Chapter -5
Query Processing and Optimization

Introduction

- Query Processing refers to range of activities involved in extracting data from a database.
- The basic steps involved in processing of a query are
    1. Parsing and translation
    2. Optimization
    3. Evaluation and execution

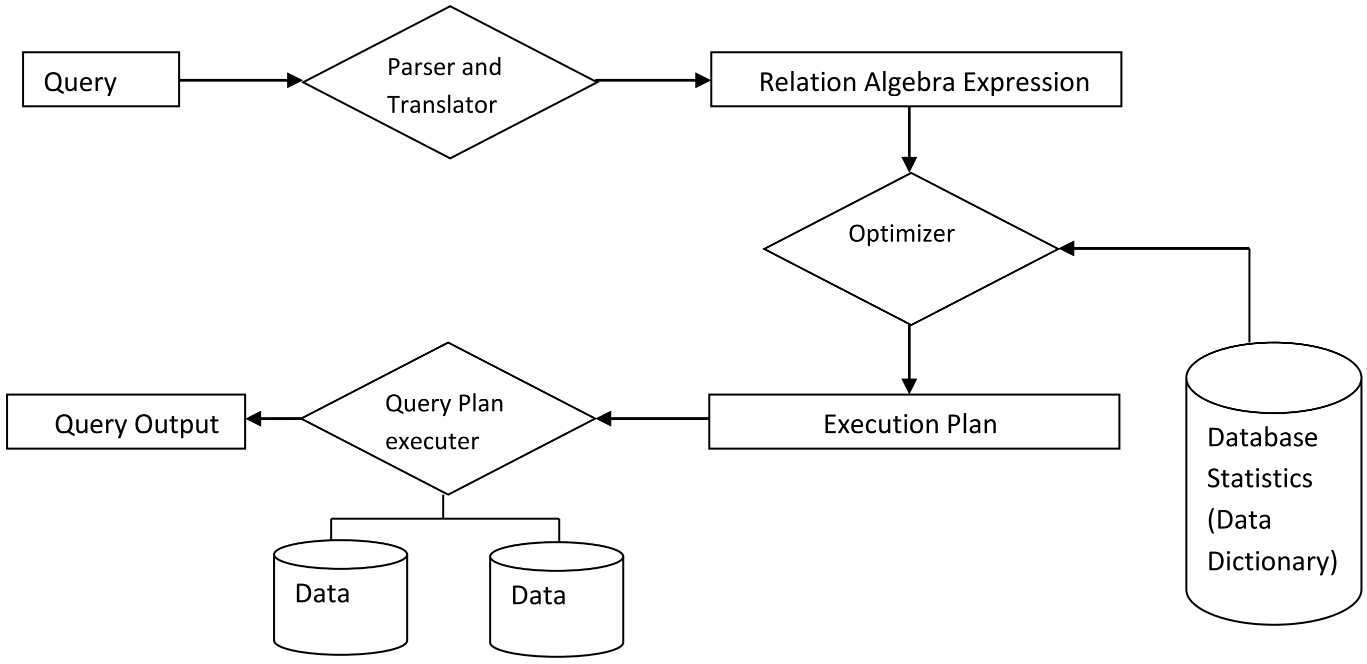


Fig: Steps in Query Processing

1. Parsing and Translation

- The first step in any query processing system is to translate a given query into its internal form.
- In generating the internal from of the query, the parser checks the syntax of the user's query, verifies that the relation is formulated according to the syntax rules of the query language.
- Then this is translated into relational algebra.

2. Optimization

- A relational algebra expression may have many equivalent expressions. For example a query

  Select balance from account where balance<2500

  This query may be translated into following equivalent relational algebraic expressoins

  $$\sigma_{balance<2500}(\prod_{balance}(\textbf{account})) \text{ is equivalent to} \prod_{balance}(\sigma_{balance<2500}(\textbf{account}))$$
- We can execute each relation algebra operation by one of several different execute algorithms. For example, to implement the preceding selection, we can search every tuple in account to find tuples with balance less than 25000. If a B+-tree index is available on the attribute balance; we can use the index instead to locate the tuples.
- The process of choosing a suitable one with lowest cost is known as query optimization.
- Cost is estimated using the statistical information from database catalog. The different statistical information is number of tuples in each relation, size of tuples etc. So among all equivalent expressions, choose the one with the cheapest possible evaluation plan (one of the possible way of executing a query).

3. Evaluation and execution

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

Query Cost Estimation

- Each query is translated into a number of semantically equivalent plans.
- So there are several alternatives, now the question is which one is the most efficient evaluation plan to be selected for execution.
- To get the answer, the cost for all alternatives must be estimated and the plan with lowest cost is selected.
- Since a database resides on disk, often the cost of reading and writing to disk dominates the cost of processing query.
- We can choose a strategy based on reliable information, database systems may store statistics (metadata) for each relation R. These statistics includes number of tuples in a relation, size of tuples in a relation etc.
- Cost is generally measured as total elapsed time for answering a query.
- Many factors contribute to time cost. Some of them are disk accesses, CPU, network communication etc.
- There **are basically two techniques for evaluation of expressions**
  1. Materialization
  2. Pipelining

1. Materialization

- Materialization is the way how to evaluate an expression containing multiple operations by simply evaluating one operation at a time, in an appropriate order.
- The result of each evaluation is materialized in a temporary relation for subsequent use.
- A disadvantage to this approach is the need to construct the temporary relations, which must be written to disk

Example
Consider the following expression

$$\Pi_{name}(\sigma_{building = \text{“Watson”}}(department) \bowtie instructor)$$

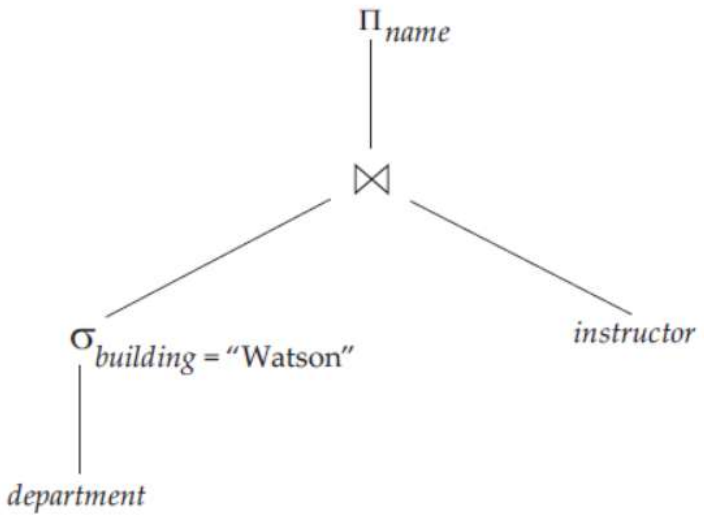


Fig: Pictorial representation of above expression

- In this approach, we start from the lowest-level operations in the expression (at the bottom of the tree).
- There is only one such operation: the selection operation on department. The inputs to the lowest-level operations are relations in the database. We execute these operations and store the results in temporary relations.
- We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database. Here, the inputs to the join are the instructor relation and the temporary relation created by the selection on department.
- The join can now be evaluated, creating another temporary relation.
- By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression
- Evaluation as just described is called **materialized evaluation,** since the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.
- The cost of a materialized evaluation is not simply the sum of the costs of the operations involved. To compute the cost of evaluating an expression as done here, we have to add the costs of all the operations, as well as the cost of writing the intermediate results to disk.

2. **Pipelining**

- We can improve query-evaluation efficiency by reducing the number of temporary files that are produced.
- We achieve this reduction by combining several relational operations into a pipeline of operations, in which the results of one operation are passed along to the next operation in the pipeline. This type of evaluation is called pipelined evaluation.

  Example,

  - Consider the expression $(\pi_{\mathbf{a1},\ \mathbf{a2}} (r \bowtie s))$. If materialization were applied, evaluation would involve creating a temporary relation to hold the result of the join, and then reading back in the result to perform the projection.
  - So, in this approach, these operations can be combined: When the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. By combining the join and the projection, we avoid creating the intermediate result,and instead create the final result directly.

Advantage of pipelining

- It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation.
- It can start generating query results quickly.

<u>Equivalence / Transformation Rules</u>

- Two algebraic expressions are said to be equivalent if they produce same result.
- By using the equivalence rule which is concerned with basic relational algebra operator, we can formulate any equivalent expressions for a single query.
- If R, S and T are relations and C1, C2……Cn are conditions then equivalent rules are

1. Commutativity of binary operators

$R \cup S \equiv S \cup R$ $R \cap S \equiv S \cap R$ $R \bowtie S \equiv S \bowtie R$ $R \times S \equiv S \times R$

2. Associativity of binary operator

$(R \cup S) \cup T \equiv R \cup (S \cup T)$ $(R \cap S) \cap T \equiv R \cap (S \cap T)$ $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ $R \times (S \times T) \equiv (R \times S) \times T$

3. Commutating projection with binary operator

$\prod_C(R \times S) \equiv \prod_A(R) \times \prod_B(S)$ where $C = A \cup B$ such that attribute A is in relation R and attribute B is in relation S.
And similar for join operator also.

4. Commutating selection with binary operator

a. $\sigma_C(R \times S) \equiv \sigma_C(R) \times S$, if the attribute involved in condition is from relation R

b. $\sigma_C(R \times S) \equiv R \times \sigma_C(S)$, if the attribute involved in condition is from relation S

c. $\sigma_C(R \times S) \equiv \sigma_A(R) \times \sigma_B(S)$, where $C = A \wedge B$ such that condition A has attribute from R and condition B has attribute from S. notes: *applying more restrictive selection first*

And similar for join operator also.

5. Commutating selection and projection

$\prod_X(\sigma_C(R)) \equiv \sigma_C(\prod_X(R))$ $\sigma_C(\prod_X(R)) \equiv \prod_X(\sigma_C(R))$

6. Idempotence of unary operator

a. Combine Cascade Selection

$\sigma_{C1}(\sigma_{C2}(R)) \equiv \sigma_{C1 \wedge \sigma\ C2}(R)$

b. Combine Cascade Projection

$\prod_X(\prod_Y(R)) \equiv \prod_X(R)$ if X is subset of Y.

Example: Suppose we have the relational algebra expression as below.

$\prod_{\text{customer-name}}(\sigma_{\text{branch-city = 'ktm'} \wedge \text{balance} > 1000}(\text{branch} \bowtie \text{account} \bowtie \text{depositer}))$

Using rule no 4a we can have equivalent expression as below

$\prod_{\text{customer-name}}((\sigma_{\text{branch-city = 'ktm'} \wedge \text{balance} > 1000}(\text{branch} \bowtie \text{account}) \bowtie \text{depositer}))$

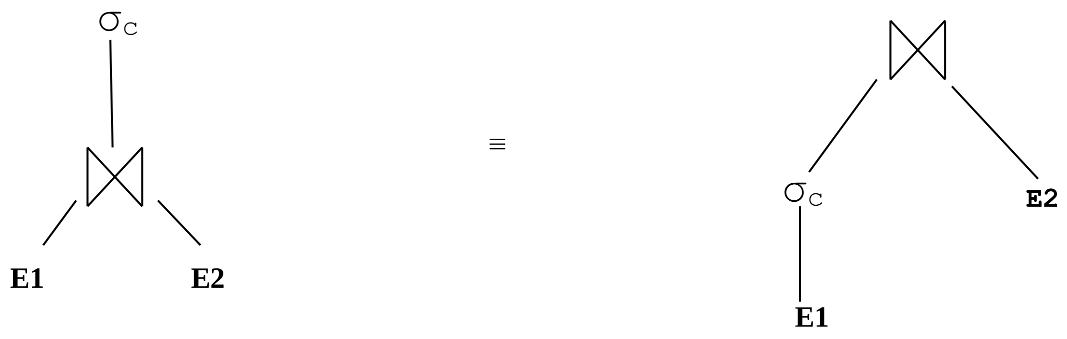
Using rule no 4c, we can have another equivalent expression as below

$\prod_{\text{customer-name}}((\sigma_{\text{branch-city = 'ktm'}}(\text{Branch}) \bowtie \sigma_{\text{balance} > 1000}(\text{account}) \bowtie \text{depositer}))$

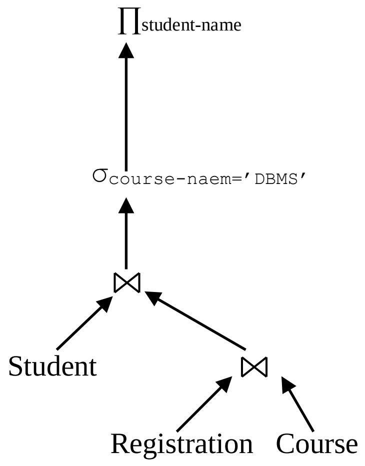
Operator Tree

- The relational algebra query can be represented graphically for simplicity by an operator tree.
- An operator tree is a tree in which leaf node is a relation stored in the database and a non-leaf node is a intermediate relation produce by a relational algebra operator.
- The sequence of operations is directed from leaves to the root, which represents the answer to the query.
-

Note, Let E1 and E2 be two relation then using rule no 4
(if c has attribute only from E1)



Example:
Suppose we have a relational algebra expression as below.
1. $\prod_{\text{student-name}}(\sigma_{\text{course-naem='DBMS'}}(\text{Student} \bowtie \text{Registration} \bowtie \text{Course}))$
The initial operator tree for the above relational expression is as below.



$\prod_{\text{student-name}}$
$\sigma_{\text{course-naem='DBMS'}}$
Student
Registration
Course

Query Optimization

- It is the process of selecting the most efficient query execution plan among the many strategies possible for processing a query.
- 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
- The query optimizer is very important component of a database system because the efficiency of the system depends on the performance of the optimizer.
- The selected plan minimizes the cost function.

Steps of optimization

1. Create an initial operator (expression) tree.
2. Move select operation down the tree for the easiest possible execution.
3. Applying more restrictive select operation first.
4. Replace Cartesian product by join.
5. Creating new projection whenever needed.
6. Adjusting rest of the tree accordingly.

Example of Optimization

Suppose we are given the following table definitions with the certain records in each table.

PROJ (<u>PNO</u>, PNAME, BUDGET)
EMP(<u>ENO</u>, ENAME, TITLE)
ASG(<u>ENO</u>, <u>PNO</u>, DUR)

Consider the following query, for the query **“Find the names of employees other than Ram Thapa who worked on CAD/CAM project for either 1 or 2 years”**

select ENAME from EMP join ASG on EMP.ENO=ASG.ENO join PROJ on ASG.PNO=PROJ.PNO where ENAME!= ‘Ram Thapa’ and PNAME=’CAD/CAM’ and (DUR = 1 or DUR =2)

The relational-algebraic expression, for the above query is

$$\prod_{ENAME}(\sigma_{ENAME \neq 'Ram\ Thapa' \wedge PNAM = 'CAD/CAM' \wedge (DUR = 1 \vee DUR = 2)}(PROJ \bowtie (EMP \bowtie ASG)))$$

The above relational algebraic expression can be graphically shown by an initial operator tree as below

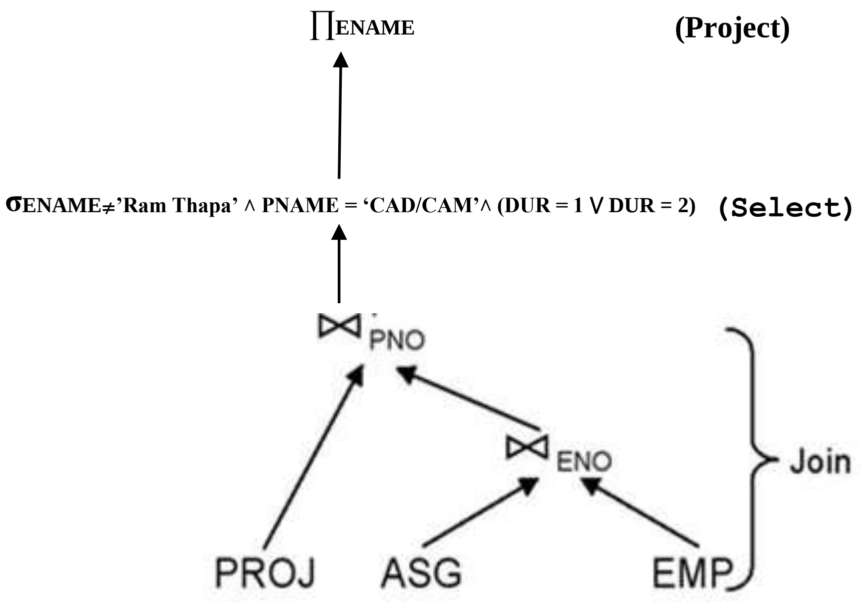


Fig: Initial Operator Tree

The above expression constructs a large intermediate relation, so for optimizing the query by applying several equivalence rules for reducing the number of tuples to be accessed in relation PROJ, ASG and EMP, we can get the following transformed expression tree.

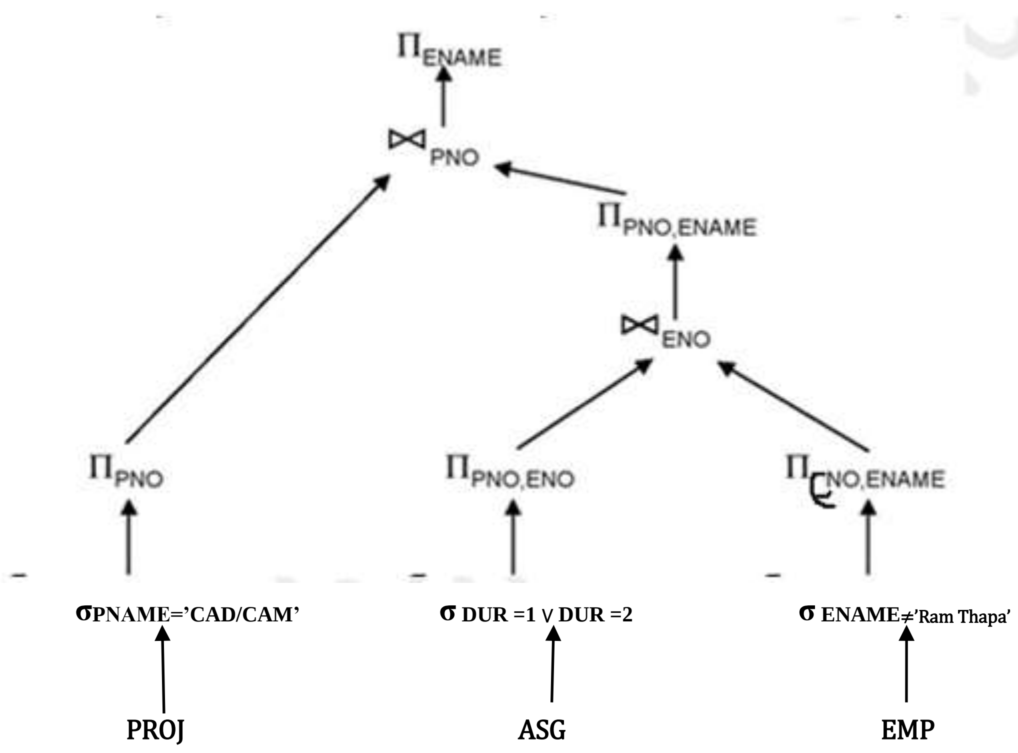


Fig: Transformed Operator Tree

So the transformed relational algebric expression can be written as

$$\prod_{ENAME}((\prod_{PNO}(\sigma_{PNAME='CAD/CAM'}(PROJ)) \bowtie (\prod_{PNO,ENAME}(\prod_{ENO,PNO}(\sigma_{DUR=1 \vee DUR=2}(ASG)) \bowtie \prod_{ENO,ENAME}(\sigma_{ENAME \neq 'Ram\ Thapa'}(EMP)))))$$

This is equivalent to our original algebra expression, but which generates smaller intermediate relations. So this is the job of the query optimizer to come up with a query-evaluation plan that computes the same result as the given expression, and is the **least-costly way of generating the result.**

<u>Cost Based Optimizer vs Heuristics Optimizer</u>

- A cost-based optimizer explores the space of all query-evaluation plans that are equivalent to the given query, and chooses the one with the least estimated cost.
- We have seen how equivalence rules can be used to generate equivalent plans. However, cost-based optimization with arbitrary equivalence rules is fairly complicated.
- Exploring the space of all possible plans may be too expensive for complex queries.
- A drawback of cost-based optimization is the cost of optimization itself. So, most optimizers include heuristics to reduce the cost of query optimization, at the potential risk of not finding the optimal plan.
- An example of a heuristic rule is the following rule for transforming relational algebra queries:

  Perform selection operations as early as possible.
- A heuristic optimizer would use this rule without finding out whether the cost is reduced by this transformation
- So, this reduces the size of resulting relations and ultimately response time. Thus, whenever we need to generate a temporary relation, it is advantageous to apply immediately any selection that are possible.
- **We say that the preceding rule is a heuristic because it usually, but not always, helps to reduce the cost**

  For example:
- For an example of where it can result in an increase in cost,
- consider an expression **$\sigma_A(B \bowtie C)$,** where the condition A refers only to the attributes of C. Now, if here is a case that relation C is extremely large and relation B is extremely small so in this case it is bad idea to perform selection first. This is because performing selection early would require doing a long scan of all tuples in C. So in this case, it is better to perform join first and then reject tuples that fail in selection.

-

Assignment.

1. **Explain the techniques for executing pipeline. (Demand driven and Producer driven pipeline)**