Chapter 15, Algorithms for Query Processing and Optimization

- A query expressed in a high-level query language such as SQL must be **scanned, parsed, and validate**.
- Scanner: identify the language tokens.
- Parser: check query syntax.
- Validate: check all attribute and relation names are valid.
- An internal representation (**query tree or query graph**) of the query is created after scanning, parsing, and validating.
- Then DBMS must devise an **execution strategy** for retrieving the result from the database files.
- How to choose a suitable (efficient) strategy for processing a query is known as **query optimization**.
- The term optimization is actually a misnomer because in some cases the chosen execution plan is not the optimal strategy – it is just a reasonably efficient one.
- There are two main techniques for implementing query optimization.
  - **Heuristic rules** for re-ordering the operations in a query.
  - **Systematically estimating** the cost of different execution strategies and choosing the lowest cost estimate.

15.1 Translating SQL Queries into Relation Algebra

- An SQL query is first translated into an equivalent extended relation algebra expression (as a query tree) that is then optimized.

- **Query block** in SQL: 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 clauses.
  - Consider the following SQL query.

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

  - We can decompose it into two blocks:

Inner block:

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

Outer block:

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

  - Then translate to algebra expressions:
    * Inner block: $\Im_{MAX\ SALARY}(\sigma_{DNO=5}\ EMPLOYEE)$
    * Outer block: $\pi_{LNAME,\ FNAME}(\sigma_{SALARY>c}\ EMPLOYEE)$

15.2 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.
- Use a **sort-merge strategy**, which starts by sorting small subfiles – called **runs** – of the main file and merges the sorted runs, creating larger sorted subfiles that are merged in turn.
- The algorithm consists of two phases: sorting phase and merging phase.
- **Sorting phase:**

- Runs of the file that can fit in the available buffer space are read into main memory, sorted using an internal sorting algorithm, and written back to disk as temporary sorted subfiles (or runs).
- **number of initial runs (n_R), number of file blocks (b), and available buffer space (n_b)**
- $n_R = \lceil b/n_B \rceil$
- If the available buffer size is 5 blocks and the file contains 1024 blocks, then there are 205 initial runs each of size 5 blocks. After the sort phase, 205 sorted runs are stored as temporary subfiles on disk.

- **Merging phase:**
  - The sorted runs are merged during one or more **passes**.
  - The **degree of merging (d_M)** 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.
  - $d_M = MIN\{n_B - 1, n_R\}$, and the number of passes is $\lceil log_{d_M}(n_R) \rceil$.
  - In previous example, $d_M = 4$, 205 runs $\rightarrow$ 52 runs $\rightarrow$ 13 runs $\rightarrow$ 4 runs $\rightarrow$ 1 run. This means 4 passes.
- The complexity of external sorting (number of block accesses): $(2 \times b) + (2 \times (b \times (log_{d_M}\ b)))$
- For example:
  - 5 initial runs $[2, 8, 11]$, $[4, 6, 7]$, $[1, 9, 13]$, $[3, 12, 15]$, $[5, 10, 14]$.
  - The available buffer $n_B = 3$ blocks $\rightarrow d_M = 2$ (two way merge)
  - After first pass: 3 runs
    $[\underline{2, 4, 6}, \underline{7, 8, 11}]$, $[\underline{1, 3, 9}, \underline{12, 13, 15}]$, $[\underline{5, 10, 14}]$

- After second pass: 2 runs

  [$\underline{1,2,3}, \underline{4,6,7}, \underline{8,9,11}, \underline{12,13,15}$], [$\underline{5,10,14}$]

- After third pass:

  [$\underline{1,2,3}, \underline{4,5,6}, \underline{7,8,9}, \underline{10,11,12}, \underline{13,14,15}$]

15.3 Algorithms for SELECT and JOIN Operations

15.3.1 Implementing the SELECT Operation

- Five operations for demonstration:
  - (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): $\sigma_{ESSN='123456789'\ and\ PNO=10}\ (WORKS_ON)$
- **Search methods for simple selection:**
  - **S1.** *Linear search (brute force)*
  - **S2.** *Binary search*: If the selection condition involves an equality comparison on a key attribute on which the file is ordered. (ex. OP1)
  - **S3.** *Using a primary index*: If the selection condition involves an equality comparison on a key attribute with a primary index. (ex. OP1)
  - **S4.** *Using a primary index to retrieve multiple records*: If the comparison condition is $>, \leq, <, \geq$ on a key field with a primary index – for example, $DNUMBER > 5$ in OP2 – use the index to find the record satisfying the equality condition ($DNUMBER = 5$), then retrieve all subsequent (preceding) 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 – for example, $DNO = 5$ in OP3 – use the index to retrieve all the records satisfying the condition.
- **S6.** *Using a secondary (B^+-Tree) index on an equality comparison*: Retrieve one record if the indexing field is a key, or multiple records if the indexing field is not a key. This method can also be used for comparison involved $>, \leq, <, \geq$.

- **Search methods for complex (conjunctive) selection:**
  - **S7.** *Conjunctive selection using an individual index*: 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 to retrieve the records and then check whether each retrieved record satisfies the remaining conditions in the conjunctive condition.
  - **S8.** *Conjunctive selection using a composite index*: If two or more attributes are involved in equality conditions and a composite index exists on the combined fields, we can use the index directly. For example, OP5 can use this method if there is a composite index $(ESSN, PNO)$ for WORKS_ON file.
  - **S9.** *Conjunctive selection by intersection of record pointers*: If secondary indexes are available on more than one of the fields in the conjunctive condition, and if the indexes include record pointers (rather than block pointers). The intersection of these sets of record pointers would satisfy the conjunctive condition, which are then used to retrieve those records directly. If only some of the simple conditions have secondary indexes, each retrieved record is further tested for the remaining conditions.

15.3.2 Implementing the JOIN Operation

- Two operations for demonstration:

- (OP6): $EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT$
- (OP7): $DEPARTMENT \bowtie_{MGRSSN=SSN} EMPLOYEE$

- **Methods for implementing JOINs:**

  The algorithm we consider are for join operation of the form

$$R \bowtie_{A=B} S$$

  - **J1.** *Nested-loop join (brute force)*: For each record t in R (outer loop), retrieve every record s from S 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 exists for one of the two attributes – say, B of S – retrieve each record t in R, one at a time (single loop), and then use the access structure to retrieve directly all matching records s from S that satisfy $s[B] = t[A]$.
  - **J3.** *Sort-merge join*: First to sort R and S based on the corresponding attributes A and B using external sorting. Second to merge sorted R and S by retrieving the matching records t and s that satisfy the join condition $t[A] = s[B]$.

    Note that if there are secondary indexes for R and S based on attributes A and B, then we can merge these two secondary indexes instead of sorting and merging the data files R and S.
  - **J4.** *Hash-join*:
    * **Partitioning phase:** a single pass through the file with fewer records (say, R) hashes its records (using A as the hash key) to the hash file buckets
    * **Probing phase:** a single pass through the other file (S) then hashes each of its records to probe the appropriate bucket.

15.4 Algorithms for PROJECT and SET Operations

- **PROJECT operation:**

- If the $< attribute\ list >$ of the PROJECT operation $\pi_{<attribute\ list>}(R)$ includes a key of R, then the number of tuples in the projection result is equal to the number of tuples in R, but only with the values for the attributes in $< attribute\ list >$ in each tuple.
- If the $< attribute\ list >$ does not contain a key of R, duplicate tuples must be eliminated. The following methods can be used to eliminate duplication.
  * Sorting the result of the operation and then eliminating duplicate tuples.
  * Hashing the result of the operation into hash file in memory and check each hashed record against those in the same bucket; if it is a duplicate, it is not inserted.

- **CARTESIAN PRODUCT operation:**
  - It is an extremely expensive operation. Hence, it is important to avoid this operation and to substitute other equivalent operations during optimization.
- **UNION, INTERSECTION, and DIFFERENCE operations:**
  - Use variations of the **sort-merge** technique.
  - Use Hashing technique. That is, first hash (partition) one file, then hash (probe) another file.

15.5 Implementing Aggregate Operations and OUTER JOINS

15.5.1 Implementing Aggregate Operations

- The aggregate operators (MIN, MAX, COUNT, AVERAGE, SUM), when applied to an entire table, can be computed by a **table scan** or by using an appropriate index.
- **SELECT** **MAX**(SALARY)
  FROM EMPLOYEE;

If an (ascending) index on SALARY exists for the EMPLOYEE relation, the optimizer can decide the largest value: the rightmost leaf (B-Tree and B^+-Tree) or the last entry in the first level index (clustering, secondary).

- The index could also be used for COUNT, AVERAGE, and SUM aggregates, if it is a **dense index**. For a **nondense index**, the actual number of records associated with each index entry must be used for a correct computation.
- For a GROUP BY clause in a query, the technique to use is to partition (either **sorting** or **hashing**) the relation on the grouping attributes and then to apply the aggregate operators on each group.
  - If a clustering index exists on the grouping attributes, then the records are already partitioned.

15.5.2 Implementing Outer Join

- Outer join can be computed by modifying one of the join algorithms, such as nested-loop join or single-loop join.
  - For a left (right) outer join, we use the left (right) relation as the outer loop or single-loop in the join algorithms.
  - If there are matching tuples in the other relation, the joined tuples are produced and saved in the result. However, if no matching tuple is found, the tuple is still included in the result but is padded with null values.
- The other join algorithms, **sort-merge and hash-join**, can also be extended to compute outer joins.

15.6 Combining Operations Using Pipelining

- A query is typically a sequence of relational operations. If we execute a single operation at a time, we must generate temporary files on disk to hold the results of these temporary operations, creating excessive overhead.

- **Pipelining** processing: Instead of generating temporary files on disk, the result tuples from one operation are provided directly as input for subsequent operations.

15.7 Using Heuristics in Query Optimization

- Using heuristic rules to modify the internal representation (query tree) of a query.
- One of the main heuristic rules is to apply SELECT and PROJECT operations before applying the JOIN or other binary operations.
- The SELECT and PROJECT operations reduce the size of a file and hence should be applied first.

15.7.1 Notation for Query Trees and Query Graphs

- **Query tree:** see Figure 15.4(a) (Fig 18.4(a) on e3).
  - A tree data structure that corresponds to a relational algebra expression. The input relations of the query – leaf nodes; the relational algebra operations – 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.
  - A query tree specifies a specific order of operations for executing a query.
- **Query Graph:** see Figure 15.4(c) (Fig 18.4(c) on e3).
  - **Relation nodes:** displayed as single circles.
  - **Constant nodes:** displayed as double circles.
  - **Graph edges:** specify the selection and join conditions.
  - The attributes to be retrieved from each relation are displayed in square brackets above each relation.

  - The query graph does not indicate an order on which operations to perform first. There is only a single graph corresponding to each query. Hence, a query graph corresponds to a *relational calculus* expression.

15.7.2 Heuristic Optimization of Query Trees

- Many different relational algebra expressions – and hence many different query trees – can be equivalent.
- The query parser will typically generate a standard initial query tree to correspond to an SQL query, without doing any optimization.
- The **initial query tree (canonical query tree)** is generated by the following sequence.
  - The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied.
  - The selection and join conditions of the WHERE clause are applied.
  - The projection on the SELECT clause attributes are applied.
- The heuristic query optimizer transform this initial query tree (inefficient) to a final query tree that is efficient to execute.
- Example of transforming a query. See Figure 15.5 (Fig 18.5 on e3).
  Consider the SQL query below.

```
SELECT  LNAME
FROM    EMPLOYEE, WORKS_ON, PROJECT
WHERE   PNAME='Aquarius' and PNUMBER=PNO and ESSN=SSN
        and BDATE > '1957-12-31';
```

- **General transformation rules for relational Algebra operations:**

- **1. Cascade of σ:**

  $\sigma_{\ c_1\ and\ c_2\ and\ \ldots\ and\ c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$

- **2. Commutativity of σ:**

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

- **3. Cascade of π:**

  $\pi_{List1}(\pi_{List2}(\ldots(\pi_{Listn}(R))\ldots)) \equiv \pi_{List1}(R)$

- **4. Commuting σ with π:** If the selection condition c involves only those attributes $A_1, A_2, \ldots, A_n$ in the projection list.

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

- **5. Commutativity of $\bowtie$ (and $\times$):**

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

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

- **6. Commuting σ with $\bowtie$ (or $\times$):** If the selection condition c can be written as $(c_1\ and\ c_2)$, where c_1 involves only the attributes of R and c_2 involves only the attributes of S.

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

- **7. Commuting π with $\bowtie$ (or $\times$):** Suppose the projection list $L = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$, where $A_1, \ldots, A_n$ are attributes of R and $B_1, \ldots, B_m$ are attributes of S

  * If the join condition c involves only attributes in L.

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

  * If the join condition c contains additional attributes not in L. For example, $A_{n+1}, \ldots, A_{n+k}$ of R and $B_{m+1}, \ldots, B_{m+p}$ of S.

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

- **8. Commutativity of set operations:** $\cap$ and $\cup$ are commutative, but not $-$.

- **9. Associativity of $\bowtie, \times, \cap, \cup$:** Let θ be one of the four operations.

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

- **10. Commuting σ with set operations:** Let θ be one of the three set operations $\cap$, $\cup$, and $-$.

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

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

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

- **12. Converting a $(\sigma, \times)$ sequence into $\bowtie$:**

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

- Another possible transformations such as DeMorgan's law:

  $not\ (c_1\ and\ c_2) \equiv (not\ c_1)\ or\ (not\ c_2)$

  $not\ (c_1\ or\ c_2) \equiv (not\ c_1)\ and\ (not\ c_2)$

- **Outline of a heuristic algebraic optimization algorithm:**

  - **1. Break up the SELECT operations:**

    Using rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations.

  - **2. Push down the SELECT operations:**

    Using rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the tree as is permitted by the attributes involved in the select condition.

  - **3. Rearrange the leaf nodes:**

    Using rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria.

    * 1) Position the leaf node relations with most restrictive SELECT operations so they are executed first in the query tree.
    * 2) Make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations.

  - **4. Change CARTESIAN PRODUCT to JOIN operations:**

    Using rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation.

– **5. Break up and push down PROJECT operations:**

  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 for pipelining:**

  Identify subtrees that represent groups of operations that can be executed by a single algorithm.

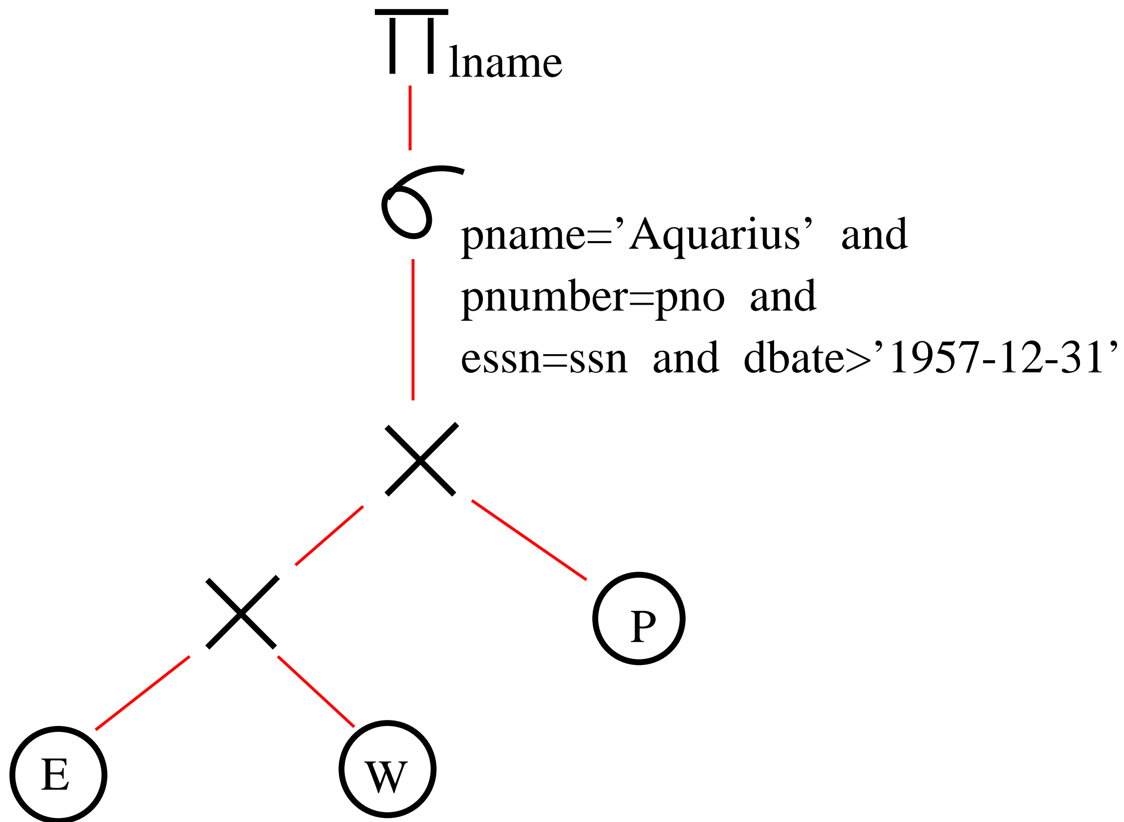
- Example for transforming SQL query ⇒ Initial query tree ⇒ Optimized query tree.

SELECT lname
FROM employee, works_on, project
WHERE pname='Aquarius' **and** pnumber=pno **and** essn=ssn
and bdate > '1957-12-31';

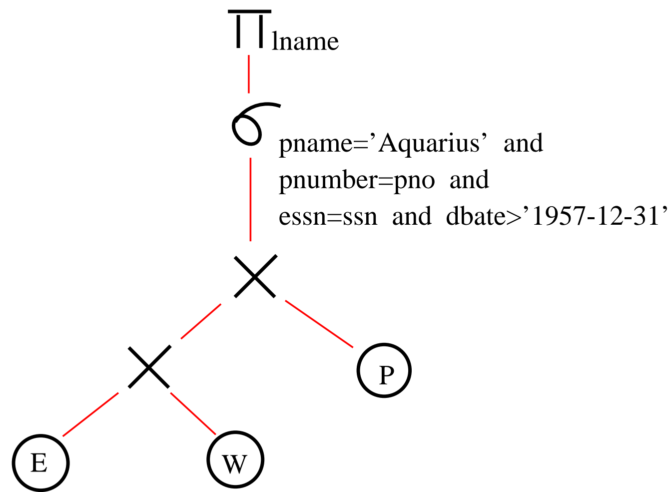
Suppose the selection cardinalities (number of tuples in the resulting relation after applying the selection operation) for all selection conditions in the above WHERE clause are as follows.

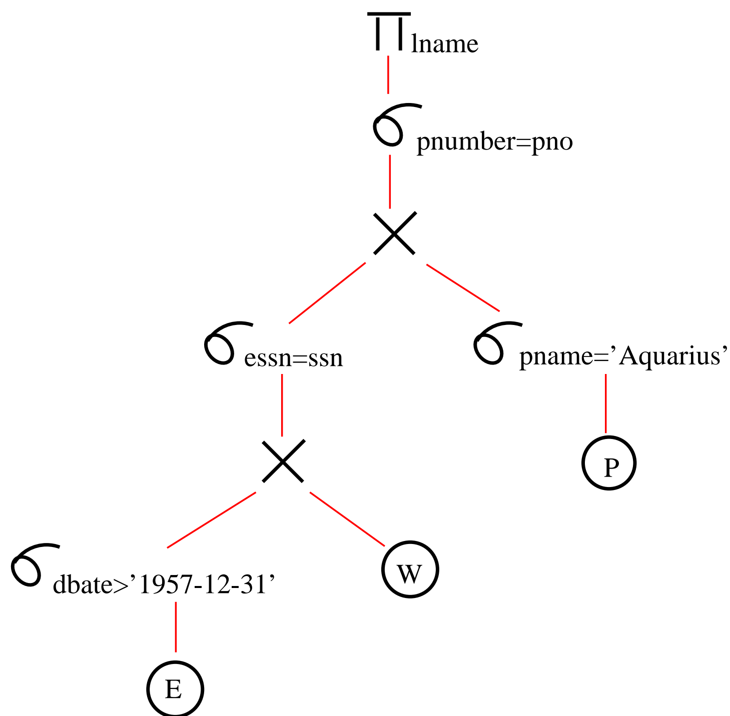
	$\sigma_{pname='Aquarius'}$	$\sigma_{bdate>'1957-12-31'}$
relation	project	employee
cardinality	1	3

– SQL query ⇒ initial query tree.

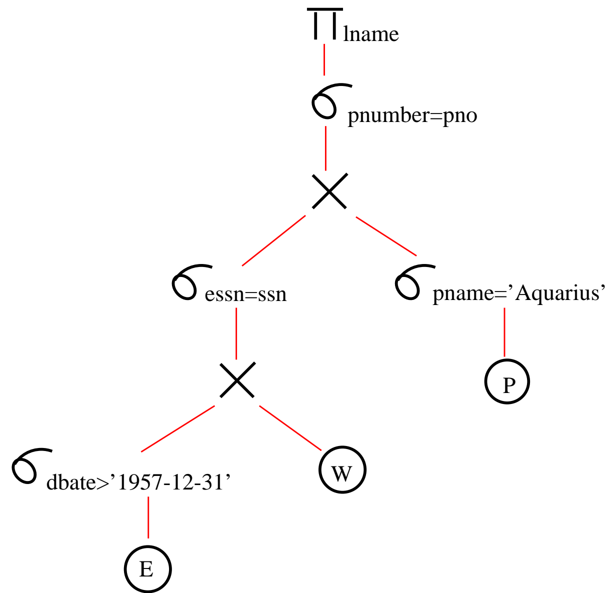


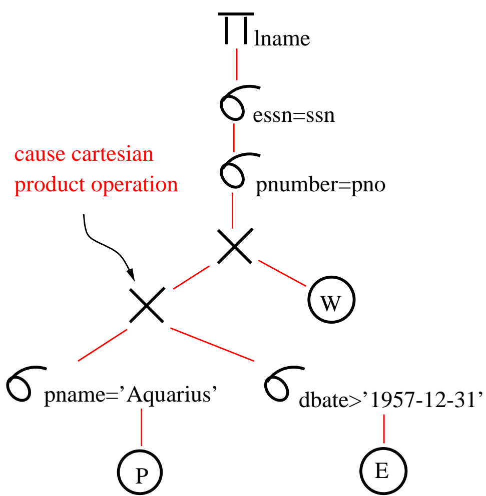
– initial query tree ⇒ query tree after pushing down selection operation.

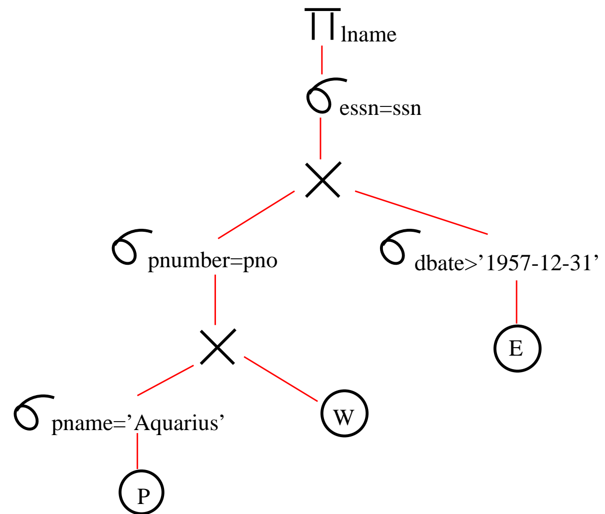




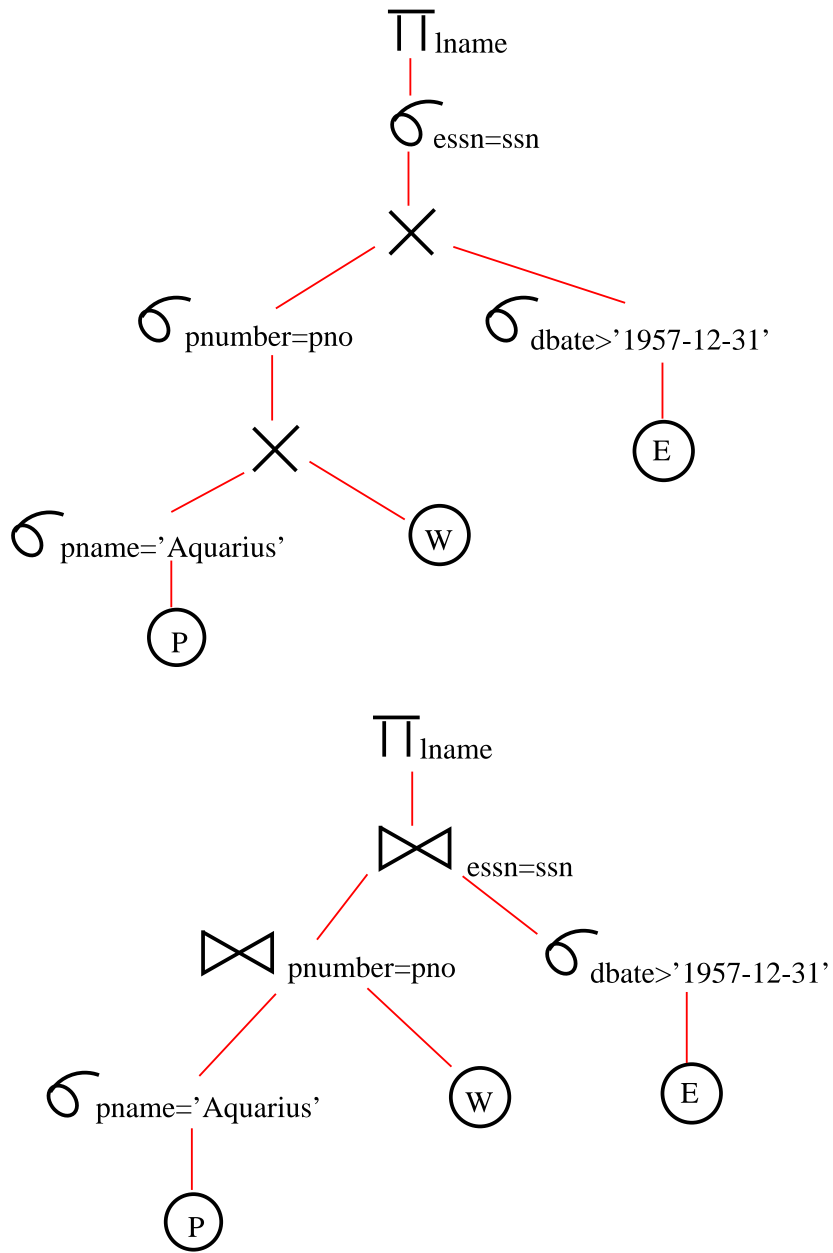
– query tree after pushing down selection ⇒ query tree after leaf nodes re-ordering.



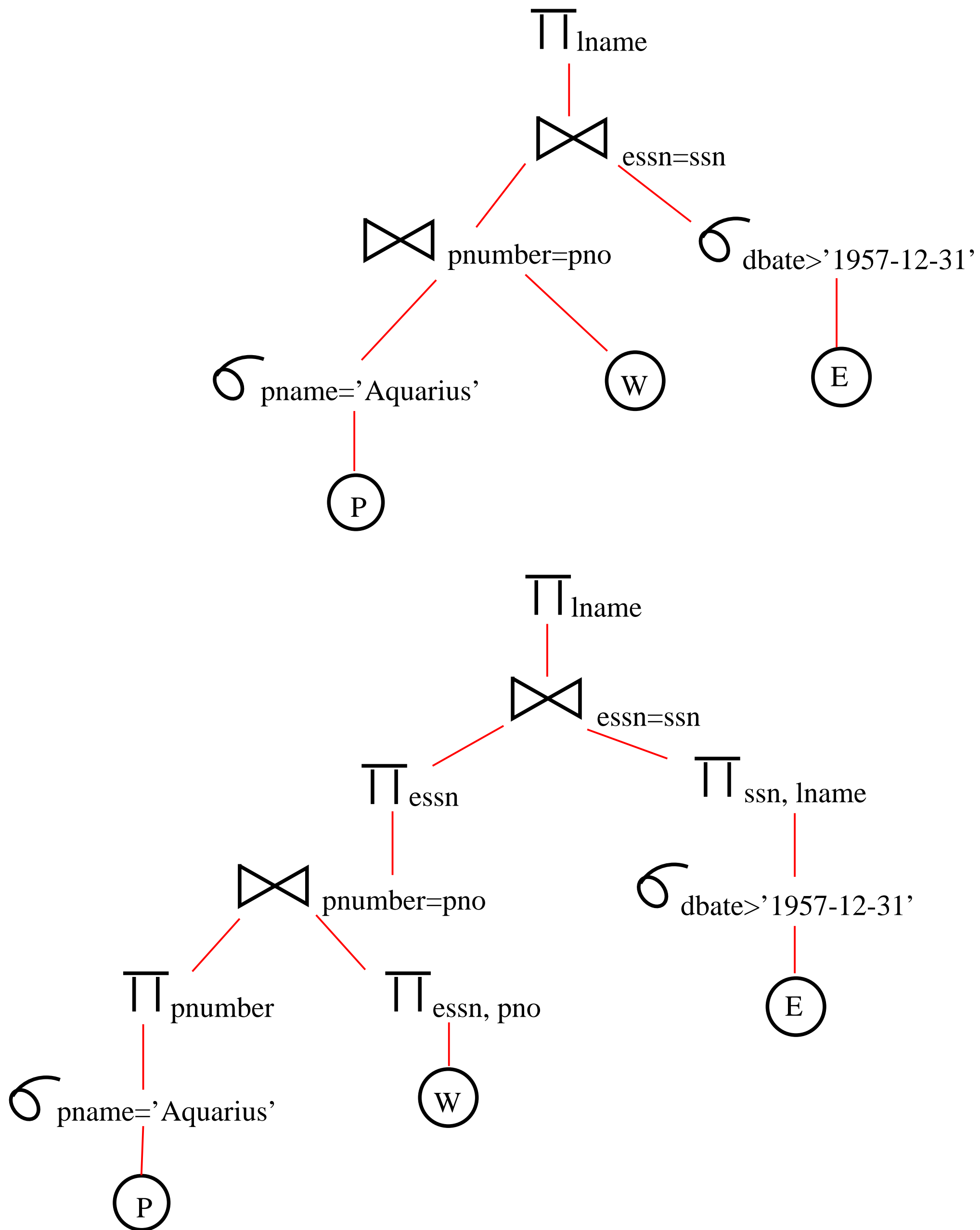




– query tree after leaf nodes re-ordering $\Rightarrow$ query tree after combining $\times, \sigma$ to $\bowtie$.



– query tree after combining $\times, \sigma$ to $\bowtie \Rightarrow$ query tree after pushing down projection operation.



15.8 Using Selectivity and Cost Estimates in Query Optimization

- A query optimizer should not depend solely on heuristic rules; it should also estimate and compare the costs of executing a query using different execution strategies and choose the lowest cost estimate.

- We need a cost function which estimates the costs of executing a query.

15.8.1 Cost Components for Query Execution

- The cost of executing a query includes the following components:

  - **1. Access cost to secondary storage:** The cost of searching for, reading, and writing data blocks that reside on secondary storage.

  - **2. Storage cost:** The cost of storing temporary files generated by an execution strategy for the query.

  - **3. Computation cost:** The cost of performing in-memory operations on the data buffers during query execution.

  - **4. Memory usage cost:** The cost of pertaining to the number of memory buffers needed during query execution.

  - **5. Communication cost:** The cost of shipping the query and its results from the database site to the site or terminal where the query originated.

- Different applications emphasize differently on individual cost components. For example,

  - For large databases, the main emphasis is on minimizing the access cost to secondary storage.

  - For smaller databases, the emphasis is on minimizing computation cost because most of the data in files involved in the query can be completely stored in memory.

- For distributed databases, communication cost must be minimized despite of other factors.
- It is difficult to include all the cost components in a weighted cost function because of the difficulty of assigning suitable weights to the cost components.

15.8.2 Catelog Information Used in Cost Functions

- The necessary information for cost function evaluation is stored in DBMS catelog.
  - The size of each file.
  - For each file, the **number of records (tuples)(r)**, the (average) **record size (R)**, the **number of blocks (b)**, and possibly the **blocking factor (bfr)**.
  - The file records may be unordered, ordered by an attribute with or without a primary or clustering index.
  - For each file, the *access methods or indexes* and the corresponding *access attributes*.
  - The **number of levels (x)** of each multilevel index is needed for cost functions that estimate the number of block accesses.
  - The **number of first-level index blocks** ($\mathbf{b}_I 1$).
  - The **number of distinct values (d)** of an attribute and its **selectivity (sl)**, which is the fraction of records satisfying an equality condition on the attribute.
    * The selectivity allows us to estimate the **selection cardinality** ($\mathbf{s = sl \times r}$) of an attribute, which is the average number of records that will satisfy an equality selectionon that attribute.
    * For a key attribute, $\mathbf{d = r, sl = 1/r}$ **and** $\mathbf{s = 1}$
    * For a nonkey attribute, by making an assumption that the **d** distinct values are uniformly distributed among the records, then $\mathbf{sl = 1/d}$ **and** $\mathbf{s = r/d}$