DBMS Unit-05

By:
Mani Butwall,
Asst. Prof. (CSE)

Introduction

- **Query:** A query is a request for information from a database.
- **Query Plans:** A query plan (or query execution plan) is an ordered set of steps used to access data in a SQL relational database management system.
- **Query Optimization:** A single query can be executed through different algorithms or re-written in different forms and structures. Hence, the question of query optimization comes into the picture – Which of these forms or pathways is the most optimal? The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.

Importance

- The goal of query optimization is to reduce the system resources required to fulfill a query, and ultimately provide the user with the correct result set faster. First, it provides the user with faster results, which makes the application seem faster to the user.
- Secondly, it allows the system to service more queries in the same amount of time, because each request takes less time than unoptimized queries.
- Thirdly, query optimization ultimately reduces the amount of wear on the hardware (e.g. disk drives), and allows the server to run more efficiently (e.g. lower power consumption, less memory usage).

- **There are broadly two ways a query can be optimized:**
- Analyze and transform equivalent relational expressions: Try to minimize the tuple and column counts of the intermediate and final query processes.
- Using different algorithms for each operation: These underlying algorithms determine how tuples are accessed from the data structures they are stored in, indexing, hashing, data retrieval and hence influence the number of disk and block accesses (discussed in query processing).

Query Processing in DBMS

- Query Processing is the activity performed in extracting data from the database. In query processing, it takes various steps for fetching the data from the database. The steps involved are:

1. **Parsing and translation**
2. **Optimization**
3. **Evaluation**

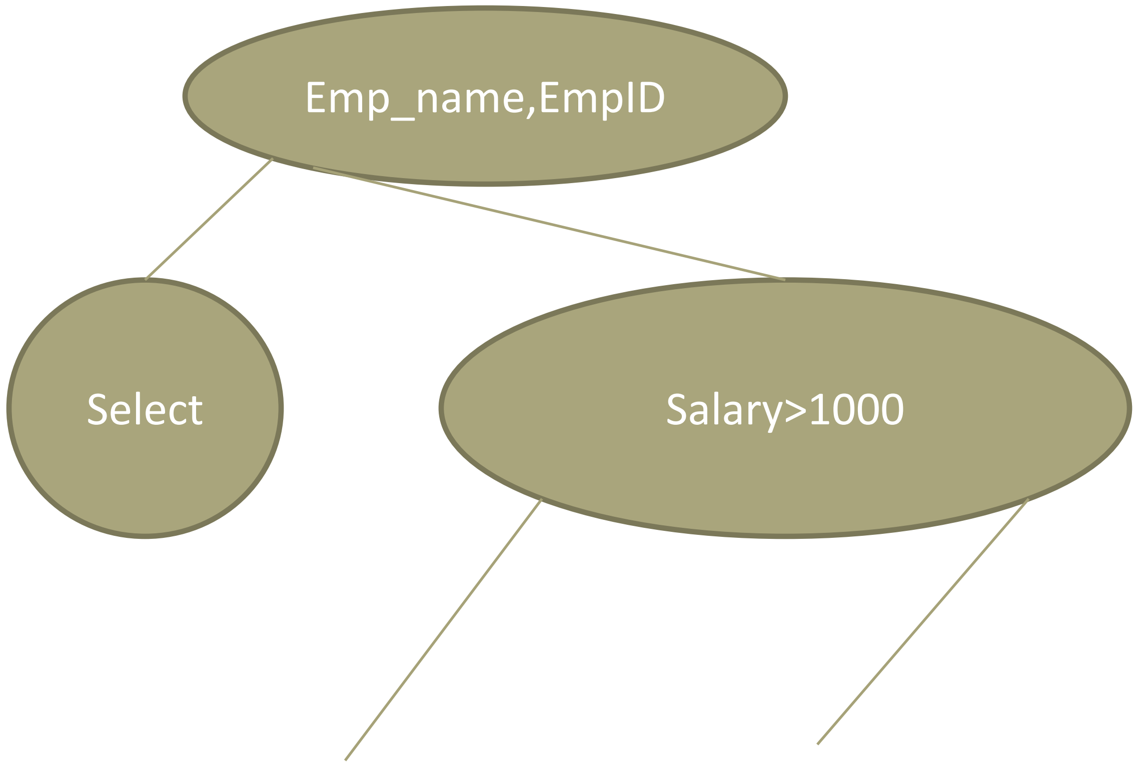
- The query processing works in the following way:

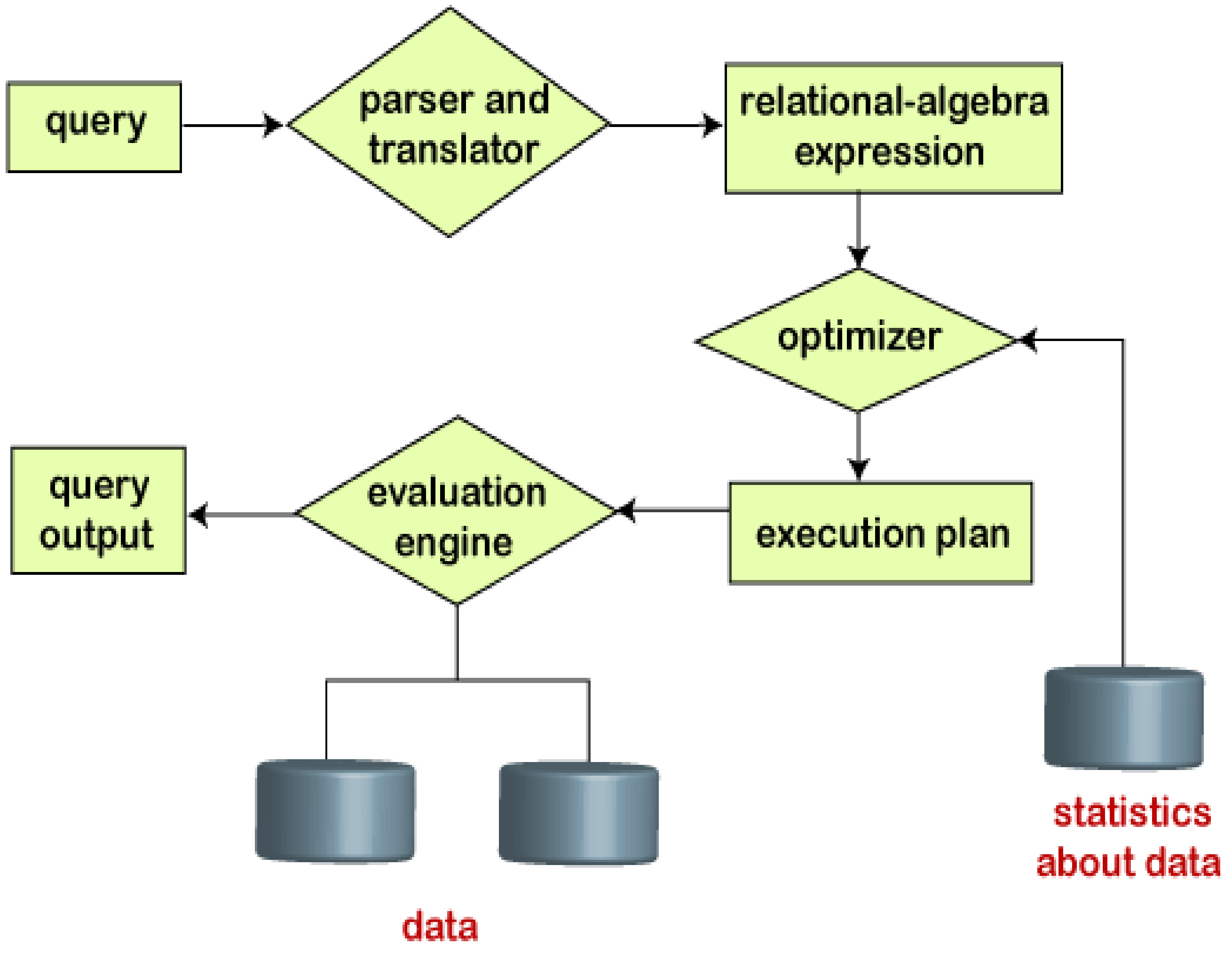
Parsing and Translation

- As query processing includes certain activities for data retrieval. Initially, the given user queries get translated in high-level database languages such as SQL.
- It gets translated into expressions that can be further used at the physical level of the file system.
- After this, the actual evaluation of the queries and a variety of query -optimizing transformations and takes place. Thus before processing a query, a computer system needs to translate the query into a human-readable and understandable language.
- Consequently, SQL or Structured Query Language is the best suitable choice for humans. But, it is not perfectly suitable for the internal representation of the query to the system.
- Relational algebra is well suited for the internal representation of a query.

- The translation process in query processing is similar to the parser of a query.
- When a user executes any query, for generating the internal form of the query, the parser in the system checks the syntax of the query, verifies the name of the relation in the database, the tuple, and finally the required attribute value.
- The parser creates a tree of the query, known as 'parse-tree.' Further, translate it into the form of relational algebra. With this, it evenly replaces all the use of the views when used in the query.

A+B*C





Steps in query processing

- Suppose a user executes a query. As we have learned that there are various methods of extracting the data from the database. In SQL, a user wants to fetch the records of the employees whose salary is greater than or equal to 10000. For doing this, the following query is undertaken:
- **select emp_name from Employee where salary>10000;**
- Thus, to make the system understand the user query, it needs to be translated in the form of relational algebra. We can bring this query in the relational algebra form as:

1. $\sigma_{salary>10000} (\pi_{salary} (Employee))$
2. $\pi_{salary} (\sigma_{salary>10000} (Employee))$

- After translating the given query, we can execute each relational algebra operation by using different algorithms. So, in this way, a query processing begins its working.

Evaluation

- For this, with addition to the relational algebra translation, it is required to annotate the translated relational algebra expression with the instructions used for specifying and evaluating each operation. Thus, after translating the user query, the system executes a query evaluation plan.
- **Query Evaluation Plan**

1. In order to fully evaluate a query, the system needs to construct a query evaluation plan.
2. The annotations in the evaluation plan may refer to the algorithms to be used for the particular index or the specific operations.
3. Such relational algebra with annotations is referred to as **Evaluation Primitives**. The evaluation primitives carry the instructions needed for the evaluation of the operation.
4. Thus, a query evaluation plan defines a sequence of primitive operations used for evaluating a query. The query evaluation plan is also referred to as **the query execution plan**.

- A **query execution engine** is responsible for generating the output of the given query. It takes the query execution plan, executes it, and finally makes the output for the user query.

Optimization

- The cost of the query evaluation can vary for different types of queries. Although the system is responsible for constructing the evaluation plan, the user does need not to write their query efficiently.
- Usually, a database system generates an efficient query evaluation plan, which minimizes its cost. This type of task performed by the database system and is known as Query Optimization.
- For optimizing a query, the query optimizer should have an estimated cost analysis of each operation. It is because the overall operation cost depends on the memory allocations to several operations, execution costs, and so on.
- Finally, after selecting an evaluation plan, the system evaluates the query and produces the output of the query.

Approaches to Query Optimization

- Among the approaches for query optimization, exhaustive search and heuristics-based algorithms are mostly used.
- **Exhaustive Search Optimization**
- In these techniques, for a query, all possible query plans are initially generated and then the best plan is selected. Though these techniques provide the best solution, it has an exponential time and space complexity owing to the large solution space. For example, dynamic programming technique.
- **Heuristic Based Optimization**
- Heuristic based optimization uses rule-based optimization approaches for query optimization. These algorithms have polynomial time and space complexity, which is lower than the exponential complexity of exhaustive search-based algorithms. However, these algorithms do not necessarily produce the best query plan.

- Some of the common heuristic rules are −
- Perform select and project operations before join operations. This is done by moving the select and project operations down the query tree. This reduces the number of tuples available for join.
- Perform the most restrictive select/project operations at first before the other operations.
- Avoid cross-product operation since they result in very large-sized intermediate tables.

SQL Tuning/SQL Optimization Techniques:

- **1)** The sql query becomes faster if you use the actual columns names in SELECT statement instead of than '*'.
- **For Example:** Write the query as
- SELECT id, first_name, last_name, age, subject FROM student_details;
- **Instead of:**
- SELECT * FROM student_details;

- **2)** HAVING clause is used to filter the rows after all the rows are selected. It is just like a filter. Do not use HAVING clause for any other purposes.
  For Example: Write the query as

-
  ```
  SELECT subject, count(subject)
  FROM student_details
  WHERE subject != 'Science'
  AND subject != 'Maths'
  GROUP BY subject;
  ```
- **Instead of:**

-
  ```
  SELECT subject, count(subject)
  FROM student_details
  GROUP BY subject
  HAVING subject!= 'Vancouver' AND subject!= 'Toronto';
  ```

- **3)** Sometimes you may have more than one subqueries in your main query. Try to minimize the number of subquery block in your query. **For Example:** Write the query as

-
  ```
  SELECT name
  FROM employee
  WHERE (salary, age ) = (SELECT MAX (salary), MAX (age)
  FROM employee_details)
  AND dept = 'Electronics';
  ```
- **Instead of:**

-
  ```
  SELECT name
  FROM employee
  WHERE salary = (SELECT MAX(salary) FROM employee_details)
  AND age = (SELECT MAX(age) FROM employee_details)
  AND emp_dept = 'Electronics';
  ```

- **4)** Use operator EXISTS, IN and table joins appropriately in your query.
  a) Usually IN has the slowest performance.
  b) IN is efficient when most of the filter criteria is in the sub-query.
  c) EXISTS is efficient when most of the filter criteria is in the main query.
- **For Example:** Write the query as
- Select * from product p
  where EXISTS (select * from order_items o
  where o.product_id = p.product_id)
- **Instead of:**
- Select * from product p
  where product_id IN
  (select product_id from order_items

- **5)** Use EXISTS instead of DISTINCT when using joins which involves tables having one-to-many relationship.
  For Example: Write the query as
- SELECT d.dept_id, d.dept
  FROM dept d
  WHERE EXISTS (SELECT 'X' FROM employee e WHERE e.dept = d.dept);
- **Instead of:**
- SELECT DISTINCT d.dept_id, d.dept
  FROM dept d,employee e
  WHERE e.dept = e.dept;

- **6)** Try to use UNION ALL in place of UNION.
  For Example: Write the query as
- SELECT id, first_name
  FROM student_details_class10
  UNION ALL
  SELECT id, first_name
  FROM sports_team;
- **Instead of:**
- SELECT id, first_name, subject
  FROM student_details_class10
  UNION
  SELECT id, first_name
  FROM sports_team;

- **7)** Be careful while using conditions in WHERE clause.
  For Example: Write the query as
- SELECT id, first_name, age FROM student_details WHERE age > 10;
- **Instead of:**
- SELECT id, first_name, age FROM student_details WHERE age != 10;

- Write the query as
- SELECT id, first_name, age
  FROM student_details
  WHERE first_name LIKE 'Chan%';

- **Instead of:**
- SELECT id, first_name, age
  FROM student_details
  WHERE SUBSTR(first_name,1,3) = 'Cha';

- Write the query as
- SELECT id, first_name, age
  FROM student_details
  WHERE first_name LIKE NVL (:name, '%');

- **Instead of:**
- SELECT id, first_name, age
  FROM student_details
  WHERE first_name = NVL (:name, first_name);

- Write the query as
- SELECT product_id, product_name
  FROM product
  WHERE unit_price BETWEEN MAX(unit_price) and MIN(unit_price)
- **Instead of:**
- SELECT product_id, product_name
  FROM product
  WHERE unit_price >= MAX(unit_price)
  and unit_price <= MIN(unit_price)
- Write the query as
- SELECT id, name, salary
  FROM employee
  WHERE dept = 'Electronics'
  AND location = 'Bangalore';
- **Instead of:**
- SELECT id, name, salary
  FROM employee
  WHERE dept || location= 'ElectronicsBangalore';

- Use non-column expression on one side of the query because it will be processed earlier.
- Write the query as
- SELECT id, name, salary
  FROM employee
  WHERE salary < 25000;
- **Instead of:**
- SELECT id, name, salary
  FROM employee
  WHERE salary + 10000 < 35000;
- Write the query as
- SELECT id, first_name, age
  FROM student_details
  WHERE age > 10;
- **Instead of:**
- SELECT id, first_name, age
  FROM student_details
  WHERE age NOT = 10;

- **8)** Use DECODE to avoid the scanning of same rows or joining the same table repetitively. DECODE can also be made used in place of GROUP BY or ORDER BY clause.
  For Example: Write the query as
- SELECT id FROM employee
  WHERE name LIKE 'Ramesh%'
  and location = 'Bangalore';
- **Instead of:**
- SELECT DECODE(location,'Bangalore',id,NULL) id FROM employee
  WHERE name LIKE 'Ramesh%';

- **9)** To store large binary objects, first place them in the file system and add the file path in the database.
- **10)** To write queries which provide efficient performance follow the general SQL standard rules.
- **a)** Use single case for all SQL verbs
  b) Begin all SQL verbs on a new line
  c) Separate all words with a single space
  d) Right or left aligning verbs within the initial SQL verb