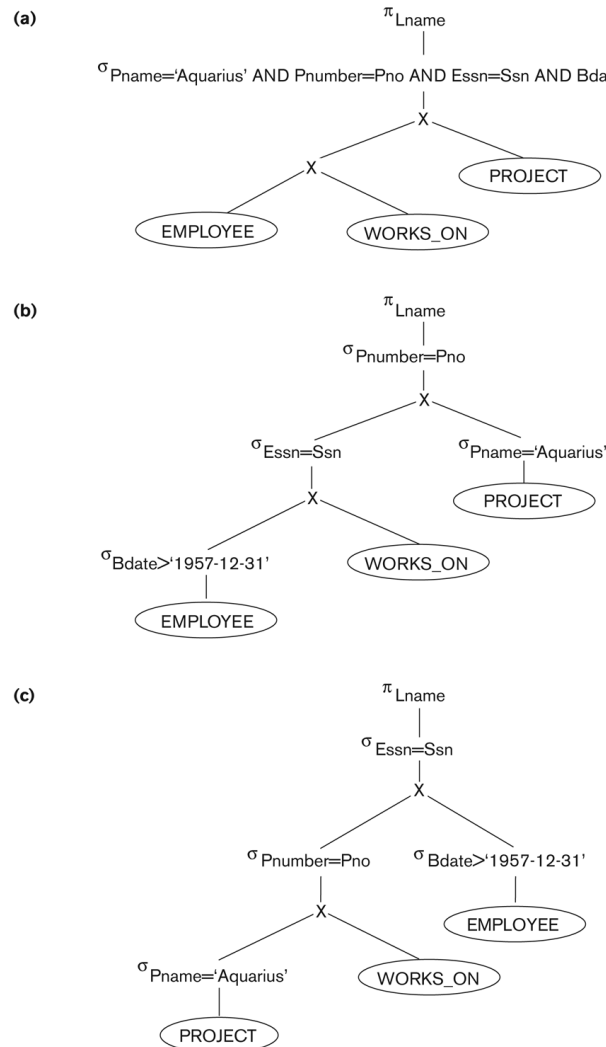


Figure 15.5

Steps in converting a query tree during heuristic optimization.

(a) Initial (canonical) query tree for SQL query Q.

(b) Moving SELECT operations down the query tree.

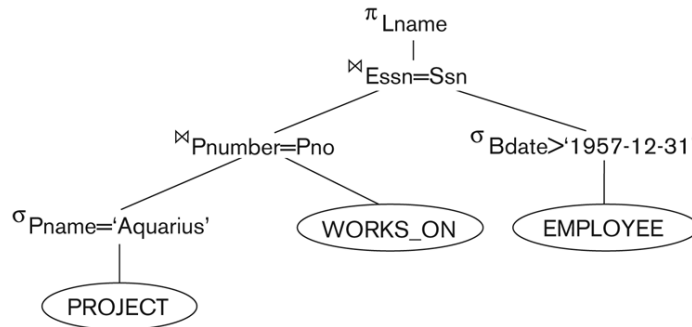
(c) Applying the more restrictive SELECT operation first.

(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.

(e) Moving PROJECT operations down the query tree.

Using Heuristics in Query Optimization

(d)



(e)

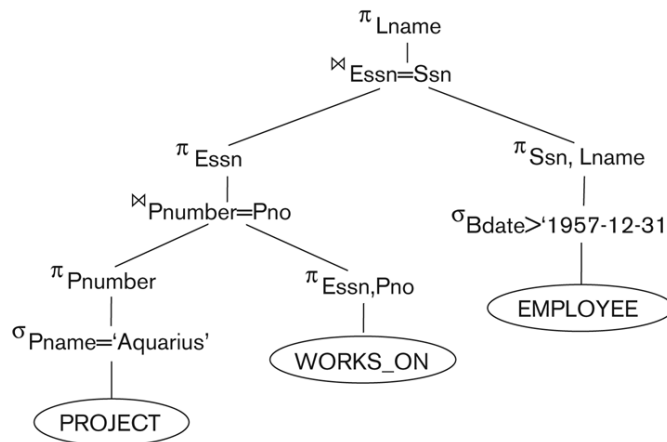


Figure 15.5

Steps in converting a query tree during heuristic optimization.

- (a) Initial (canonical) query tree for SQL query Q.
- (b) Moving SELECT operations down the query tree.
- (c) Applying the more restrictive SELECT operation first.
- (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
- (e) Moving PROJECT operations down the query tree.

Using Heuristics in Query Optimization (9)



■ General Transformation Rules for Relational Algebra Operations:

1. Cascade of : A conjunctive selection condition can be broken up into a cascade (sequence) of individual  operations:

$$\text{■ } \text{c1 AND c2 AND ... AND cn} (R) = \text{c1} \left(\text{c2} \left(\dots \left(\text{cn} (R) \right) \dots \right) \right)$$

2. Commutativity of : The  operation is commutative:

$$\text{■ } \text{c1} \left(\text{c2} (R) \right) = \text{c2} \left(\text{c1} (R) \right)$$

3. Cascade of : In a cascade (sequence) of  operations, all but the last one can be ignored:

$$\text{■ } \text{List1} \left(\text{List2} \left(\dots \left(\text{Listn} (R) \right) \dots \right) \right) = \text{List1} (R)$$

4. Commuting  with : If the selection condition c involves only the attributes A1, ..., An in the projection list, the two operations can be commuted:


$$\text{■ } \text{A1, A2, ..., An} \left(\text{c} (R) \right) = \text{c} \left(\text{A1, A2, ..., An} (R) \right)$$

Using Heuristics in Query Optimization

- General Transformation Rules for Relational Algebra Operations (contd.):

5. Commutativity of \bowtie (and \times): The \bowtie operation is commutative as is the \times operation:

- $R \bowtie_c S = S \bowtie_c R, R \times S = S \times R$

6. Commuting  with \bowtie (or \times): If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R —the two operations can be commuted as follows:

- $\sigma_c (R \bowtie S) = (\sigma_c (R)) \bowtie S$

- Alternatively, if the selection condition c can be written as $(c1 \text{ and } c2)$, where condition $c1$ involves only the attributes of R and condition $c2$ involves only the attributes of S , the operations commute as follows:

- $\sigma_c (R \bowtie S) = (\sigma_{c1} (R)) \bowtie (\sigma_{c2} (S))$

Using Heuristics in Query Optimization

- General Transformation Rules for Relational Algebra Operations (contd.):

7. Commuting π_L with \bowtie_C (or \times): Suppose that the projection list is $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$, where A_1, \dots, A_n are attributes of R and B_1, \dots, B_m are attributes of S . If the join condition c involves only attributes in L , the two operations can be commuted as follows:

$$\pi_L (R \bowtie_C S) = (\pi_{A_1, \dots, A_n} (R)) \bowtie_C (\pi_{B_1, \dots, B_m} (S))$$

- If the join condition C contains additional attributes not in L , these must be added to the projection list, and a final π_L operation is needed.

Using Heuristics in Query Optimization

- General Transformation Rules for Relational Algebra Operations (contd.):
- 8. Commutativity of set operations: The set operations \cup and \cap are commutative but “ $-$ ” is not.
- 9. Associativity of \bowtie , \times , \cup , and \cap : These four operations are individually associative; that is, if \triangle stands for any one of these four operations (throughout the expression), we have
 - $(R \triangle S) \triangle T = R \triangle (S \triangle T)$
- 10. Commuting \Join_c with set operations: The \Join_c operation commutes with \cup , \cap , and $-$. If \triangle stands for any one of these three operations, we have
 - $\Join_c (R \triangle S) = (\Join_c (R)) \triangle (\Join_c (S))$

Using Heuristics in Query Optimization

- General Transformation Rules for Relational Algebra Operations (contd.):
- The σ_L operation commutes with \cup . $\sigma_L (R \cup S) = (\sigma_L (R)) \cup (\sigma_L (S))$
- Converting a (\Join, x) sequence into \Join If the condition c of a \Join that follows a x Corresponds to a join condition, convert the (\Join, x) sequence into a \Join as follows: $(\Join_c (R \Join S)) = (R \Join_c S)$
- Other transformations

Using Heuristics in Query Optimization (14)

- Outline of a Heuristic Algebraic Optimization Algorithm:
 1. Using rule 1, break up any select operations with conjunctive conditions into a cascade of select operations.
 2. Using rules 2, 4, 6, and 10 concerning the commutativity of select with other operations, move each select operation as far down the query tree as is permitted by the attributes involved in the select condition.
 3. Using rule 9 concerning associativity of binary operations, rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive select operations are executed first in the query tree representation.
 4. Using Rule 12, combine a Cartesian product operation with a subsequent select operation in the tree into a join operation.
 5. Using rules 3, 4, 7, and 11 concerning the cascading of project and the commuting of project with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new project operations as needed.
 6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

Using Heuristics in Query Optimization (15)

- Summary of Heuristics for Algebraic Optimization:
 1. The main heuristic is to apply first the operations that reduce the size of intermediate results.
 2. Perform select operations as early as possible to reduce the number of tuples and perform project operations as early as possible to reduce the number of attributes. (This is done by moving select and project operations as far down the tree as possible.)
 3. The select and join operations that are most restrictive should be executed before other similar operations. (This is done by reordering the leaf nodes of the tree among themselves and adjusting the rest of the tree appropriately.)

Using Heuristics in Query Optimization (16)

■ Query Execution Plans

- An execution plan for a relational algebra query consists of a combination of the relational algebra query tree and information about the access methods to be used for each relation as well as the methods to be used in computing the relational operators stored in the tree.
- **Materialized evaluation:** the result of an operation is stored as a temporary relation.
- **Pipelined evaluation:** as the result of an operator is produced, it is forwarded to the next operator in sequence.

8. Using Selectivity and Cost Estimates in Query Optimization (1)

- **Cost-based query optimization:**
 - Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate.
 - (Compare to heuristic query optimization)
- **Issues**
 - Cost function
 - Number of execution strategies to be considered

Using Selectivity and Cost Estimates in Query Optimization (2)

- Cost Components for Query Execution
 1. Access cost to secondary storage
 2. Storage cost
 3. Computation cost
 4. Memory usage cost
 5. Communication cost

- Note: Different database systems may focus on different cost components.

Using Selectivity and Cost Estimates in Query Optimization (3)

- Catalog Information Used in Cost Functions
 - Information about the size of a file
 - number of records (tuples) (r),
 - record size (R),
 - number of blocks (b)
 - blocking factor (bfr)
 - Information about indexes and indexing attributes of a file
 - Number of levels (x) of each multilevel index
 - Number of first-level index blocks ($bI1$)
 - Number of distinct values (d) of an attribute
 - Selectivity (sl) of an attribute
 - Selection cardinality (s) of an attribute. ($s = sl * r$)

Using Selectivity and Cost Estimates in Query Optimization (4)

- Examples of Cost Functions for SELECT
- S1. Linear search (brute force) approach
 - $C_{S1a} = b$;
 - For an equality condition on a key, $C_{S1a} = (b/2)$ if the record is found; otherwise $C_{S1a} = b$.
- S2. Binary search:
 - $C_{S2} = \log_2 b + (s/bfr) - 1$
 - For an equality condition on a unique (key) attribute, $C_{S2} = \log_2 b$
- S3. Using a primary index (S3a) or hash key (S3b) to retrieve a single record
 - $C_{S3a} = x + 1$; $C_{S3b} = 1$ for static or linear hashing;
 - $C_{S3b} = 1$ for extendible hashing;

Using Selectivity and Cost Estimates in Query Optimization (5)

- Examples of Cost Functions for SELECT (contd.)
- S4. Using an ordering index to retrieve multiple records:
 - For the comparison condition on a key field with an ordering index, $C_{S4} = x + (b/2)$
- S5. Using a clustering index to retrieve multiple records:
 - $C_{S5} = x + \lceil (s/bfr) \rceil$
- S6. Using a secondary (B+-tree) index:
 - For an equality comparison, $C_{S6a} = x + s$;
 - For an comparison condition such as $>$, $<$, $>=$, or $<=$,
 - $C_{S6a} = x + (b_{I1}/2) + (r/2)$

Using Selectivity and Cost Estimates in Query Optimization (6)

- Examples of Cost Functions for SELECT (contd.)
- S7. Conjunctive selection:
 - Use either S1 or one of the methods S2 to S6 to solve.
 - For the latter case, use one condition to retrieve the records and then check in the memory buffer whether each retrieved record satisfies the remaining conditions in the conjunction.
- S8. Conjunctive selection using a composite index:
 - Same as S3a, S5 or S6a, depending on the type of index.
- Examples of using the cost functions.

Using Selectivity and Cost Estimates in Query Optimization (7)

■ Examples of Cost Functions for JOIN

■ Join selectivity (js)

$$■ \text{js} = | (R \bowtie_c S) | / | R \times S | = | (R \bowtie_c S) | / (|R| * |S|)$$

- If condition C does not exist, $\text{js} = 1$;
- If no tuples from the relations satisfy condition C, $\text{js} = 0$;
- Usually, $0 \leq \text{js} \leq 1$;

■ Size of the result file after join operation

$$■ | (R \bowtie_c S) | = \text{js} * |R| * |S|$$

Using Selectivity and Cost Estimates in Query Optimization (8)

- Examples of Cost Functions for JOIN (contd.)
- J1. Nested-loop join:
 - $C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|) / bfr_{RS})$
 - (Use R for outer loop)
- J2. Single-loop join (using an access structure to retrieve the matching record(s))
 - If an index exists for the join attribute B of S with index levels x_B , we can retrieve each record s in R and then use the index to retrieve all the matching records t from S that satisfy $t[B] = s[A]$.
 - The cost depends on the type of index.

Using Selectivity and Cost Estimates in Query Optimization (9)

- Examples of Cost Functions for JOIN (contd.)
- J2. Single-loop join (contd.)
 - For a secondary index,
 - $C_{J2a} = b_R + (|R| * (x_B + s_B)) + ((js * |R| * |S|)/bfr_{RS});$
 - For a clustering index,
 - $C_{J2b} = b_R + (|R| * (x_B + (s_B/bfr_B))) + ((js * |R| * |S|)/bfr_{RS});$
 - For a primary index,
 - $C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS});$
 - If a hash key exists for one of the two join attributes — B of S
 - $C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|)/bfr_{RS});$
- J3. Sort-merge join:
 - $C_{J3a} = C_S + b_R + b_S + ((js * |R| * |S|)/bfr_{RS});$
 - (CS: Cost for sorting files)

Using Selectivity and Cost Estimates in Query Optimization (10)

■ Multiple Relation Queries and Join Ordering

- A query joining n relations will have $n-1$ join operations, and hence can have a large number of different join orders when we apply the algebraic transformation rules.
- Current query optimizers typically limit the structure of a (join) query tree to that of left-deep (or right-deep) trees.

■ Left-deep tree:

- A binary tree where the right child of each non-leaf node is always a base relation.
 - Amenable to pipelining
 - Could utilize any access paths on the base relation (the right child) when executing the join.

9. Overview of Query Optimization in Oracle

- Oracle DBMS V8
 - **Rule-based query optimization:** the optimizer chooses execution plans based on heuristically ranked operations.
 - (Currently it is being phased out)
 - **Cost-based query optimization:** the optimizer examines alternative access paths and operator algorithms and chooses the execution plan with lowest estimate cost.
 - The query cost is calculated based on the estimated usage of resources such as I/O, CPU and memory needed.
 - Application developers could specify hints to the ORACLE query optimizer.
 - The idea is that an application developer might know more information about the data.

10. Semantic Query Optimization

- **Semantic Query Optimization:**

- Uses constraints specified on the database schema in order to modify one query into another query that is more efficient to execute.

- Consider the following SQL query,

```
SELECT      E.LNAME, M.LNAME
FROM        EMPLOYEE E M
WHERE       E.SUPERSSN=M.SSN AND E.SALARY>M.SALARY
```

- **Explanation:**

- Suppose that we had a constraint on the database schema that stated that no employee can earn more than his or her direct supervisor. If the semantic query optimizer checks for the existence of this constraint, it need not execute the query at all because it knows that the result of the query will be empty. Techniques known as theorem proving can be used for this purpose.

Summary

0. Introduction to Query Processing
1. Translating SQL Queries into Relational Algebra
2. Algorithms for External Sorting
3. Algorithms for SELECT and JOIN Operations
4. Algorithms for PROJECT and SET Operations
5. Implementing Aggregate Operations and Outer Joins
6. Combining Operations using Pipelining
7. Using Heuristics in Query Optimization
8. Using Selectivity and Cost Estimates in Query Optimization
9. Overview of Query Optimization in Oracle
10. Semantic Query Optimization