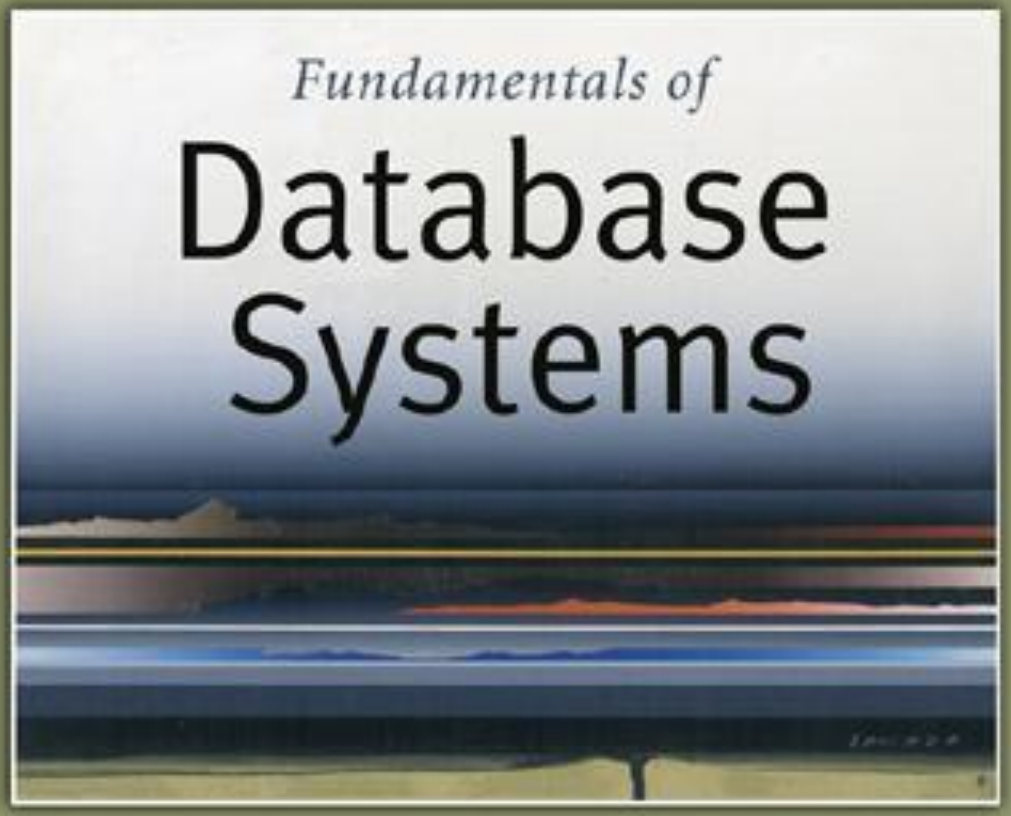
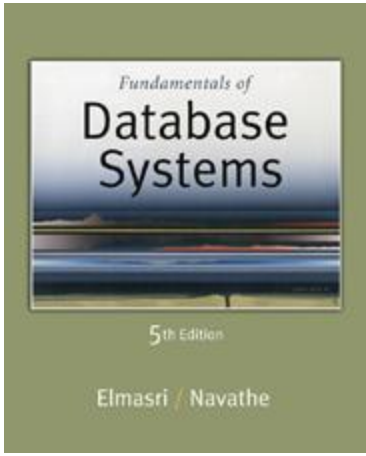


5th Edition

Elmasri / Navathe

Chapter 15

Algorithms for Query Processing and Optimization



Chapter Outline (1)

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

Introduction to Query Processing (3)

- Scanner – identifies language tokens- sql keywords, relation,attributes names

- Parser –checks syntax of the query(grammar rules)

- Validation- all attributes and relation names are valid and semantically meaningful names as given in schema

- Two internal representations of a query:
  Query Tree
  Query Graph

Introduction to Query Processing (2)

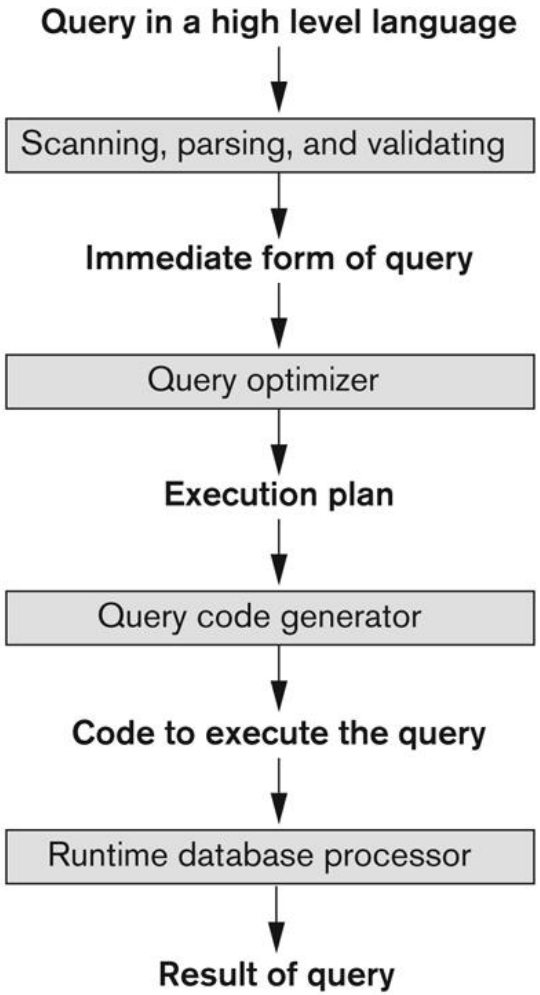


Figure 15.1
Typical steps when processing a high-level query.

Code can be:

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

Introduction to Query Processing (1)

- **Query optimization**:
  - The process of choosing a suitable execution strategy /plan for processing a query.
  - Query code generator
    - Generators code to execute the plan
  - Runtime database processor

    runs the query code in compiled/interpreted mode to produce the query result

Queries are translated into relational algebra queries and then optimized

Optimization is done in two ways
1.Heuristic rules and
2.Systematic estimation(Lowest cost estimate)

1. Translating SQL Queries into Relational Algebra (1)

- **SQL query is translated into query blocks**
- **Query block**:
  - The basic unit that can be translated into the algebraic operators and optimized.

Translating SQL Queries into Relational Algebra (1)

- 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 (2)

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

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

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

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

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

Algorithms for External Sorting

- Result should me sorted if the query uses

 - order by, join, union, intersection, unique/distinct clause

- Ordering can be avoided if indexing exists

2. Algorithms for External Sorting (1)

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**:

  2 phases

  - Sort phase- sorting small subfiles (**runs**) of the main file
  - Merge Phase – Merges the sorted runs, creating larger sorted subfiles that are merged in turn.

2. Algorithms for External Sorting (1)

- Sorting phase:
  - n_R: number of initial runs;
  - b: number of file blocks of data;
  - n_B: available buffer space in chace;
  - $n_R = \lceil (b/n_B) \rceil$
- Merging phase:
  - $d_M = \text{Min}\ (n_B\text{-}1, n_R)$;
  - $n_P = \lceil (\log_{dM}(n_R)) \rceil$

- d_M: degree of merging, number of sorted subfiles that can be merged in each merge step;
- n_P: number of merge passes
- Buffer size= block size.

For example,

- size *of* the file *b* = 1024 blocks,
- *if nB* = 5 blocks

- Then find runs, degree of merging

- nR = [*(b/nB)* l , or 205 initial runs each *of* size 5 blocks (except the last run which will have 4 blocks).
- Hence, after the sort phase, 205 sorted runs are stored as temporary subfiles on disk.

- -, the sorted runs are merged during one or more passes.
- -degree of merging *(dM)* is the number *of* runs that can be merged together in each pass.
- In each pass,one buffer block is needed to hold one block *from* each *of* the runs being merged, and one block is needed *for* containing one block *of* the merge result.

- , dM = min(5-1, 205) =4

- 4(four-way merging), so the 205 initial sorted runs would be merged into 52 at the end *of* the first pass,
- which are then merged into 13, then 4, then 1 run, which means that *four passes* are needed.
- The minimum dM *of* 2 gives the worst-case performance *of* the algorithm, which is

Algorithms for External Sorting (2)

```
set     i  ← 1;
        j  ← b;                 {size of the file in blocks}
        k  ← nB;                {size of buffer in blocks}
        m  ← ⌈(j/k)⌉;

{Sort Phase}
while (i ≤ m)
do {
        read next k blocks of the file into the buffer or if there are less than k blocks
            remaining, then read in the remaining blocks;
        sort the records in the buffer and write as a temporary subfile;
        i  ← i + 1;
}

{Merge Phase: merge subfiles until only 1 remains}
set     i  ← 1;
        p  ← ⌈log(k−1) m⌉;      {p is the number of passes for the merging phase}
        j  ← m;
while (i ≤ p)
do {
        n  ← 1;
        q  ← ⌈(j/(k−1))⌉;       {number of subfiles to write in this pass}
        while (n ≤ q)
        do {
                read next k−1 subfiles or remaining subfiles (from previous pass)
                    one block at a time;
                merge and write as new subfile one block at a time;
                n ← n + 1;
        }
        j ← q;
        i ← i + 1;
}
```

Figure 15.2
Outline of the sort-merge algorithm for external sorting.

3. Algorithms for SELECT and JOIN Operations (1)

- Implementing the SELECT Operation

- Examples:
  - (OP1): $\sigma_{SSN='123456789'}(EMPLOYEE)$
  - (OP2): $\sigma_{DNUMBER>5}(DEPARTMENT)$
  - (OP3): $\sigma_{DNO=5}(EMPLOYEE)$
  - (OP4): $\sigma_{DNO=5\ AND\ SALARY>30000\ AND\ SEX=F}(EMPLOYEE)$
  - (OP5): $\sigma_{ESSN=123456789\ AND\ PNO=10}(WORKS\_ON)$

Algorithms for SELECT and JOIN Operations (2)

- 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 on a key attribute on which the file is ordered, binary search (which is more efficient than linear search) can be used. (See OP1).
  - S3 **Using a primary index or hash key to retrieve a single record**:
    - If the selection condition involves an equality comparison 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 (3)

- Implementing the SELECT Operation (contd.):
- Search Methods for Simple Selection:
  - S4 **Using a primary index to retrieve multiple records**:
    - If the comparison condition is >, ≥, <, or ≤ on a key field with a primary index, 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 with a clustering index, 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 >,>=, <, or <=. (FOR RANGE QUERIES)

Algorithms for SELECT and JOIN Operations (4)

- Implementing the SELECT Operation (contd.):
- Search Methods for Simple Selection:
  - S7 **Conjunctive selection**:
    - If an attribute involved in any single simple condition in the conjunctive condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.
  - S8 **Conjunctive selection using a composite index**
    - If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined field, we can use the index directly.

Algorithms for SELECT and JOIN Operations (5)

- Implementing the SELECT Operation (contd.):
- Search Methods for Complex Selection:
  - S9 **Conjunctive selection by intersection of record pointers**:
    - This method is possible if secondary indexes are available on all (or some of) the fields involved in equality comparison conditions in the conjunctive condition and if the indexes include record pointers (rather than block pointers).
    - Each index can be used to retrieve the record pointers that satisfy the individual condition.
    - The intersection of these sets of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly.
    - If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.

Algorithms for SELECT and JOIN Operations (7)

- Implementing the SELECT Operation (contd.):
  - Whenever a **single condition** specifies the selection, we can only check whether an access path exists on the attribute involved in that condition.
    - If an access path exists, the method corresponding to that access path is used; otherwise, the "brute force" linear search approach of method S1 is used. (See OP1, OP2 and OP3)
  - For **conjunctive selection conditions**, whenever *more than one* of the attributes involved in the conditions have an access path, query optimization should be done to choose the access path that *retrieves the fewest records* in the most efficient way.
  - **Disjunctive selection conditions**

Algorithms for SELECT and JOIN Operations (8)

- 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): EMPLOYEE $\bowtie_{DNO=DNUMBER}$ DEPARTMENT
  - (OP7): DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE

Algorithms for SELECT and JOIN Operations (9)

- 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 (10)

- 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 (11)

- 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.

Implementing Join operation-using sort merge

operation $T \leftarrow R \bowtie_{A=B} S$.

```
(a) sort the tuples in R on attribute A;      (* assume R has n tuples (records) *)
    sort the tuples in S on attribute B;      (* assume S has m tuples (records) *)
    set i ← 1, j ← 1;
    while (i ≤ n) and (j ≤ m)
    do {  if R(i)[A] > S(j)[B]
             then  set j ← j + 1
          elseif R(i)[A] < S(j)[B]
             then  set i ← i + 1
          else  {  (* R(i)[A] = S(j)[B], so we output a matched tuple *)
                   output the combined tuple <R(i), S(j)> to T;

                   (* output other tuples that match R(i), if any *)
                   set l ← j + 1;
                   while (l ≤ m) and (R(i)[A] = S(l)[B])
                   do {  output the combined tuple <R(i), S(l)> to T;
                         set l ← l + 1
                   }

                   (* output other tuples that match S(j), if any *)
                   set k ← i + 1;
                   while (k ≤ n) and (R(k)[A] = S(j)[B])
                   do {  output the combined tuple <R(k), S(j)> to T;
                         set k ← k + 1
                   }
                   set i ← k, j ← l
          }
    }
```

Implementing project operation

```
(b) create a tuple t[<attribute list>] in T' for each tuple t in R;
      (* T' contains the projection results before duplicate elimination *)
    if <attribute list> includes a key of R
        then T ← T'
    else  {  sort the tuples in T';
             set i ← 1, j ← 2;
             while i ≤ n
             do {  output the tuple T'[i] to T;
                   while T'[i] = T'[j] and j ≤ n do j ← j + 1;   (* eliminate duplicates *)
                   i ← j; j ← i + 1
             }
    }
    (* T contains the projection result after duplicate elimination *)
```

Implementing T<-RUS

```
(c) sort the tuples in R and S using the same unique sort attributes;
    set i ← 1, j ← 1;
    while (i ≤ n) and (j ≤ m)
    do {  if R(i) > S(j)
             then  {  output S(j) to T;
                      set j ← j + 1
                   }
          elseif R(i) < S(j)
             then  {  output R(i) to T;
                      set i ← i + 1
                   }
          else set  j ← j + 1    (* R(i)=S(j), so we skip one of the duplicate tuples *)
    }
    if (i ≤ n) then add tuples R(i) to R(n) to T;
    if (j ≤ m) then add tuples S(j) to S(m) to T;
```

T<- R ∩ S

(d) sort the tuples in R and S using the same unique sort attributes;
set $i \leftarrow 1, j \leftarrow 1$;
while $(i \leq n)$ and $(j \leq m)$
do { if $R(i) > S(j)$
then set $j \leftarrow j + 1$
elseif $R(i) < S(j)$
then set $i \leftarrow i + 1$
else { output $R(j)$ to T; (* $R(i)=S(j)$, so we output the tuple *)
set $i \leftarrow i + 1, j \leftarrow j + 1$
}
}

T<- R-S

(e) sort the tuples in R and S using the same unique sort attributes;
set $i \leftarrow 1, j \leftarrow 1$;
while $(i \leq n)$ and $(j \leq m)$
do { if $R(i) > S(j)$
then set $j \leftarrow j + 1$
elseif $R(i) < S(j)$
then { output $R(i)$ to T; (* $R(i)$ has no matching $S(j)$, so output $R(i)$ *)
set $i \leftarrow i + 1$
}
else set $i \leftarrow i + 1, j \leftarrow j + 1$
}
if $(i \leq n)$ then add tuples $R(i)$ to $R(n)$ to T;

Algorithms for SELECT and JOIN Operations (14)

- Implementing the JOIN Operation (contd.):
- Factors affecting JOIN performance
  - Available buffer space
  - Join selection factor
  - Choice of inner VS outer relation

Algorithms for SELECT and JOIN Operations (15)

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:
      - R1 , R2 , R3 , ... Rm and S1 , S2 , S3 , ... Sm
    - 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 Ri and Si.

Algorithms for SELECT and JOIN Operations (16)

Implementing the JOIN Operation (contd.):

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

Algorithms for SELECT and JOIN Operations (17)

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:
  - $3^* (b_R + b_S) + b_{RES}$

Algorithms for SELECT and JOIN Operations (18)

- 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.

4. Algorithms for PROJECT and SET Operations (1)

- Algorithm for PROJECT operations (Figure 15.3b)

  $\pi_{<\text{attribute list}>}(R)$

  1. If <attribute list> has a key of relation R, extract all tuples from R with only the values for the attributes in <attribute list>.
  2. If <attribute list> does NOT include a key of relation R, duplicated tuples must be removed from the results.

- Methods to remove duplicate tuples
  1. Sorting
  2. Hashing

Algorithms for PROJECT and SET Operations (2)

- **Algorithm for SET operations**
- **Set operations**:
  - UNION, INTERSECTION, SET DIFFERENCE and CARTESIAN PRODUCT
- **CARTESIAN PRODUCT** of relations R and S include all possible combinations of records from R and S. The attribute of the result include all attributes of R and S.
- **Cost analysis** of CARTESIAN PRODUCT
  - If R has n records and j attributes and S has m records and k attributes, the result relation will have n*m records and j+k attributes.
- CARTESIAN PRODUCT operation is **very expensive** and should be avoided if possible.

Algorithms for PROJECT and SET Operations (3)

- **Algorithm for SET operations (contd.)**
- **UNION** (See Figure 15.3c)
  - Sort the two relations on the same attributes.
  - Scan and merge both sorted files concurrently, whenever the same tuple exists in both relations, only one is kept in the merged results.
- **INTERSECTION** (See Figure 15.3d)
  - Sort the two relations on the same attributes.
  - Scan and merge both sorted files concurrently, keep in the merged results only those tuples that appear in both relations.
- **SET DIFFERENCE R-S** (See Figure 15.3e)
  - Keep in the merged results only those tuples that appear in relation R but not in relation S.

5. Implementing Aggregate Operations and Outer Joins (1)

- Implementing Aggregate Operations:
- **Aggregate operators**:
  - **MIN, MAX, SUM, COUNT** and **AVG**
- Options to implement aggregate operators:
  - **Table Scan**
  - **Index**
- Example
  - SELECT MAX (SALARY)
  - FROM EMPLOYEE;
- If an (ascending) index on SALARY exists for the employee relation, then the optimizer could decide on traversing the index for the largest value, which would entail following the right most pointer in each index node from the root to a leaf.

Implementing Aggregate Operations and Outer Joins (2)

Implementing Aggregate Operations (contd.):

- **SUM, COUNT and AVG**
- For a **dense index** (each record has one index entry):
  - Apply the associated computation to the values in the index.
- For a **non-dense index**:
  - Actual number of records associated with each index entry must be accounted for
- With **GROUP BY**: the aggregate operator must be applied separately to each group of tuples.
  - Use sorting or hashing on the group attributes to partition the file into the appropriate groups;
  - Computes the aggregate function for the tuples in each group.

Implementing Aggregate Operations and Outer Joins (3)

Implementing Outer Join:

- **Outer Join Operators**:
  - **LEFT OUTER JOIN**
  - **RIGHT OUTER JOIN**
  - **FULL OUTER JOIN**.
- The full outer join produces a result which is equivalent to the union of the results of the left and right outer joins.
- Example:

```
SELECT    FNAME, DNAME
FROM      (EMPLOYEE LEFT OUTER JOIN DEPARTMENT
          ON DNO = DNUMBER);
```

- Note: The result of this query is a table of employee names and their associated departments. It is similar to a regular join result, with the exception that if an employee does not have an associated department, the employee's name will still appear in the resulting table, although the department name would be indicated as null.

Implementing Aggregate Operations and Outer Joins (4)

Implementing Outer Join (contd.):

- **Modifying Join Algorithms**:
  - Nested Loop or Sort-Merge joins can be modified to implement outer join. E.g.,
    - For left outer join, use the left relation as outer relation and construct result from every tuple in the left relation.
    - If there is a match, the concatenated tuple is saved in the result.
    - However, if an outer tuple does not match, then the tuple is still included in the result but is padded with a null value(s).

Implementing Aggregate Operations and Outer Joins (5)

- Implementing Outer Join (contd.):
- Executing a combination of relational algebra operators.
- Implement the previous left outer join example
  - {Compute the JOIN of the EMPLOYEE and DEPARTMENT tables}
    - TEMP1 $\leftarrow \pi_{FNAME,DNAME}$ (EMPLOYEE $\bowtie_{DNO=DNUMBER}$ DEPARTMENT)
  - {Find the EMPLOYEEs that do not appear in the JOIN}
    - TEMP2 $\leftarrow \pi_{FNAME}$ (EMPLOYEE) - π_{FNAME} (Temp1)
  - {Pad each tuple in TEMP2 with a null DNAME field}
    - TEMP2 $\leftarrow$ TEMP2 x 'null'
  - {UNION the temporary tables to produce the LEFT OUTER JOIN}
    - RESULT $\leftarrow$ TEMP1 $\cup$ TEMP2
- The cost of the outer join, as computed above, would include the cost of the associated steps (i.e., join, projections and union).

6. Combining Operations using Pipelining (1)

- Motivation
  - A query is mapped into a sequence of operations.
  - Each execution of an operation produces a temporary result.
  - Generating and saving temporary files on disk is time consuming and expensive.
- Alternative:
  - Avoid constructing temporary results as much as possible.
  - Pipeline the data through multiple operations - pass the result of a previous operator to the next without waiting to complete the previous operation.

Combining Operations using Pipelining (2)

- Example:
  - For a 2-way join, combine the 2 selections on the input and one projection on the output with the Join.
- Dynamic generation of code to allow for multiple operations to be pipelined.
- Results of a select operation are fed in a "Pipeline" to the join algorithm.
- Also known as stream-based processing.

7. Using Heuristics in Query Optimization(1)

- 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 (2)

- **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.

- Emp-

(FNAME, LNAME, SSN, BDATE, ADDRESS,CITY, SUPERVISOR-SSN)

- DEPT

-(DNAME, DNUMBER, MGR-SSN, START-DATE)

- WORKS- ON(ESSN,PNO,HOURS)
- PROJECT

(PNAME, PNUMBER,PLOCATION,DNUM)

Using Heuristics in Query Optimization (3)

- 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.

- 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’;
```

- Relation algebra:

$$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}(((\sigma_{\text{PLOCATION='STAFFORD'}}(\text{PROJECT})) \bowtie_{\text{DNUM=DNUMBER}} (\text{DEPARTMENT})) \bowtie_{\text{MGRSSN=SSN}} (\text{EMPLOYEE}))$$

Using Heuristics in Query Optimization (4)

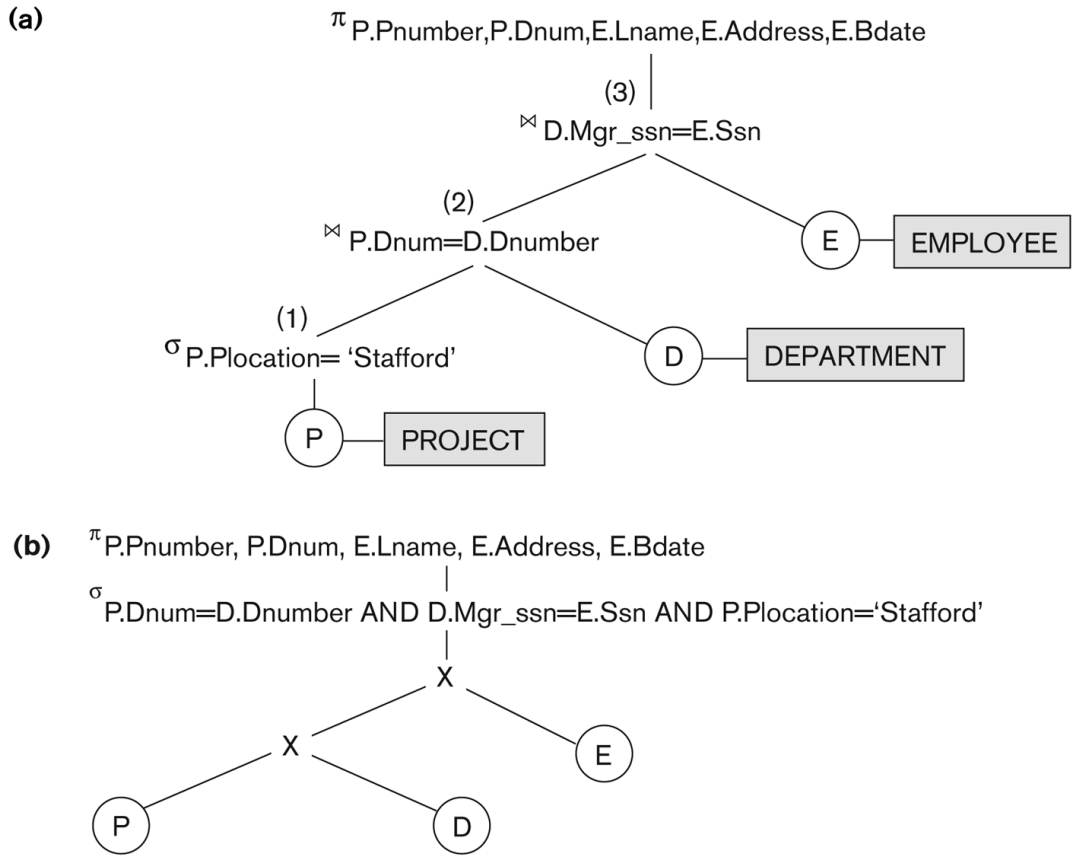


Figure 15.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

Using Heuristics in Query Optimization (5)

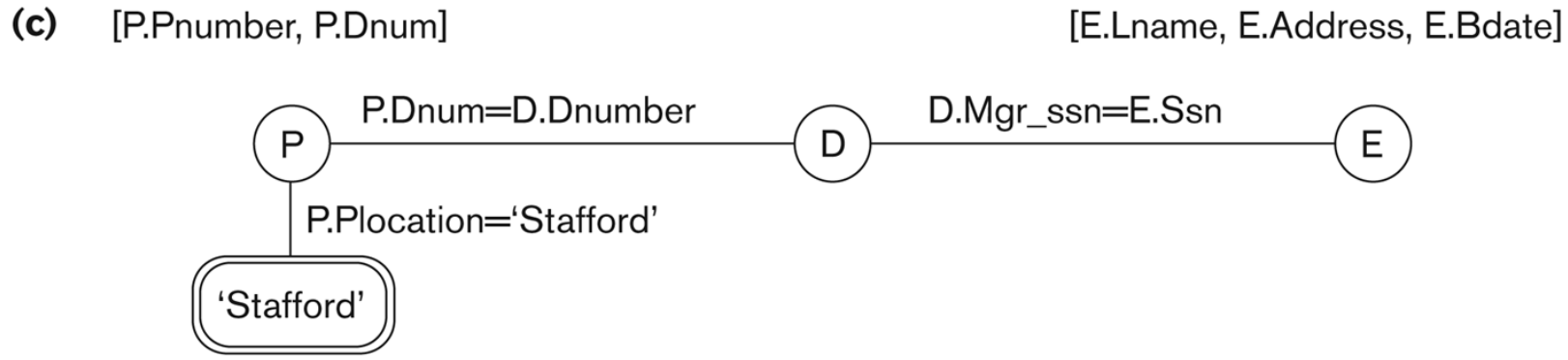


Figure 15.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

Using Heuristics in Query Optimization (6)

- 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: Find the last name of employees working on project ‘AQUARIUS ‘ and born after 1957-12-31

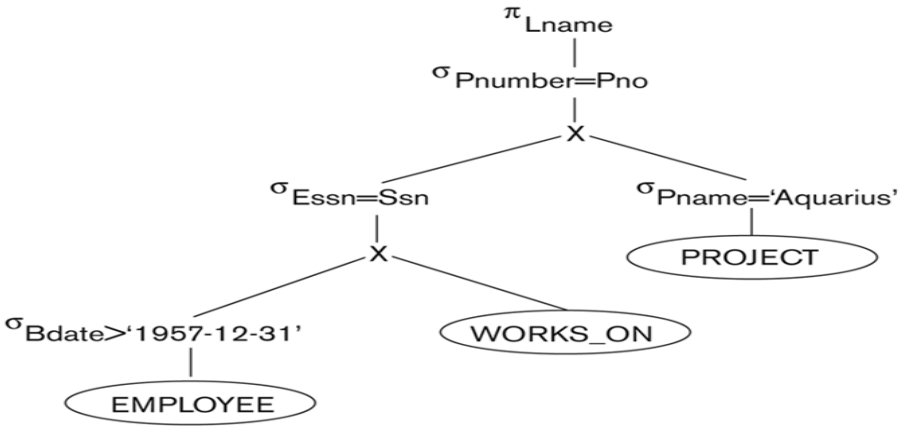
Using Heuristics in Query Optimization (6)

- Heuristic Optimization of Query Trees:

- Example:

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

(b)



(c)

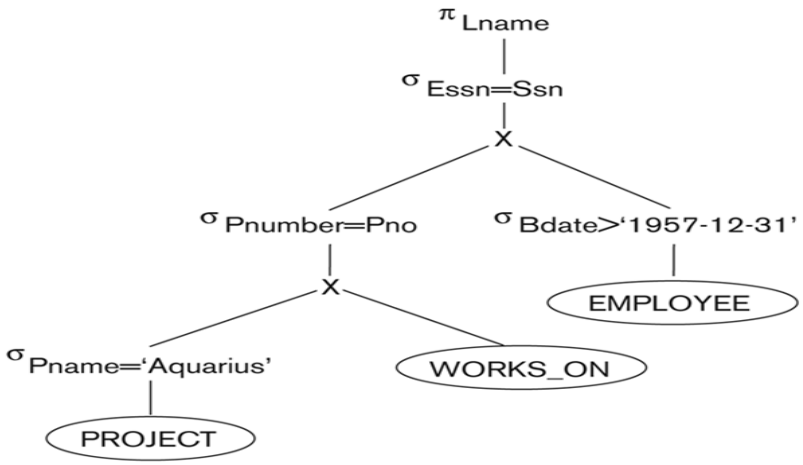


Figure 15.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOI
(e) Moving PROJECT operations down the query tree.

Using Heuristics in Query Optimization (7)

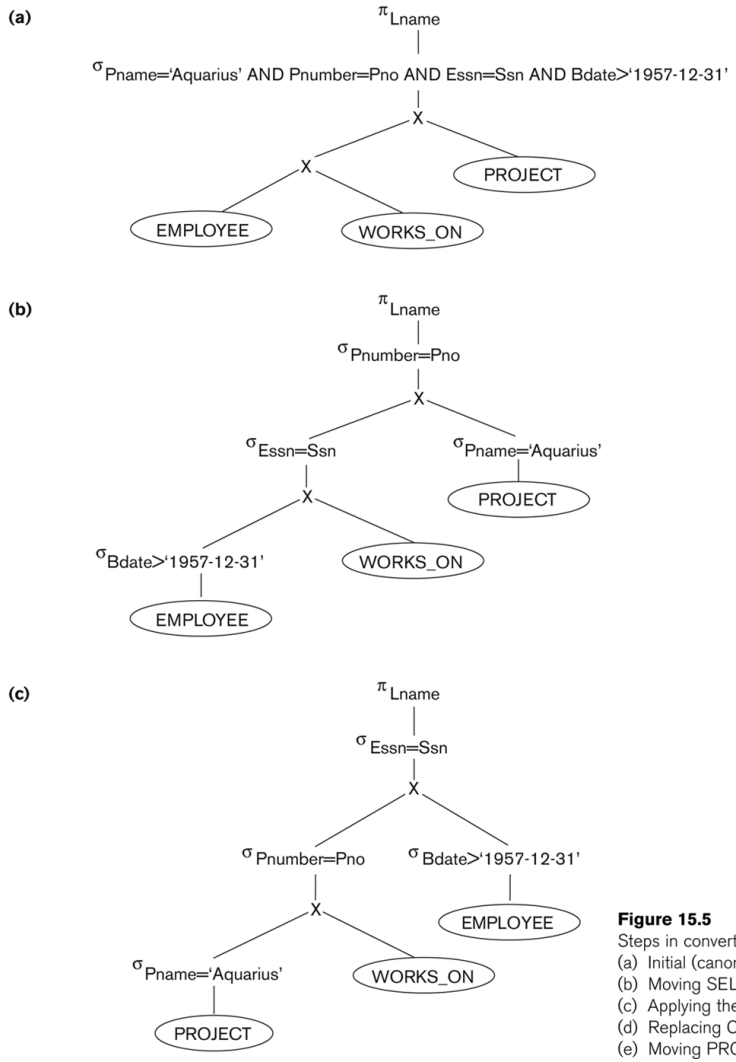


Figure 15.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.

Using Heuristics in Query Optimization (7)

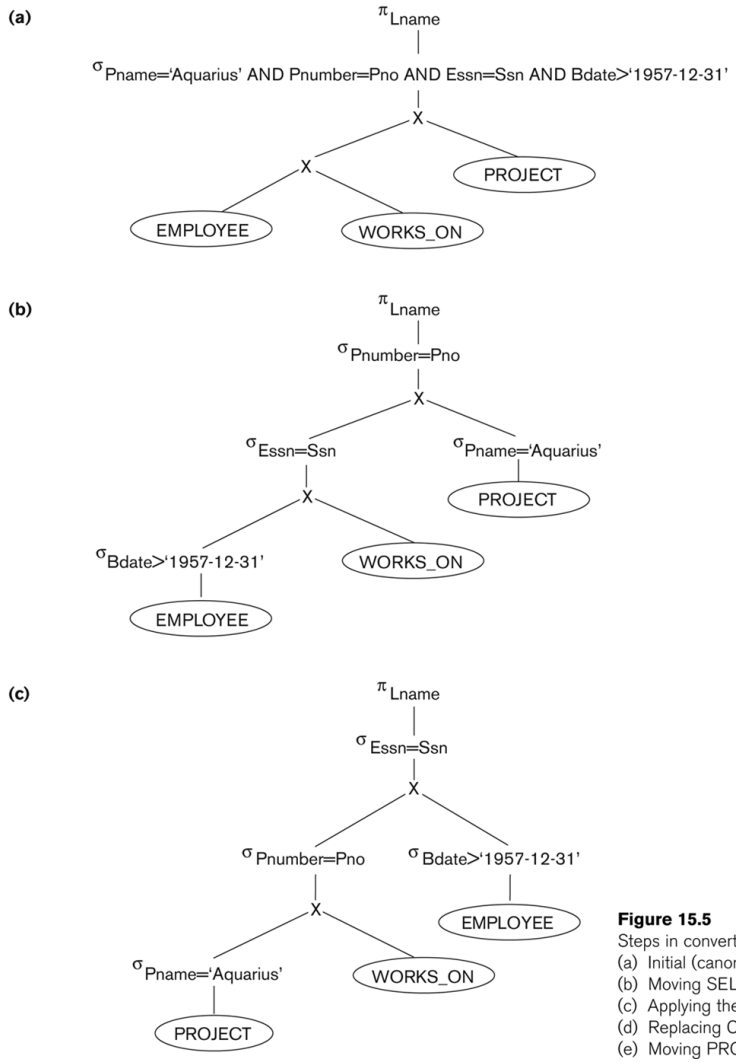


Figure 15.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.

Using Heuristics in Query Optimization (7)

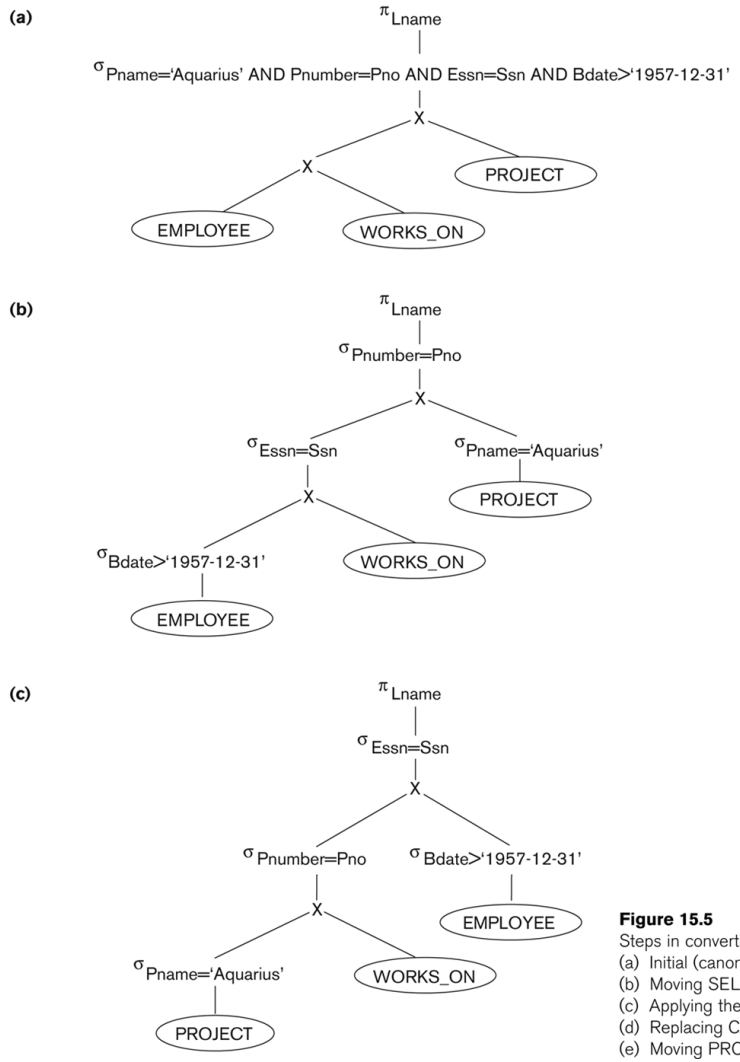


Figure 15.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.

Using Heuristics in Query Optimization (8)

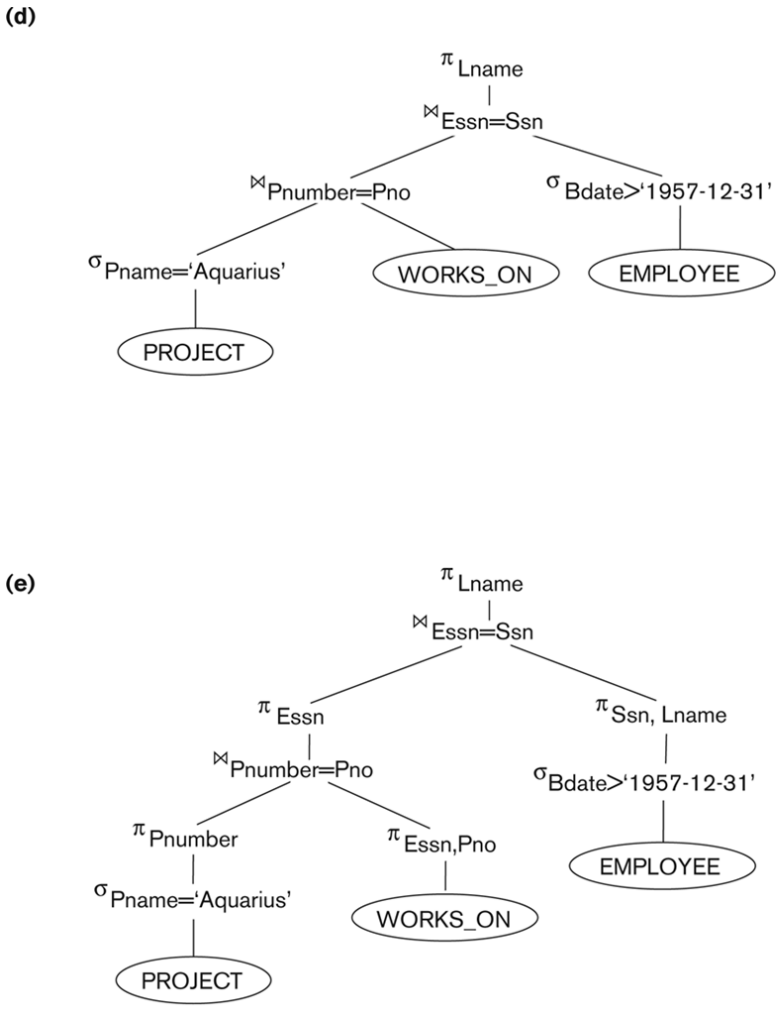


Figure 15.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.

Using Heuristics in Query Optimization (9)

- General Transformation Rules for Relational Algebra Operations:

1. Cascade of σ: A conjunctive selection condition can be broken up into a cascade (sequence) of individual σ operations:
   - $\sigma_{c1 \text{ AND } c2 \text{ AND } ... \text{ AND } cn}(R) = \sigma_{c1}(\sigma_{c2}(...(\sigma_{cn}(R))...))$
2. Commutativity of σ: The σ operation is commutative:
   - $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$
3. Cascade of π: In a cascade (sequence) of π operations, all but the last one can be ignored:
   - $\pi_{List1}(\pi_{List2}(...(\pi_{Listn}(R))...)) = \pi_{List1}(R)$
4. Commuting σ with π: If the selection condition c involves only the attributes A1, ..., An in the projection list, the two operations can be commuted:
   - $\pi_{A1, A2, ..., An}(\sigma_c(R)) = \sigma_c(\pi_{A1, A2, ..., An}(R))$

Using Heuristics in Query Optimization (10)

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

5. Commutativity of $\bowtie$ (and x): The $\bowtie$ operation is commutative as is the x operation:
   - $R \bowtie_C S = S \bowtie_C R$; $R \times S = S \times R$
6. Commuting σ with $\bowtie$ (or x): 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 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 (11)

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

7. Commuting π with $\bowtie$ (or x): Suppose that the projection list is L = {A1, ..., An, B1, ..., Bm}, where A1, ..., An are attributes of R and B1, ..., Bm are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:
   - $\pi_L (R \bowtie_C S) = (\pi_{A1, ..., An} (R)) \bowtie_C (\pi_{B1, ..., Bm} (S))$
- If the join condition C contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed.

Using Heuristics in Query Optimization (12)

- 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 $\bowtie$, x, $\cup$, and $\cap$: These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have
   - $(R \theta S) \theta T = R \theta (S \theta T)$
10. Commuting σ with set operations: The σ operation commutes with $\cup$, $\cap$, and –. If θ stands for any one of these three operations, we have
   - $\sigma_c (R \theta S) = (\sigma_c (R)) \theta (\sigma_c (S))$

Using Heuristics in Query Optimization (13)

- General Transformation Rules for Relational Algebra Operations (contd.):
- The π operation commutes with υ.

$$\pi_L (R \upsilon S) = (\pi_L (R)) \upsilon (\pi_L (S))$$

- Converting a (σ, x) sequence into $\bowtie$: If the condition c of a σ that follows a x Corresponds to a join condition, convert the (σ, x) sequence into a $\bowtie$ as follows:

$$(\sigma_C (R \times S)) = (R \bowtie_C S)$$

- 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 is the fraction of records satisfying the equality condition on the 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:
  - $C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$
  - 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;

number

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)
  - js = | (R $\bowtie_C$ S) | / | R x S | = | (R $\bowtie_C$ S) | / (|R| * |S |)
    - If condition C does not exist, js = 1;
    - If no tuples from the relations satisfy condition C, js = 0;
    - Usually, 0 <= js <= 1;
- Size of the result file after join operation
  - | (R $\bowtie_C$ S) | = 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