



Chapter 18: Strategies for Query Processing
Chapter 19: Query Optimization

CS-6360 Database Design

Chris Irwin Davis, Ph.D.

Email: chrisirwindavis@utdallas.edu

Phone: (972) 883-3574

Office: ECSS 4.705

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

19.1 – Translating SQL Queries into Relational Algebra

- In practice, SQL is the query language that is used in most commercial RDBMSs.
- An SQL query is first translated into an equivalent extended relational algebra expression—represented as a **query tree data structure**—that is then optimized.


- **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.

```
SELECT  Lname, Fname
FROM    EMPLOYEE
WHERE   Salary > (      SELECT  MAX (Salary)
                        FROM    EMPLOYEE
                        WHERE   Dno = 5);
```

Nested query without correlation with the outer query

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
```

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

Must be a single value, i.e. exactly one column and one row

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
```

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

$\pi_{\text{Lname, Fname}} (\sigma_{\text{Salary} > \mathbf{C}} (\text{EMPLOYEE}))$

$\mathcal{F}_{\text{MAX Salary}} (\sigma_{\text{Dno}=5} (\text{EMPLOYEE}))$

19.2 – Algorithms for External Sorting

- Data access occurs in main memory
- It is necessary to copy from data stored in files into main memory
- Available memory is (usually) less than the size of data files
- How to access data in files?
 - n_B – buffer size in disk blocks (int)
 - b – file size in disk blocks (int)
 - n_R – number of “runs” to access the entire file (int)

- How to sort records in a file?
- Sort + Merge
 - Sort the records in each block
 - Recursively merge the blocks

- **Sorting Phase**
 - The **size of each run** and the **available buffer space** (n_B).
 - For example, if the number of available main memory buffers $n_B = 5$ **disk blocks** and the size of the file $b = 1024$ **disk blocks**, then $n_R = \lceil (b/n_B) \rceil$ or 205 initial runs each of size 5 blocks (except the last run which will have only 4 blocks).
 - Hence, after the sorting phase, 205 sorted runs (or 205 sorted subfiles of the original file) are stored as temporary subfiles on disk.

■ Merging Phase

- The sorted runs are merged during one or more merge passes.
- Each merge pass can have one or more merge steps.
- The degree of merging (d_M) is the number of sorted subfiles that can be merged in each merge step.

■ Performance

- The performance of the **sort-merge algorithm** can be measured in the number of disk block reads and writes (between the disk and main memory) before the sorting of the whole file is completed. The following formula approximates this cost:

$$(2 * b) + (2 * b * (\log_{d_M} n_R))$$

19.3 – Algorithms for SELECT and JOIN

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

- 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.

- Implementing the SELECT Operation (cont'd.):
- 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, 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 $>$, \geq , $<$, or \leq . (FOR RANGE QUERIES)

- Implementing the SELECT Operation (cont'd.):
- 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.

- Implementing the SELECT Operation (cont'd.):
- 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.

- Implementing the SELECT Operation (cont'd.):
 - 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...**

■ Disjunctive selection conditions

- Compared to a conjunctive selection condition, a disjunctive condition (where simple conditions are connected by the OR logical connective rather than by AND) is much harder to process and optimize. For example, consider OP4':

OP4': $\sigma_{Dno=5 \text{ OR } Salary>30000 \text{ OR } Sex='F'}(EMPLOYEE)$

- With such a condition, little optimization can be done, because the records satisfying the disjunctive condition are the *union* of the records satisfying the individual conditions.
- Hence, if any one of the conditions does not have an access path, we are compelled to use the brute force, linear search approach.
- Only if an access path exists on *every* simple condition in the disjunction can we optimize the selection by retrieving the records satisfying each condition—or their record ids—and then applying the *union* operation to eliminate duplicates.

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. $R \bowtie_{A=B} S \bowtie_{C=D} T$
 - Implemented as combinations of two-way joins
 - $(R \bowtie_{A=B} S) \bowtie_{C=D} T$
 - $R \bowtie_{A=B} (S \bowtie_{C=D} T)$

■ Examples

- (OP6): EMPLOYEE $\bowtie_{\text{Dno=Dnumber}}$ DEPARTMENT
- (OP7): DEPARTMENT $\bowtie_{\text{Mgrssn=Ssn}}$ EMPLOYEE

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 $t[A] = s[B]$.
- Computational cost?

```
for rRow in R:
    for sRow in S:
        if(R.A == S.A)
            rRow.join(sRow)
```


Implementing the JOIN Operation (contd.):

- Methods for implementing joins:
 - **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 $s[B] = t[A]$.
 - Computational cost?

```
for rRow in R:  
    rRow.lookup(sRow)
```

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.
- Computational cost?

```
while(R.hasMore() && S.hasMore())  
    if    (R[0]==(S[0])) then R[0].join(S[0])  
    elseif(R[0]>(S[0]))  then S.next()  
    elseif(R[0]<(S[0]))  then R.next()
```

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, the **partitioning phase**
- 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 , the **probing phase**.
- Computational cost?

```
for r_row in R:  
    r_row.hashLookup()
```

Implementing the JOIN Operation (contd.):

■ J4 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 .

Implementing the JOIN Operation (contd.):

■ J4 Partitioned Hash Join Procedure (cont'd):

- 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 R_i .
3. Write matching record from R_i after joining to the record from S_i into the result file.

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

Implementing the JOIN Operation (contd.):

- Factors affecting JOIN performance
 - Join selection factor
 - Available buffer space

19.4 – Algorithms for PROJECT and Set Operations

■ Algorithm for **PROJECT** operations (Figure 15.3b)

$\pi_{\langle \text{attribute list} \rangle}(R)$

- If $\langle \text{attribute list} \rangle$ has a key of relation R , extract all tuples from R with only the values for the attributes in $\langle \text{attribute list} \rangle$.
 - If $\langle \text{attribute list} \rangle$ does NOT include a key of relation R , duplicated tuples must be removed from the results.
-
- Methods to remove duplicate tuples
 - Sorting
 - Hashing

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.

Algorithm for SET operations (contd.)

- UNION (See Figure 19.3c)
 - 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 19.3d)
 - 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 19.3e)
 - Keep in the merged results only those tuples that appear in relation R but not in relation S .

19.6 – Combining Operations Using Pipelining

■ 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.

- 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**.

19.7 – 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 first apply the operations that reduce the size of intermediate results.
 - e.g., Apply **SELECT** and **PROJECT** operations before applying the **JOIN** or other binary operations.

- 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:

$$\pi_{Pnumber, Dnum, Lname, Address, Bdate} (((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber} (DEPARTMENT)) \bowtie_{Mgr_ssn=Ssn} (EMPLOYEE))$$

- SQL query:

Q2: SELECT P.Pnumber, 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.Mgr_ssn=E.Ssn **AND**
P.Plocation= 'Stafford';

Using Heuristics in Query Optimization

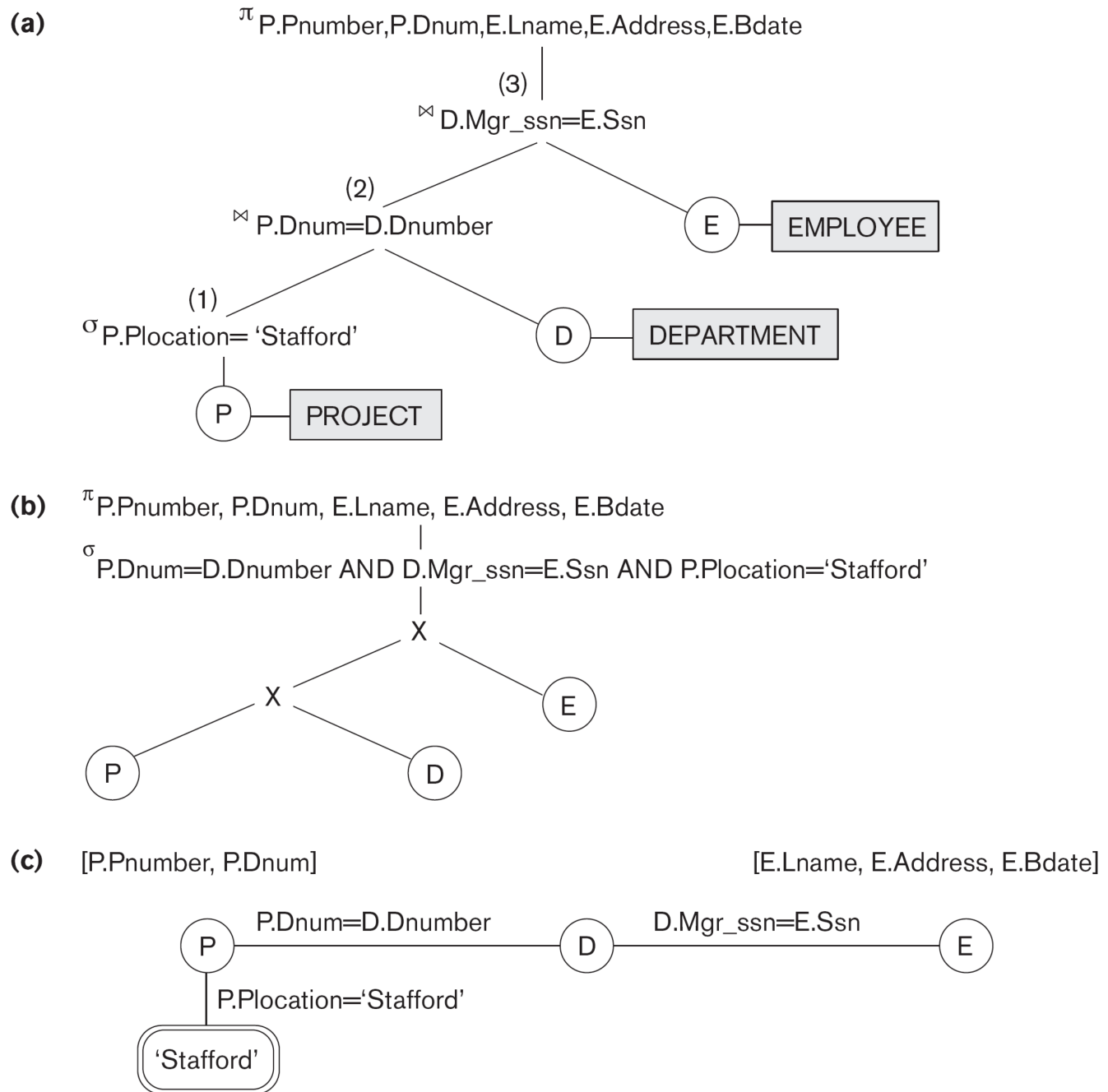
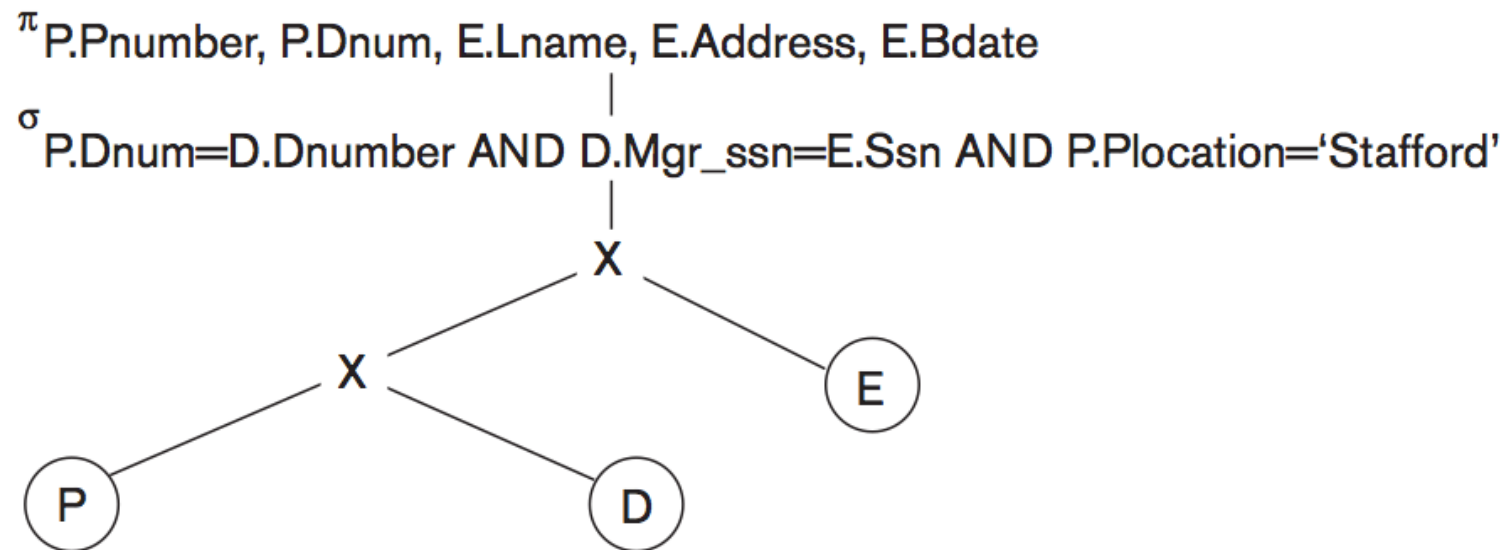


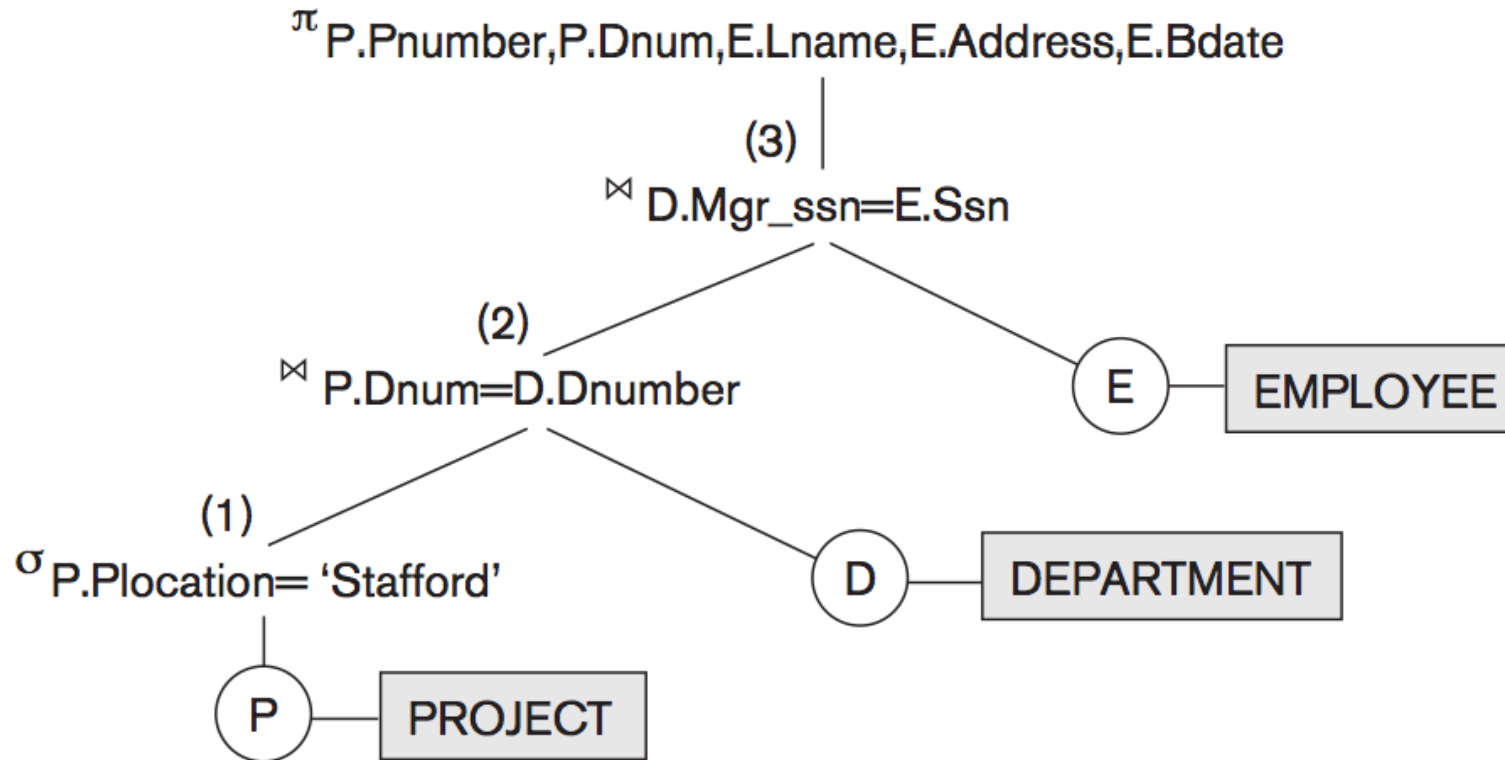
Figure 19.4

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



Initial (canonical) query tree for SQL query Q2.



Query tree corresponding to the relational algebra expression for Q2.

- 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 Pnumber=Pno AND Essn=Ssn
            AND Bdate > '1957-12-31';
```

Left-deep Tree Examples

Figure 19.7

Two left-deep (JOIN) query trees.

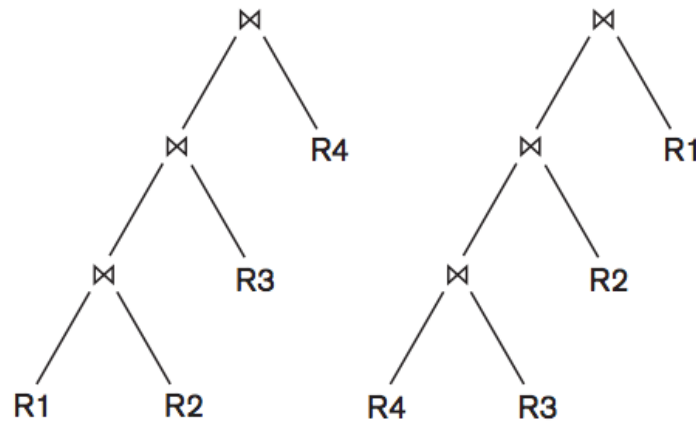
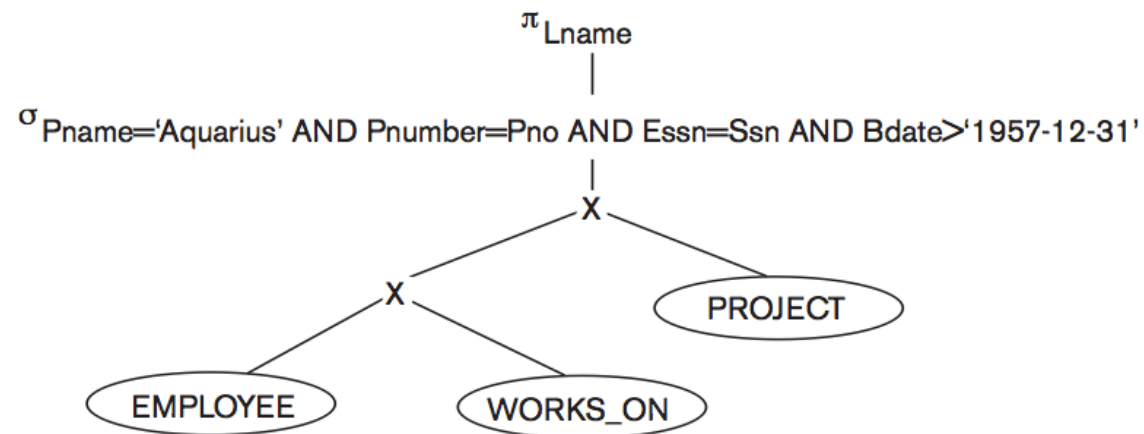


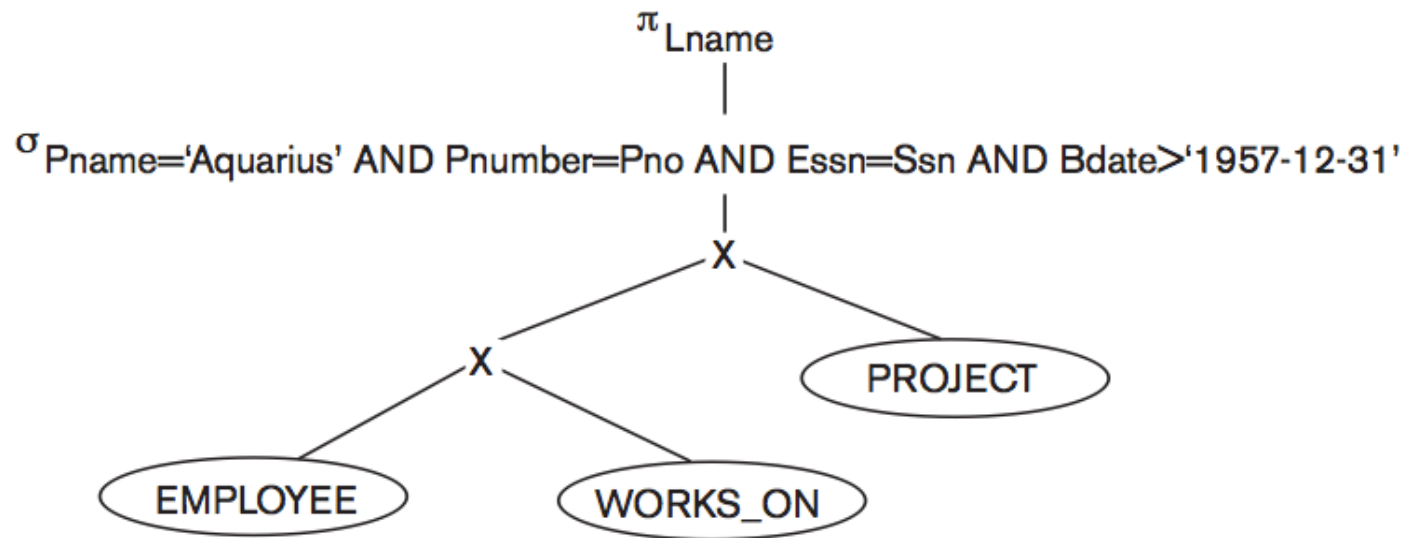
Figure 19.5 (a)

Query Q



Steps in converting a query tree during heuristic optimization

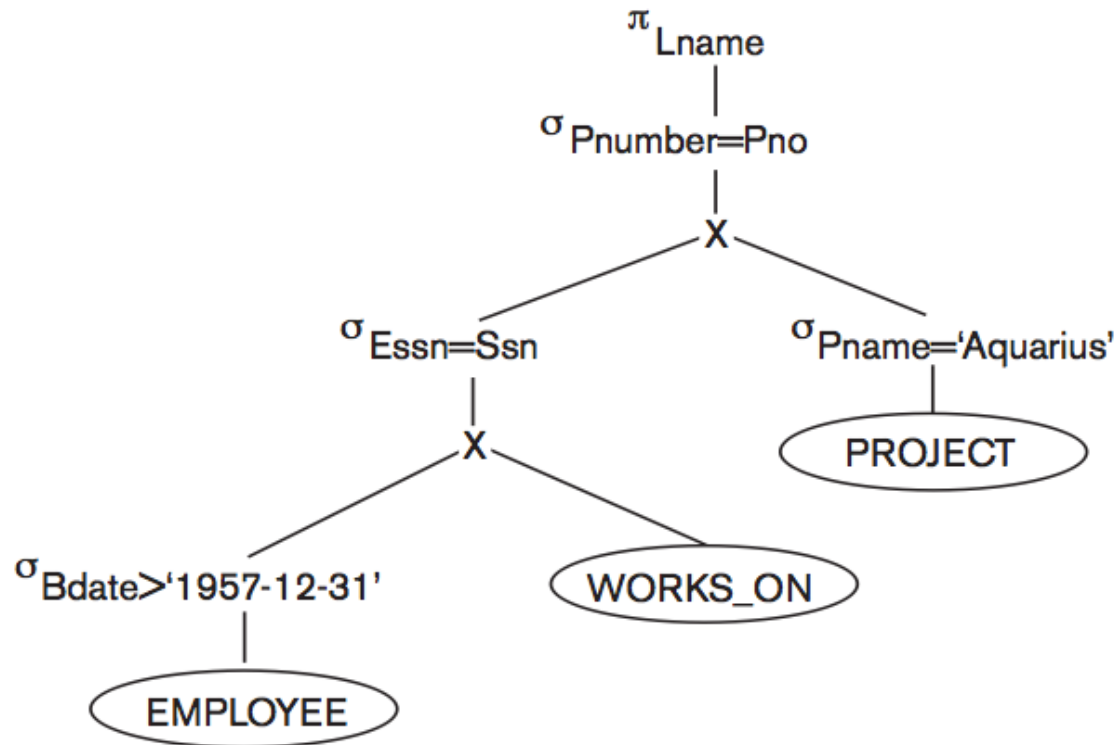
(a)



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

Steps in converting a query tree during heuristic optimization

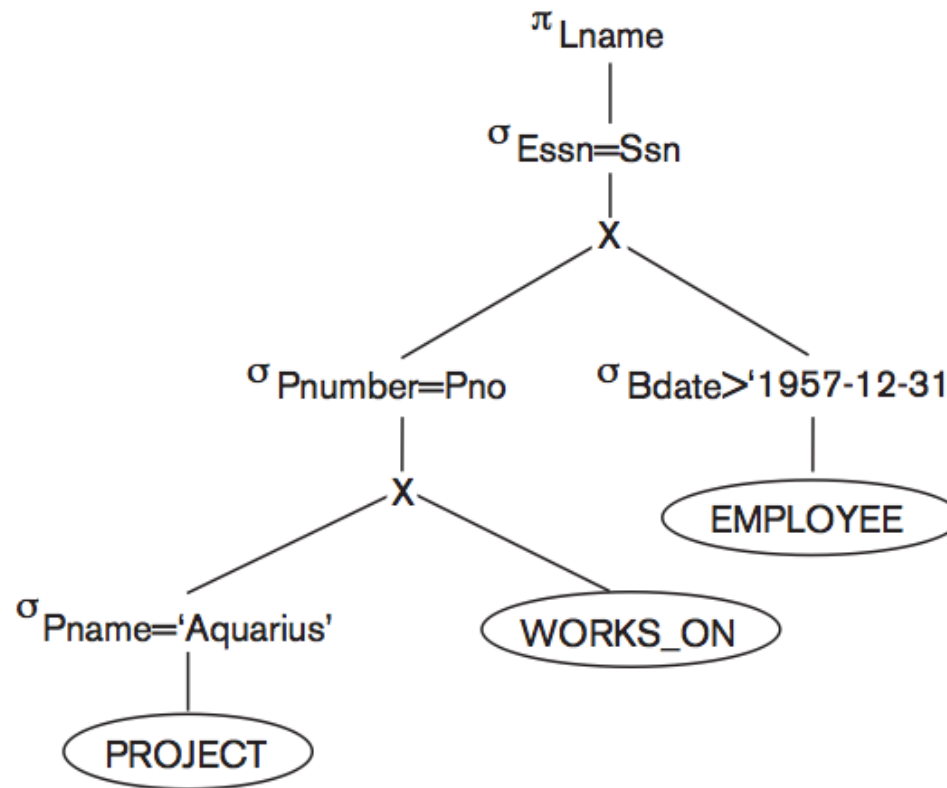
(b)



(b) Moving SELECT operations down the query tree.

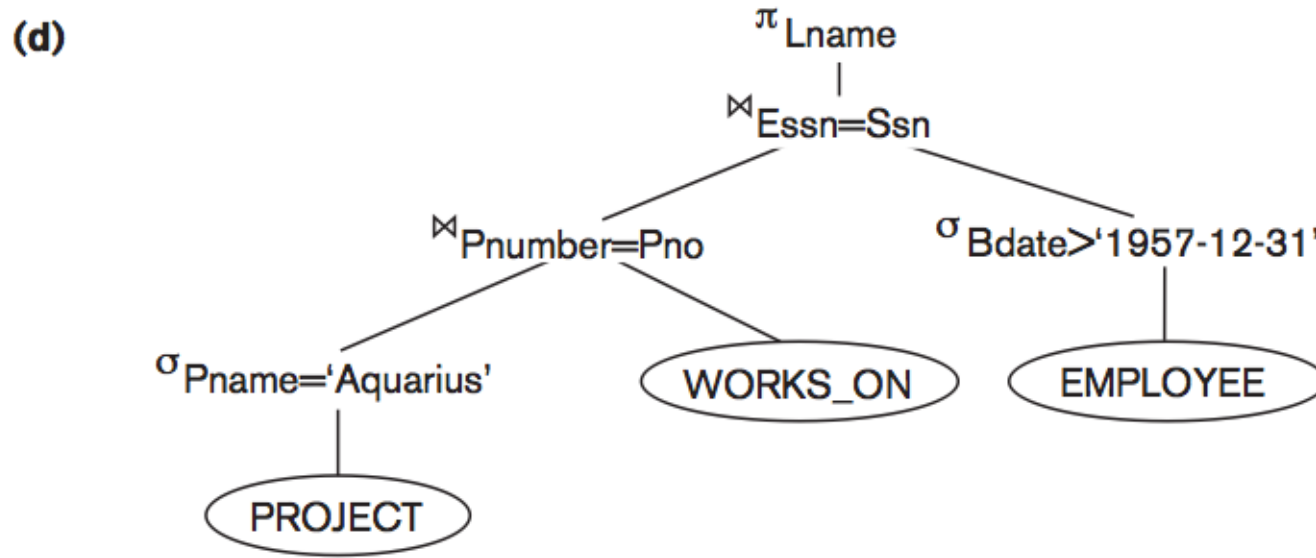
Steps in converting a query tree during heuristic optimization

(c)



(c) Applying the more restrictive **SELECT** operation first.

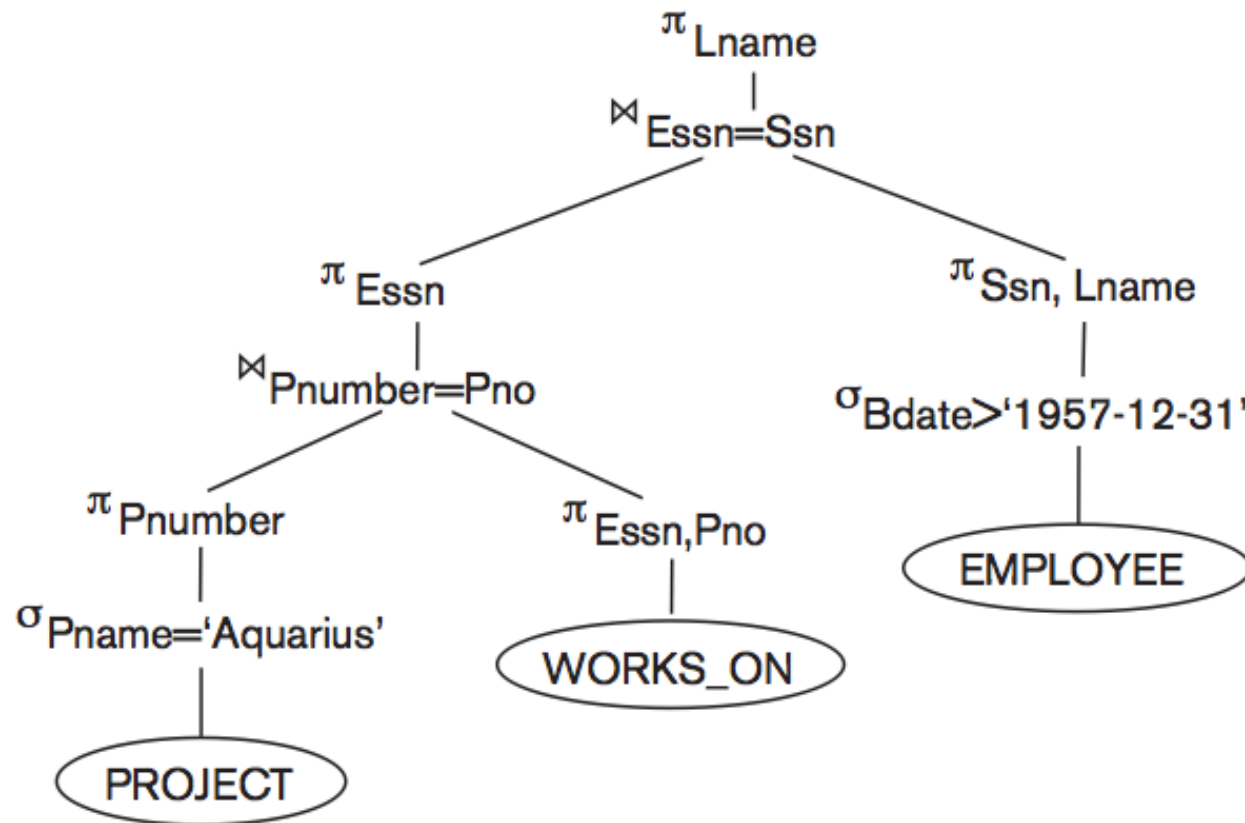
Steps in converting a query tree during heuristic optimization



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

Steps in converting a query tree during heuristic optimization

(e)



(e) Moving **PROJECT** operations down the query tree.

-
- General Transformation Rules for Relational Algebra Operations:

■ General Transformation Rules for Relational Algebra Operations:

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

$$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n}(R) \equiv \sigma_{c_1} (\sigma_{c_2} (\dots (\sigma_{c_n}(R)) \dots))$$

■ General Transformation Rules for Relational Algebra Operations:

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

$$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$

2. **Commutativity of σ .** The σ operation is commutative:

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

■ General Transformation Rules for Relational Algebra Operations:

- 1. Cascade of σ** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:

$$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$

- 2. Commutativity of σ .** The σ operation is commutative:

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

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

$$\pi_{\text{List}_1}(\pi_{\text{List}_2}(\dots(\pi_{\text{List}_n}(R))\dots)) \equiv \pi_{\text{List}_1}(R)$$

■ General Transformation Rules for Relational Algebra Operations:

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

$$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$

2. **Commutativity of σ .** The σ operation is commutative:

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

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

$$\pi_{\text{List}_1}(\pi_{\text{List}_2}(\dots(\pi_{\text{List}_n}(R))\dots)) \equiv \pi_{\text{List}_1}(R)$$

4. **Commuting σ with π .** If the selection condition c involves only those attributes A_1, \dots, A_n in the projection list, the two operations can be commuted:

$$\pi_{A_1, A_2, \dots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \dots, A_n}(R))$$

■ General Transformation Rules for Relational Algebra Operations:

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

$$R \bowtie_c S \equiv S \bowtie_c R$$

$$R \times S \equiv S \times R$$

Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.

■ General Transformation Rules for Relational Algebra Operations:

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

$$R \bowtie_c S \equiv S \bowtie_c R$$

$$R \times S \equiv S \times R$$

Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.

- 6. Commuting σ 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:

$$\sigma_c (R \bowtie S) \equiv (\sigma_c (R)) \bowtie S$$

Alternatively, if the selection condition c can be written as $(c_1 \text{ AND } c_2)$, where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S , the operations commute as follows:

$$\sigma_c (R \bowtie S) \equiv (\sigma_{c_1} (R)) \bowtie (\sigma_{c_2} (S))$$

The same rules apply if the \bowtie is replaced by a \times operation.

■ General Transformation Rules for Relational Algebra Operations:

7. Commuting π with \bowtie (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:

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

If the join condition c contains additional attributes not in L , these must be added to the projection list, and a final π operation is needed. For example, if attributes A_{n+1}, \dots, A_{n+k} of R and B_{m+1}, \dots, B_{m+p} of S are involved in the join condition c but are not in the projection list L , the operations commute as follows:

$$\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A_1, \dots, A_n, A_{n+1}, \dots, A_{n+k}} (R)) \bowtie_c (\pi_{B_1, \dots, B_m, B_{m+1}, \dots, B_{m+p}} (S)))$$

For \times , there is no condition c , so the first transformation rule always applies by replacing \bowtie_c with \times .

■ General Transformation Rules for Relational Algebra Operations:

7. Commuting π with \bowtie (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:

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

If the join condition c contains additional attributes not in L , these must be added to the projection list, and a final π operation is needed. For example, if attributes A_{n+1}, \dots, A_{n+k} of R and B_{m+1}, \dots, B_{m+p} of S are involved in the join condition c but are not in the projection list L , the operations commute as follows:

$$\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A_1, \dots, A_n, A_{n+1}, \dots, A_{n+k}} (R)) \bowtie_c (\pi_{B_1, \dots, B_m, B_{m+1}, \dots, B_{m+p}} (S)))$$

For \times , there is no condition c , so the first transformation rule always applies by replacing \bowtie_c with \times .

8. Commutativity of set operations. The set operations \cup and \cap are commutative but $-$ is not.

- General Transformation Rules for Relational Algebra Operations:

9. Associativity of \bowtie , \times , \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:

$$(R \theta S) \theta T \equiv R \theta (S \theta T)$$

■ General Transformation Rules for Relational Algebra Operations:

9. Associativity of \bowtie , \times , \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:

$$(R \theta S) \theta T \equiv R \theta (S \theta T)$$

10. Commuting σ with set operations. The σ operation commutes with \cup , \cap , and $-$. If θ stands for any one of these three operations (throughout the expression), we have:

$$\sigma_c(R \theta S) \equiv (\sigma_c(R)) \theta (\sigma_c(S))$$

■ General Transformation Rules for Relational Algebra Operations:

9. Associativity of \bowtie , \times , \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:

$$(R \theta S) \theta T \equiv R \theta (S \theta T)$$

10. Commuting σ with set operations. The σ operation commutes with \cup , \cap , and $-$. If θ stands for any one of these three operations (throughout the expression), we have:

$$\sigma_c (R \theta S) \equiv (\sigma_c (R)) \theta (\sigma_c (S))$$

11. The π operation commutes with \cup .

$$\pi_L (R \cup S) \equiv (\pi_L (R)) \cup (\pi_L (S))$$

■ General Transformation Rules for Relational Algebra Operations:

- 9. Associativity of \bowtie , \times , \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:

$$(R \theta S) \theta T \equiv R \theta (S \theta T)$$

- 10. Commuting σ with set operations.** The σ operation commutes with \cup , \cap , and $-$. If θ stands for any one of these three operations (throughout the expression), we have:

$$\sigma_c (R \theta S) \equiv (\sigma_c (R)) \theta (\sigma_c (S))$$

- 11. The π operation commutes with \cup .**

$$\pi_L (R \cup S) \equiv (\pi_L (R)) \cup (\pi_L (S))$$

- 12. Converting a (σ, \times) sequence into \bowtie .** If the condition c of a σ that follows a \times corresponds to a join condition, convert the (σ, \times) sequence into a \bowtie as follows:

$$(\sigma_c (R \times S)) \equiv (R \bowtie_c S)$$

■ 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.

- 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.)

19.8 – Using Selectivity and Cost Estimates in Query Optimization

Using Selectivity and Cost Estimates in Query Optimization



- **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)
- **Considerations and Issues**
 - Cost function
 - Number of execution strategies to be considered

Using Selectivity and Cost Estimates in Query Optimization



- **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.

- 1) Access cost to secondary storage
 - This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers.
 - This is also known as disk I/O (input/output) cost.
 - The cost of searching for records in a disk file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes.
 - In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

- 2) Disk storage cost

- This is the cost of storing any intermediate files on disk that are generated by an execution strategy for the query.

- 3) Computation cost

- This is the cost of performing in-memory operations on the records within the data buffers during query execution.
- Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values.
- This is also known as *CPU (central processing unit)* cost.

- **4) Memory usage cost**
 - This is the cost pertaining to the number of main memory buffers needed during query execution.
- **5) Communication cost**
 - This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated.
 - In distributed databases (see Chapter 25), it would also include the cost of transferring tables and results among various computers during query evaluation.

Using Selectivity and Cost Estimates in Query Optimization



- Catalog Information used in Cost Functions
 - To estimate the costs of various execution strategies, we must keep track of any information that is needed for the cost functions.
 - This information may be stored in the DBMS catalog, where it is accessed by the query optimizer.
 - **First**, we must know the size of each file. For a file whose records are all of the same type, the
 - **number of records (tuples) (r)**,
 - the (average) **record size (R)**, and
 - the **number of file blocks (b)** (or close estimates of them) are needed.
 - The **blocking factor (bfr)** for the file may also be needed.

Using Selectivity and Cost Estimates in Query Optimization



- Catalog Information used in Cost Functions
- Information about the size of a **file**
 - B – block size (in kb)
 - R – record size (in kb)
 - r – number of records (tuples)
 - b – number of blocks
 - bfr – blocking factor ($\lfloor B / R \rfloor$ records per block)

Using Selectivity and Cost Estimates in Query Optimization



- Catalog Information used in Cost Functions
- Information about **indices** and **indexing attributes** of a file
 - r – Number of tuples (rows)
 - sl – Selectivity of an attribute
 - fraction of records satisfying an equality condition
 - s – Selection cardinality of an attribute. ($s = sl * r$)
 - d – Number of distinct values of an attribute
 - x – Number of levels of each multilevel index
 - b_{I1} – Number of first-level index blocks
 - i.e. “root” level

Using Selectivity and Cost Estimates in Query Optimization

- Catalog Information used in Cost Functions
- Information about **joins**
 - $|R \bowtie S|$, $|R * S|$, $|R \times S|$ – join cardinality
 - js – join selectivity
 - The ratio of the expected size of the join result divided by the maximum size $n_R * n_S$ of the join

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;

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 $<=$,
 - $C_{S6a} = x + (b_{I1}/2) + (r/2)$

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.

Examples of Cost Functions for JOIN (contd.)

- **J1. Nested-loop join:**
 - Using R for outer loop
 - $C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|) / bfr_{RS})$
- **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

Examples of Cost Functions for JOIN (contd.)

■ J2. Single-loop join (contd.)

■ For a **secondary index** on S ,

$$\square C_{J2a} = b_R + (|R| * (x_B + 1 + s_B)) + ((j_s * |R| * |S|) / bfr_{RS})$$

■ For a **clustering index** on S ,

$$\square C_{J2b} = b_R + (|R| * (x_B + (s_B / bfr_B))) + ((j_s * |R| * |S|) / bfr_{RS});$$

■ For a **primary index** on S ,

$$\square C_{J2c} = b_R + (|R| * (x_B + 1)) + ((j_s * |R| * |S|) / bfr_{RS})$$

■ If a hash key exists for one of the two join attributes — B of S

$$\square C_{J2d} = b_R + (|R| * h) + ((j_s * |R| * |S|) / bfr_{RS})$$

■ J3. Sort-merge join:

$$\square C_{J3a} = C_S + b_R + b_S + ((j_s * |R| * |S|) / bfr_{RS})$$

$$\square (CS: \text{Cost for sorting files})$$

Example: Join Cost

- Suppose that we have the EMPLOYEE file described in the example in the previous section (of the textbook, per p.717), and
- Assume that the DEPARTMENT file in Figure 3.5 consists of $r_D = 125$ records stored in $b_D = 13$ disk blocks
 - EMPLOYEE cost values are from the table in Figure 19.8 (book never says so)
 - DEPARTMENT cost values are given (different than Figure 19.8)
- Consider the following two join operations:
 - OP6: $\text{EMPLOYEE} \bowtie_{\text{Dno=Dnumber}} \text{DEPARTMENT}$
 - OP7: $\text{DEPARTMENT} \bowtie_{\text{Mgr_ssn=Ssn}} \text{EMPLOYEE}$

Example: Join Cost

- Suppose that we
 - primary index on Dnumber of DEPARTMENT with $x_{\text{Dnumber}} = 1$ level
 - secondary index on Mgr_ssn of DEPARTMENT with selection cardinality $s_{\text{Mgr_ssn}} = 1$ and levels $x_{\text{Mgr_ssn}} = 2$.
- Assume that the join selectivity for OP6 is
$$js_{\text{OP6}} = (1 / |\text{DEPARTMENT}|) = 1 / 125$$
 - because Dnumber is a key of DEPARTMENT.
- Also assume that the blocking factor for the resulting join file is
$$bfr_{\text{ED}} = 4 \text{ records per block}$$

Addendum Slides

Example of Using the Cost Functions. Suppose that we have the EMPLOYEE file described in the example in the previous section, and assume that the DEPARTMENT file in Figure 3.5 consists of $r_D = 125$ records stored in $b_D = 13$ disk blocks. Consider the following two join operations:

OP6: EMPLOYEE $\bowtie_{Dno=Dnumber}$ DEPARTMENT
 OP7: DEPARTMENT $\bowtie_{Mgr_ssn=Ssn}$ EMPLOYEE

Suppose that we have a primary index on Dnumber of DEPARTMENT with $x_{Dnumber} = 1$ level and a secondary index on Mgr_ssn of DEPARTMENT with selection cardinality $s_{Mgr_ssn} = 1$ and levels $x_{Mgr_ssn} = 2$. Assume that the join selectivity for OP6 is $js_{OP6} = (1/|DEPARTMENT|) = 1/125$ because Dnumber is a key of DEPARTMENT. Also assume that the blocking factor for the resulting join file is $bfr_{ED} = 4$ records per block. We can estimate the worst-case costs for the JOIN operation OP6 using the applicable methods J1 and J2 as follows:

1. Using method J1 with EMPLOYEE as outer loop:

$$\begin{aligned} C_{J1} &= b_E + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 2000 + (2000 * 13) + (((1/125) * 10,000 * 125)/4) = 30,500 \end{aligned}$$

2. Using method J1 with DEPARTMENT as outer loop:

$$\begin{aligned} C_{J1} &= b_D + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 13 + (13 * 2000) + (((1/125) * 10,000 * 125)/4) = 28,513 \end{aligned}$$

3. Using method J2 with EMPLOYEE as outer loop:

$$\begin{aligned} C_{J2c} &= b_E + (r_E * (x_{Dnumber} + 1)) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 2000 + (10,000 * 2) + (((1/125) * 10,000 * 125)/4) = 24,500 \end{aligned}$$

4. Using method J2 with DEPARTMENT as outer loop:

$$\begin{aligned} C_{J2a} &= b_D + (r_D * (x_{Dno} + s_{Dno})) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 13 + (125 * (2 + 80)) + (((1/125) * 10,000 * 125)/4) = 12,763 \end{aligned}$$

1. Using method J1 with EMPLOYEE as outer loop:

$$\begin{aligned} C_{J1} &= b_E + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 2000 + (2000 * 13) + (((1/125) * 10,000 * 125)/4) = 30,500 \end{aligned}$$

2. Using method J1 with DEPARTMENT as outer loop:

$$\begin{aligned} C_{J1} &= b_D + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 13 + (13 * 2000) + (((1/125) * 10,000 * 125)/4) = 28,513 \end{aligned}$$

3. Using method J2 with EMPLOYEE as outer loop:

$$\begin{aligned} C_{J2c} &= b_E + (r_E * (x_{Dnumber} + 1)) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 2000 + (10,000 * 2) + (((1/125) * 10,000 * 125)/4) = 24,500 \end{aligned}$$

4. Using method J2 with DEPARTMENT as outer loop:

$$\begin{aligned} C_{J2a} &= b_D + (r_D * (x_{Dno} + s_{Dno})) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 13 + (125 * (2 + 80)) + (((1/125) * 10,000 * 125)/4) = 12,763 \end{aligned}$$

$x_{Dno} + 1$

- 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.

- **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.