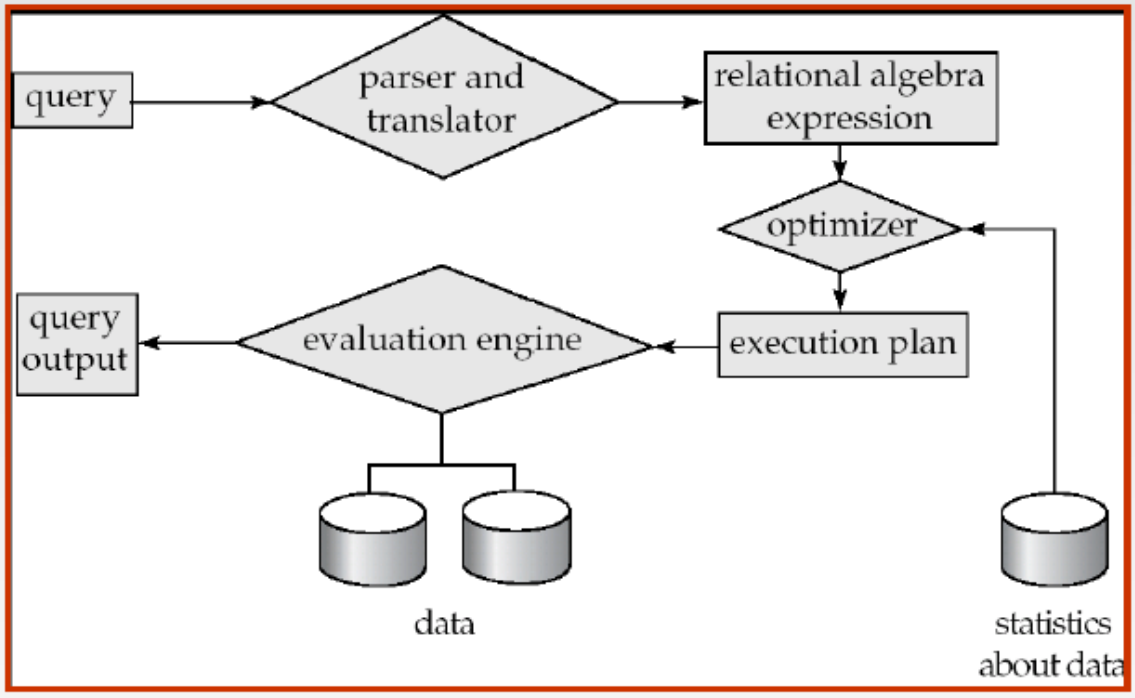
Chapter 7: Query processing

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



- **Parsing and translation:** The first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression.
- **Evaluation:** A relational-algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**. A sequence of primitive operations

that can be used to evaluate a query is a **query-execution plan** or **query-evaluation plan**.

As an illustration, consider the query

select *balance* **from** *account* **where** *balance* < 2500

Figure below illustrates an evaluation plan for our example query,

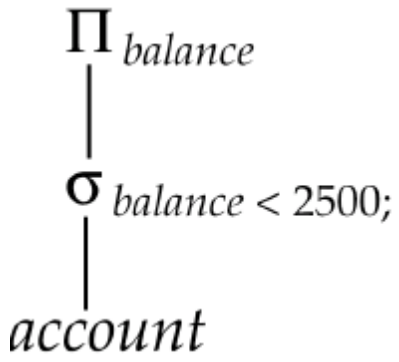


Fig: A query-evaluation plan.

The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost. Cost is estimated using statistical information from the database catalog e.g. number of tuples in each relation, size of tuples, etc. The different evaluation plans for a given query can have different costs.

Measures of Query Cost

Cost is generally measured as total elapsed time for answering query. The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication.

Typically disk access is the predominant cost, and is also relatively easy to estimate. It is measured by taking into account

Number of seeks * average-seek-cost
+ Number of blocks read * average-block-read-cost
+ Number of blocks written * average-block-write-cost

For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures

- t_T – time to transfer one block
- t_S – time for one seek
- Cost for b block transfers plus S seeks

$b * t_T + S * t_S$

We ignore CPU costs for simplicity but Real systems do take CPU cost into account. We do not include cost to writing output to disk in our cost formulae.

Query Optimization:

It is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex. We do not expect users to write their queries so that they can be processed efficiently. Rather, we expect the system to construct a query-evaluation plan that minimizes the cost of query evaluation. This is where query optimization comes into play**.** One aspect of optimization occurs at the relational-algebra level, where the system attempts to find an expression that is equivalent to the given expression, but more efficient to execute.

Consider relational-algebra expression for the query "Find the names of all customers who have an account at any branch located in Brooklyn."

$$\Pi_{customer\text{-}name}\left(\sigma_{branch-city="Brooklyn"}\left(branch \bowtie \left(account \bowtie depositor\right)\right)\right)$$

This expression constructs a large intermediate relation, *branch* ⋈ *account* ⋈ *depositor*.

However, we are interested in only a few tuples of this relation (those pertaining to branches located in Brooklyn), and in only one of the six attributes of this relation. Since we are concerned with only those tuples in the *branch* relation that pertain to branches located in Brooklyn, we do not need to consider those tuples that do not have *branch-city* = "Brooklyn". By reducing the number of tuples of the *branch* relation that we need to access, we reduce the size of the intermediate result. Our query is now represented by the relational-algebra expression

$$\Pi_{customer\text{-}name}\left(\left(\sigma_{branch\text{-}city="Brooklyn"}\left(branch\right)\right) \bowtie \left(account \bowtie depositor\right)\right)$$

which is equivalent to our original algebra expression, but which generates smaller intermediate relations. Figure below depicts the initial and transformed expressions.

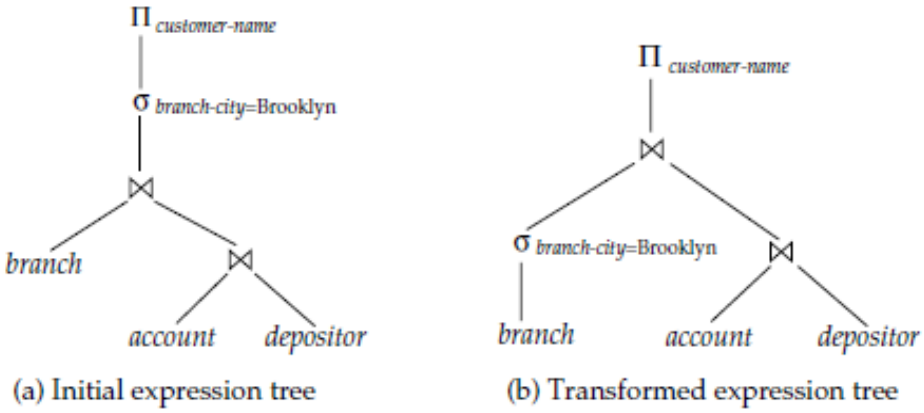


Fig: Equivalent Expressions

- Cost difference between evaluation plans for a query can be enormous e.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - ✓ Statistical information about relations. Examples: number of tuples, number of distinct values for an attribute
- Statistics estimation for intermediate results
  - ✓ to compute cost of complex expressions

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every legal database instance
  - ✓ Note: order of tuples is irrelevant
- An **equivalence rule** says that expressions of two forms are equivalent i.e can replace expression of first form by second, or vice versa.

Equivalence Rules

We now list a number of general equivalence rules on relational-algebra expressions. We use ϑ, $\vartheta 1$, $\vartheta 2$, and so on to denote predicates, $L1$, $L2$, $L3$, and so on to denote lists of attributes, and
E, $E1$, $E2$, and so on to denote relational-algebra expressions.

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections. This transformation is referred to as a cascade of σ.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are **commutative**.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the final operations in a sequence of projection operations are needed, the others can be omitted. This transformation can also be referred to as a cascade of Π.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   - **a.** $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
     This expression is just the definition of the theta join.
   - **b.** $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Theta-join operations are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. **a.** Natural-join operations are **associative**.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

   b. Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

   where θ_2 involves attributes from only E_2 and E_3. Any of these conditions may be empty; hence, it follows that the Cartesian product ($\times$) operation is also associative. The commutativity and associativity of join operations are important for join reordering in query optimization.

7. The selection operation distributes over the theta-join operation under the following two conditions:
   - **a.** It distributes when all the attributes in selection condition θ_0 involve only the attributes of one of the expressions (say, E_1) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

   - **b.** It distributes when selection condition θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

8. The projection operation distributes over the theta-join operation under the following conditions.

   a. Let L_1 and L_2 be attributes of E_1 and E_2, respectively. Suppose that the join condition θ involves only attributes in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

   b. Consider a join $E_1 \bowtie_\theta E_2$. Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively. Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. The set operations union and intersection are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

   Set difference is not commutative.

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over the union, intersection, and set–difference operations.

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

    Similarly, the preceding equivalence, with $-$ replaced with either $\cup$ or $\cap$, also holds. Further,

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

    The preceding equivalence, with $-$ replaced by $\cap$, also holds, but does not hold if $-$ is replaced by $\cup$.

12. The projection operation distributes over the union operation.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Example of Transformations: Pushing Selections

We now illustrate the use of the equivalence rules. We use our bank example with the relation schemas:

$$\begin{aligned} &\textit{Branch-schema} = (\textit{branch-name}, \textit{branch-city}, \textit{assets}) \\ &\textit{Account-schema} = (\textit{account-number}, \textit{branch-name}, \textit{balance}) \\ &\textit{Depositor-schema} = (\textit{customer-name}, \textit{account-number}) \end{aligned}$$

The relations *branch*, *account*, and *depositor* are instances of these schemas.

In our example in Section 14.1, the expression

$$\Pi_{\textit{customer-name}}(\sigma_{\textit{branch-city} = \text{"Brooklyn"}}(\textit{branch} \bowtie (\textit{account} \bowtie \textit{depositor})))$$

was transformed into the following expression,

$$\Pi_{\textit{customer-name}}((\sigma_{\textit{branch-city} = \text{"Brooklyn"}}(\textit{branch})) \bowtie (\textit{account} \bowtie \textit{depositor}))$$

which is equivalent to our original algebra expression, but generates smaller intermediate relations. We can carry out this transformation by using rule 7.a. Remember that the rule merely says that the two expressions are equivalent; it does not say that one is better than the other.

Multiple equivalence rules can be used, one after the other, on a query or on parts of the query. As an illustration, suppose that we modify our original query to restrict attention to customers who have a balance over $1000. The new relational-algebra query is

$$\Pi_{\textit{customer-name}}(\sigma_{\textit{branch-city} = \text{"Brooklyn"} \wedge \textit{balance} > 1000}(\textit{branch} \bowtie (\textit{account} \bowtie \textit{depositor})))$$

We cannot apply the selection predicate directly to the *branch* relation, since the predicate involves attributes of both the *branch* and *account* relation. However, we can first apply rule 6.a (associativity of natural join) to transform the join $\textit{branch} \bowtie (\textit{account} \bowtie \textit{depositor})$ into $(\textit{branch} \bowtie \textit{account}) \bowtie \textit{depositor}$:

$$\Pi_{\textit{customer-name}}(\sigma_{\textit{branch-city} = \text{"Brooklyn"} \wedge \textit{balance} > 1000}((\textit{branch} \bowtie \textit{account}) \bowtie \textit{depositor}))$$

Then, using rule 7.a, we can rewrite our query as

$$\Pi_{\textit{customer-name}}((\sigma_{\textit{branch-city} = \text{"Brooklyn"} \wedge \textit{balance} > 1000}(\textit{branch} \bowtie \textit{account})) \bowtie \textit{depositor})$$

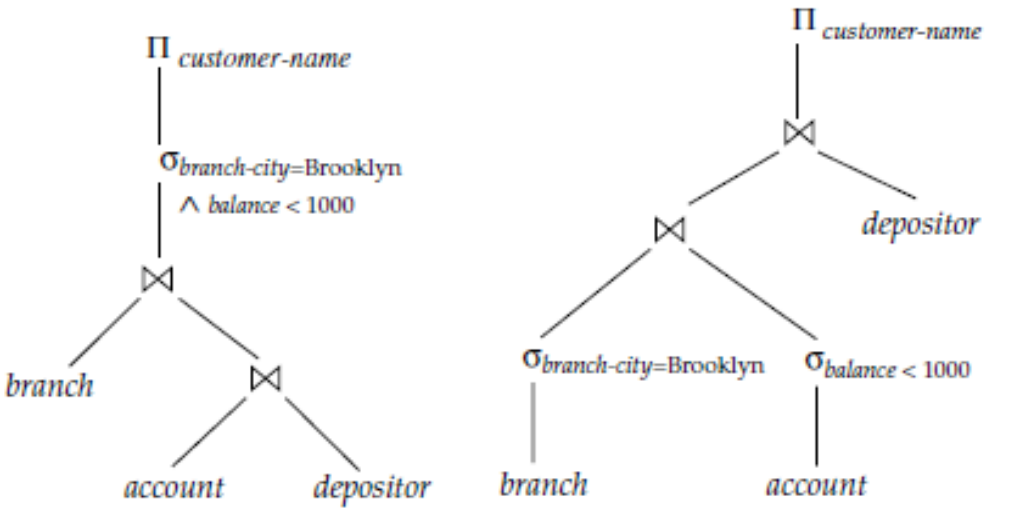
Let us examine the selection subexpression within this expression. Using rule 1, we can break the selection into two selections, to get the following subexpression:

$$\sigma_{\textit{branch-city} = \text{"Brooklyn"}}(\sigma_{\textit{balance} > 1000}(\textit{branch} \bowtie \textit{account}))$$

Both of the preceding expressions select tuples with *branch-city* = "Brooklyn" and *balance* > 1000. However, the latter form of the expression provides a new opportunity to apply the "perform selections early" rule, resulting in the subexpression

$$\sigma_{\textit{branch-city} = \text{"Brooklyn"}}(\textit{branch}) \bowtie \sigma_{\textit{balance} > 1000}(\textit{account})$$

Figure 14.3 depicts the initial expression and the final expression after all these transformations. We could equally well have used rule 7.b to get the final expression directly, without using rule 1 to break the selection into two selections. In fact, rule 7.b can itself be derived from rules 1 and 7.a



(a) Initial expression tree (b) Tree after multiple transformations

Figure 14.3 Multiple transformations.

Transformation Example: Pushing Projections:

Now consider the following form of our example query:

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch) \bowtie account) \bowtie depositor)$$

When we compute the subexpression

$$(\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch) \bowtie account)$$

we obtain a relation whose schema is

(*branch-name, branch-city, assets, account-number, balance*)

We can eliminate several attributes from the schema, by pushing projections based on equivalence rules 8.a and 8.b. The only attributes that we must retain are those that either appear in the result of the query or are needed to process subsequent operations. By eliminating unneeded attributes, we reduce the number of columns of the intermediate result. Thus, we reduce the size of the intermediate result. In our example, the only attribute we need from the join of *branch* and *account* is *account-number*. Therefore, we can modify the expression to

$$\Pi_{customer\text{-}name}\ (\ (\Pi_{account\text{-}number}\ ((\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch)) \bowtie account)) \bowtie depositor)$$

The projection $\Pi_{account\text{-}number}$ reduces the size of the intermediate join results.

Solved Examples:

Q. Show how to derive the following equivalences by a sequence of Transformations using the equivalence rules.

a. $\sigma\theta1\theta2\theta3(E) = \sigma\theta1\ (\sigma\theta2\ (\sigma\theta3\ (E)))$

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$, where θ_2 involves only attributes from E_2

Answer:

a. Using rule 1, $\sigma\theta1\theta2\theta3(E)$ becomes $\sigma\theta1\ (\sigma\theta2\theta3(E))$. On applying rule 1 again, we get $\sigma\theta1\ (\sigma\theta2\ (\sigma\theta3\ (E)))$.

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2)$ on applying rule 1 becomes $\sigma_{\theta_1}(\sigma_{\theta_2}(E_1 \bowtie_{\theta_3} E_2))$. This on applying rule 7.a becomes $\sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$.