CHAPTER - 7
Query Processing

7.1 Introduction

In SQL, queries are expressed in high level declarative form. So the query has to be processed and optimized so that query of internal form gets a suitable execution strategy for processing. This optimization helps getting the result in a lesser time.

Query processing (or *Query Interpretation*) is a set of activities to obtain the desired information from a database system in a predictable and reliable fashion. These activities are: *Parsing and translation*, *Optimization*, and *Evaluation*.

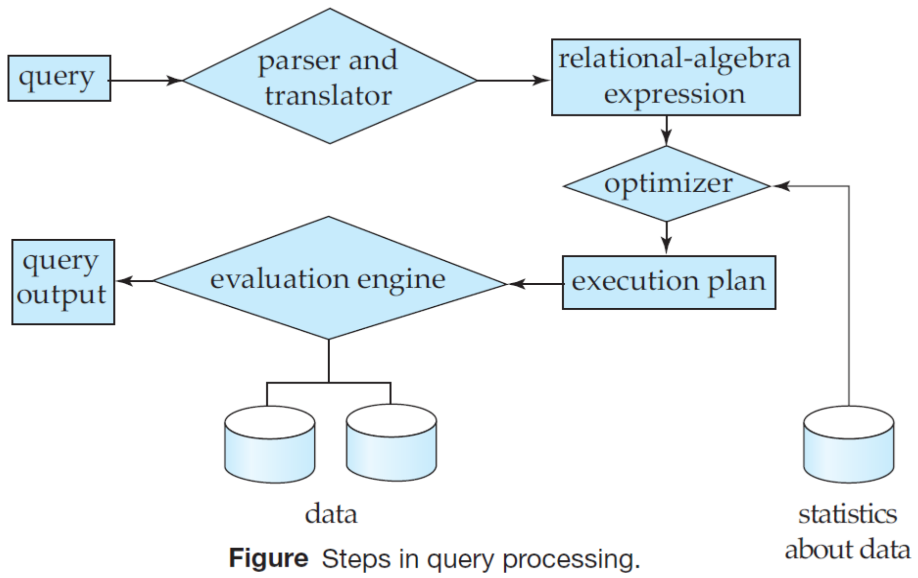

Figure Steps in query processing.

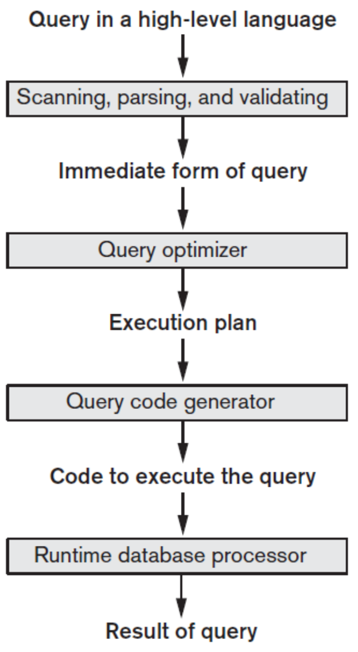
Query in a high-level language
Scanning, parsing, and validating
Immediate form of query
Query optimizer
Execution plan
Query code generator
Code to execute the query
Runtime database processor
Result of query

Figure high-level query processing steps

Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use, but is ill suited to be the system's internal representation of a query. A more useful internal representation is one based on the extended relational algebra.

Thus, 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. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view. Most compiler texts cover parsing in detail.

Given a query, there are generally a variety of methods for computing the answer. For example, we have seen that, in SQL, a query could be expressed in several different ways. Each SQL query can itself be translated into a relational algebra expression in one of several ways. Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions. As an illustration, consider the query:

SQL Statements	Relational-algebra expressions
select *salary* **from** *instructor* **where** *salary* < 75000;	$\sigma_{salary} < 75000\ (\Pi_{salary}\ (instructor))$ **OR** $\Pi_{salary}\ (\sigma_{salary} < 75000\ (instructor))$

Further, we can execute each relational-algebra operation by one of several different algorithms. For example, to implement the preceding selection, we can search every tuple in *instructor* to find tuples with salary less than 75000. If a B+-tree index is available on the attribute *salary,* we can use the index instead to locate the tuples.

To specify fully how to evaluate a query, we need not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to evaluate each operation. Annotations may state the algorithm to be used for a specific operation, or the particular index or indices to use. 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**. Figure illustrates an evaluation plan for our example query, in which a particular index (denoted in the figure as "index 1") is specified for the selection operation. The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.

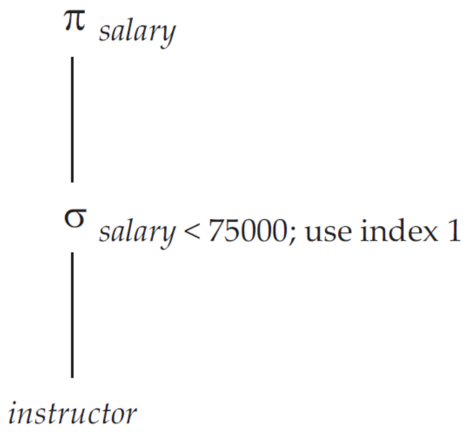

Figure A query-evaluation plan.

The different evaluation plans for a given query can have different costs. We do not expect users to write their queries in a way that suggests the most efficient evaluation plan. Rather, it is the responsibility of the system to construct a query evaluation plan that minimizes the cost of query evaluation; this task is called *query optimization*. Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output.

7.2 Equivalence of Expressions

It is always better to find out a query-processing strategy to find a relational algebra expression that is equivalent to the given query. It gives efficiency in execution.

An SQL query is first translated into an equivalent extended relational algebra expression–represented as a query tree data structure–that is then optimized. This can be done by first decomposing the SQL queries into query blocks. This basic unit can be translated into an extended relational algebra expression and then optimized. In fact Nested queries are not query blocks, but are identified as separate query blocks.

A **query block** contains a **single** SELECT-FROM-WHERE expression (may contain GROUP BY and HAVING)

Let us look the SQL query in the example of a University database on the FACULTY relation.

```
FACULTY (FID: string, FNAME: string, LNAME:string, DEPT:
string, SALARY: real, ENO: int, DNO: int, SEX:string)
```

FACULTY	FID	FNAME	LNAME	DEPT	SALARY	DNO	ENO	SEX
	46	Mitra	Guru	IT	$88,000	2	76	M
	47	Lara	Maya	MBA	$88,999	3	85	F

Consider the following SQL query on the Faculty relation

```
SELECT LNAME, FNAME
FROM FACULTY
WHERE SALARY > (SELECT MAX (SALARY) FROM FACULTY WHERE DNO=2);
```

This query includes a nested subquery and hence would be decomposed into two blocks.

Query	SQL Statements	Relational algebra expression
Inner	`SELECT MAX (SALARY) FROM FACULTY WHERE DNO=2`	$\pi_{<MAX\ SALARY>}(\sigma_{<DNO=2>}(FACULTY))$
Outer	`SELECT LNAME, FNAME FROM FACULTY WHERE SALARY > c`	$\pi_{<FNAME,LNAME>}(\sigma_{<SALARY>c>}(FACULTY))$
Where ***c*** represents the result returned from the inner query block.		

After the SQL query is decomposed then the query optimizer chooses an execution plan for each block. It should be noted that in the above example, the inner block needs to be evaluated only once to produce the maximum salary, which is then used–as the constant **c**–by the outer block. This is known as *uncorrelated nested query.* It is difficult to optimize the more complex *correlated nested queries,* where a tuple variable from the outer block appears in the WHERE-clause of the inner block.

7.3 Query Optimization

Query optimization 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. Another aspect is selecting a detailed strategy for processing the query, such as choosing the algorithm to use for executing an operation, choosing the specific indices to use, and so on. The *heuristic* (by graph and tree) and *semantic* are two popular query optimization techniques.

The difference in cost (in terms of evaluation time) between a good strategy and a bad strategy is often substantial, and may be several orders of magnitude. Hence, it is worthwhile for the system to spend a substantial amount of time on the selection of a good strategy for processing a query, even if the query is executed only once.

Given a relational-algebra expression, it 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 (or, at least, is not much costlier than the least-costly way).

To find the least-costly query-evaluation plan, the optimizer needs to generate alternative plans that produce the same result as the given expression, and to choose the least-costly one. Generation of query-evaluation plans involves three steps:

1. Generating expressions that are logically equivalent to the given expression,
2. Annotating the resultant expressions in alternative ways to generate alternative query-evaluation plans, and
3. Estimating the cost of each evaluation plan, and choosing the one who's estimated cost is the least.

Consider the following relational-algebra expression, for the query "Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach."

$$\Pi_{name,title}\ (\sigma_{dept\_name=\text{“Music”}}\ (instructor\ \bowtie\ (teaches\ \bowtie\ \Pi_{course\_id,title}(course))))$$

Note that the projection of *course* on (*course_id*, *title*) is required since *course* shares an attribute *dept_name* with *instructor*; if we did not remove this attribute using the projection, the above expression using natural joins would return only courses from the Music department, even if some Music department instructors taught courses in other departments. The above expression constructs a large intermediate relation,

$$instructor\ \bowtie\ (teaches\ \bowtie\ \Pi_{course\_id,title}(course))$$

However, we are interested in only a few tuples of this relation (those pertaining to instructors in the Music department), and in only two of the ten attributes of this relation. Since we are concerned with only those tuples in the *instructor* relation that pertain to the Music department, we do not need to consider those tuples that do not have *dept_name* = "Music". By reducing the number of tuples of the *instructor* relation that we need to access, we reduce the size of the intermediate result. Our query is now represented by the relational-algebra expression:

$$\Pi_{name,title}\ ((\sigma_{dept\_name=\text{“Music”}}\ (instructor))\ \bowtie\ (teaches\ \bowtie\ \Pi_{course\_id,title}(course)))$$

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

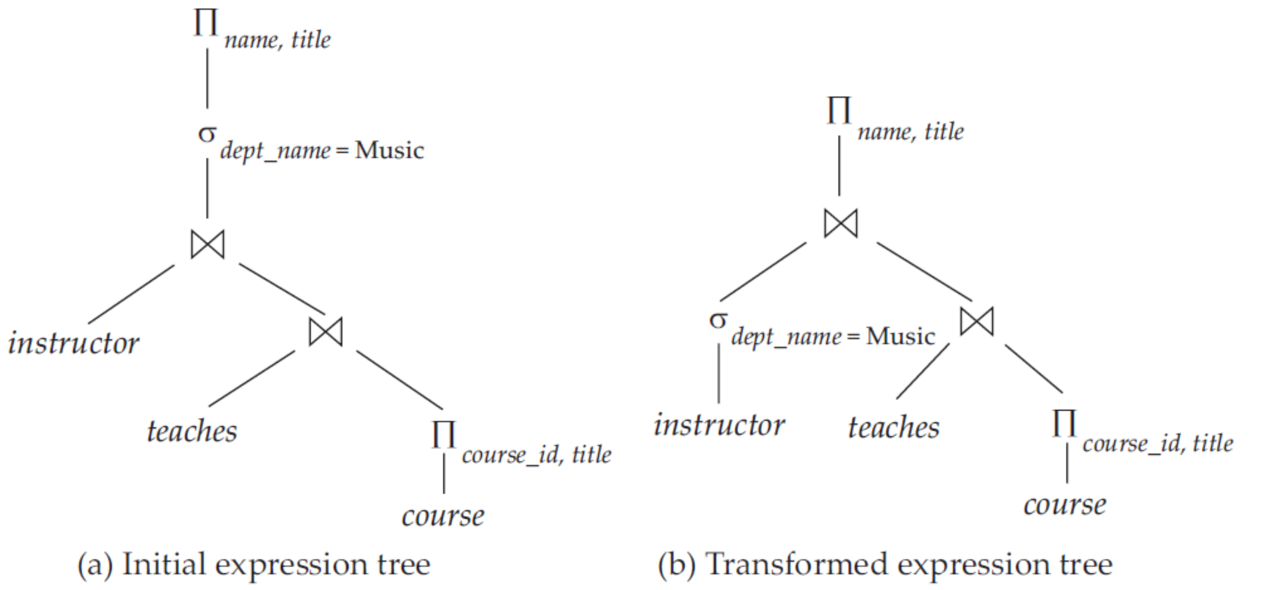

Figure Equivalent expressions.

An evaluation plan defines exactly what algorithm should be used for each operation, and how the execution of the operations should be coordinated. Following figure illustrates one possible evaluation plan for the expression from above second figure.

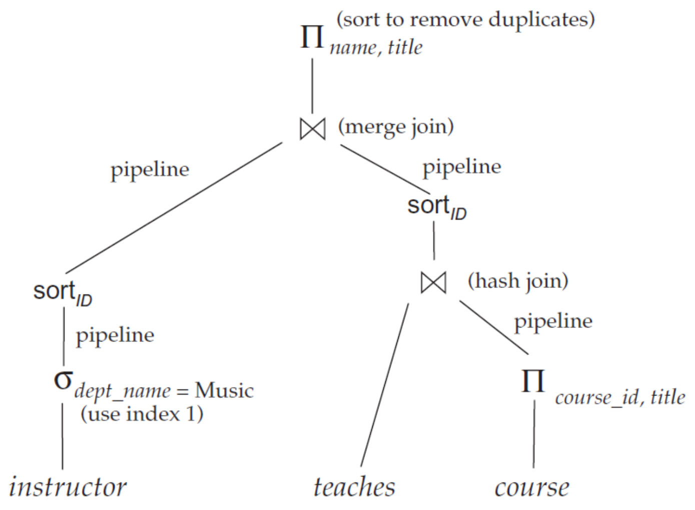

Figure An evaluation plan.

7.4 Cost Estimates in Query Optimization

The strategy of query optimizer depends on estimation and comparison of the cost of executing different strategies for the query. Query Optimizer should be able to find out the lowest cost estimates. After considering all the strategies, one has to choose the query execution plans with lowest cost and least execution time. So accurate cost estimates are required which can be compared fairly and realistically.

There are different components involved in the cost of query execution and different types of information needed in cost functions. This information is kept in DBMS catalog.

Measure of query cost

The cost of executing a query includes the following components:

A. **Access cost to secondary storage:** This is the cost measured by the search performed while reading, and writing data blocks that reside on secondary storage, mainly on disk. The cost of searching for records in a file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, access cost is affected by the way file blocks are allocated contiguously on the same disk cylinder or scattered on the disk. This cost is usually more important in the case of large database since disk accesses are slow compared to in-memory operations.
B. **Storage cost:** This is the cost of storing any intermediate files that are generated by an execution strategy for the query.
C. **Computation cost:** This is the cost measured by the performance of in-memory operations on the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join, and performing computations on field values. This cost is important when database is small where almost all data reside in the memory.
D. **Memory usage cost:** This is the cost pertaining to the number of memory buffers needed during query execution.
E. **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 the distributed system, this cost should be minimized.

There can be other factors also that may be the cost components in a cost function which may not be so much of importance. Therefore, some cost functions consider only disk access cost as the reasonable measure of the cost of a query-evaluation plan.