3 Chapter

QUERY PROCESSING AND OPTIMIZATION

Chapter Outline

After comprehensive study of this chapter, you will be able to:

- Concept of Query Processing
- Query Trees and Heuristics for Query Optimization
- Choice of Query Execution Plans
- Cost-Based Optimization.

OVERVIEW OF QUERY PROCESSING

Energy efficiency is an important feature in designing and executing databases. The aims of query processing are to transform a query written in a high-level language, typically SQL, into a correct and efficient execution strategy expressed in a low-level language (implementing the relational algebra), and to execute the strategy to retrieve the required data. Thus, **Query Processing** is the activities involved in parsing, validating, optimizing, and executing a query. The steps involved in processing a query processing is shown in figure 8.1 and they are:

1. Parsing and translation
2. Optimization
3. Evaluation

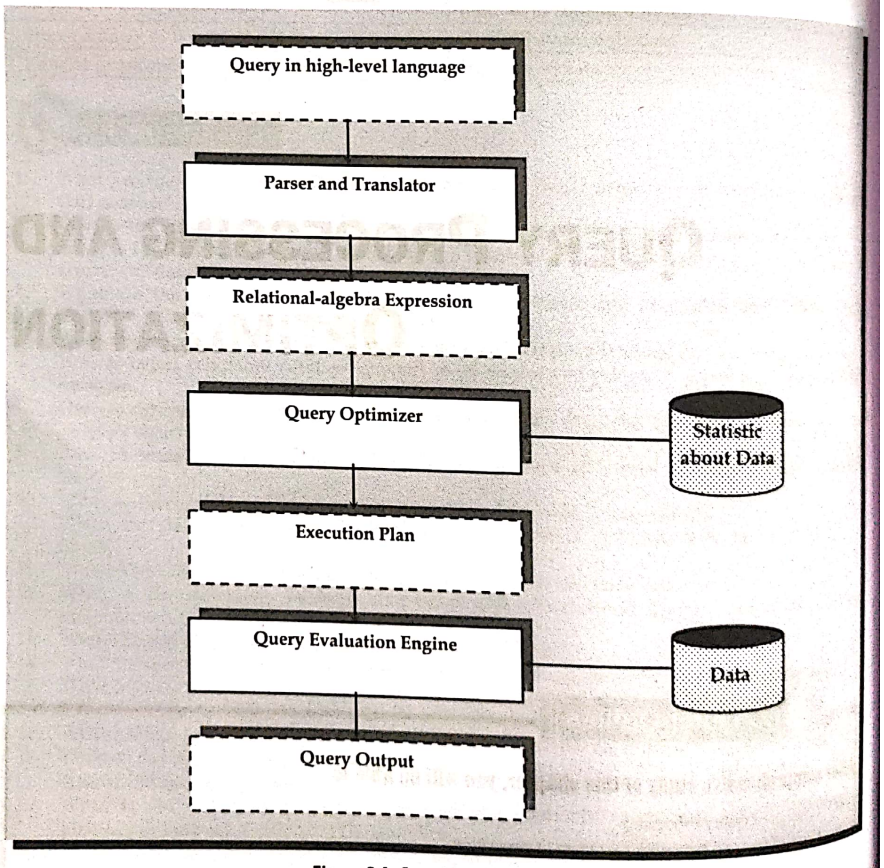

Figure 3.1: Steps in query processing

Parsing and Translating the Query

The main work of a query processor is to convert a query string into query objects i.e., converting the query submitted by the user, into a form understood by the query processing engine. It converts the search string into definite instructions. The query parser must analyze the query language i.e., recognizing and interpreting operators (AND, OR, NOT, +, - etc.), placing the operators into groups etc. The basic job of the parser is to extract the tokens (e.g., keywords, operators, operands, literal strings etc.) into their corresponding internal data elements (i.e., relational algebra operations and operands) and structures (i.e., query tree, query graph). Parser also verifies the validity and syntax of the query string.

Optimizing the Query

In this stage, Query optimizer tries to find the most efficient way of executing a given query by considering the possible plans. It maximizes the performance of a query. It portrays the query plans as a **tree**, results flowing from bottom to top. Query processor applies rules to the internal data structures of the query to transform these structures into their equivalent but more efficient representations. Rules may be based on various mathematical models and heuristics.

Evaluating the Query

The final step in processing a query is the evaluation phase. An evaluation plan tells precisely the algorithm for each operation along with the coordination among the operations. The best evaluation plan that a user generates by optimization engine is selected and then executed (There may exist various methods for executing the same query). The evaluation plan comprises of a relational algebra tree, providing information at each node (for each table) along with the implementation methods to be employed for each relational operator.

Example 3.1: Consider the following SQL query respectively:

```
SELECT Stu_name, Stu_address
FROM Student
WHERE age ≤ 25;
```

This query can be translated into either of the following relational-algebra expressions:

- $\sigma_{age \leq 25}(\Pi_{Stu\_name, Stu\_address}(Student))$
- $\Pi_{Stu\_name, Stu\_address}(\sigma_{age \leq 25}(Student))$

This can be represented as either of the following query trees:

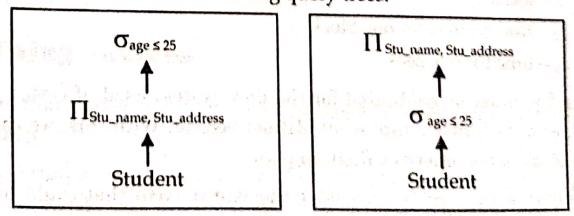

Figure 3.2: Query Tree

After parsing and translation into a relational algebra expression, the query is then transformed into a form, usually a query tree or query graph, that can be handled by the optimization engine.

The optimization engine then performs various analyses on the query data, generating a number of valid evaluation plans. From there, it determines the most appropriate evaluation plan to execute.

After the evaluation plan has been selected, it is passed into the DMBS' query-execution engine (also referred to as the runtime database processor), where the plan is executed and the results are returned.

MEASURING OF QUERY COST

Cost of query is the time taken by the query to hit the database and return the result. It involves query processing time i.e., *time taken to parse and translate the query, optimize it, evaluate, execute* and *return the result to the user* is called **cost of the query**. Executing the optimized query involves hitting the primary and secondary memory based on the file organization method. Depending on file organization and the indexes used, time taken to retrieve the data may vary. Query cost considers the number of different resources that are listed below:

- The number of disk accesses / the number of disk block transfers / the size of the table
- Time taken by CPU for executing the query

The time taken by **CPU is negligible** in most systems when compared with the number of disk accesses. If we consider the number of block transfers as the main component in calculating the cost of a query, it would include more sub-components. Those are:

- **Rotational latency:** time taken to bring and spin the required data under the read-write head of the disk.
- **Seek time:** time taken to position the read-write head over the required track or cylinder.
- **Sequential I/O:** reading data that are stored in contiguous blocks of the disk
- **Random I/O:** reading data that are stored in different blocks that are not contiguous.

For simplicity we just use the **number of block transfers** from disk and **the number of seeks** as the cost measures of a query-evaluation plan. Suppose a query need to seek S times to fetch a record and there are **b** blocks needs to be returned to the user. The disk I/O cost is calculated as below

$$\text{Query Cost} = b \times t_T + S \times t_S$$

Where,

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

The values of t_T and t_S must be calibrated for the disk system used, if t_T=0.1 ms, t_S =4 ms, the block size is 4 KB, and its transfer rate is 40 MB per second. With this, we can easily calculate the estimated cost of the given query evaluation plan.

Generally, for estimating the cost, we consider the **worst case** that could happen. The users assume that initially, the data is read from the disk only. But there must be a chance that the information is already present in the main memory. However, the users usually ignore this effect, and due to this, the actual cost of execution comes out less than the estimated value.

The **response time**, i.e., the time required to execute the plan, could be used for estimating the cost of the query evaluation plan. But due to the following reasons, it becomes difficult to calculate the response time without actually executing the query evaluation plan:

1. The response time depends on the contents of the buffer when the query begins execution; this information is not available when the query is optimized, and is hard to account for even if it were available.
2. In a system with multiple disks, the response time depends on how accesses are distributed among disks, which is hard to estimate without detailed knowledge of data layout on disk.

SELECTION OPERATION (σ)

Queries are ultimately reduced to a number of **file scan** operations on the underlying physical file structures. For each relational operation there can exist several different access paths to the particular records needed. The query execution engine can have a multitude of specialized algorithms designed to process particular relational operation and access path combinations.

Selections Using File Scans

File scans are search algorithms that locate and retrieve records that fulfill a selection condition. The Select operation must search through the data files for records meeting the selection criteria. The following are some ways of simple (one attribute) selection algorithms:

- **A1 (linear search):** Retrieve every record in the file, and test whether its attribute values satisfy the selection condition.
  - Worst Case Costs = $b_r \times t_T + t_S$. Where, b_r is the number of blocks containing records from relation r.
    If a selection is on a key attribute, can stop on finding record
  - Average Cost = $(b_r/2) \times t_T + t_S$.
    Linear search is slow, but it is general because it can be applied regardless of the ordering of the file, or the availability of indices, or the nature of the selection operation.
- **A2 (binary search):** If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search (which is more efficient than linear search) can be used.
  - Worst Case Costs = $\lceil log_2(b_r) \rceil \times (t_T + t_S)$.

Selections Using Indices

A search algorithm that makes use of an index is called an **index scan** and the index structure is called **access path.**

- **A3 (primary index, equality on key):** For an equality comparison on a key attribute with a primary index, we can use the index to retrieve a single record that satisfies the corresponding equality condition.
  - Cost = $(h_i + 1) \times (t_T + t_S)$, where h_i is the height of the index.

- **A4 (primary index, equality on non-key):** For an equality comparison on a non-key attribute with a primary index, we can use the index to retrieve multiple records (possibly spread over b successive blocks) that satisfy the corresponding equality condition.
  - Cost = $h_i \times (t_T + t_S) + t_S + t_T \times b$, where h_i is the height of the index.
- **A5 (secondary index, equality):** Selection specifying an equality condition can use a secondary index. This strategy can retrieve if the indexing field is not a key. Retrieve a single record if the search-key is a candidate key
  - Cost = $(h_i + 1) \times (t_T + t_S)$, where h_i is the height of the index.
    Retrieve multiple records if search-key is not a candidate key each of n matching records may be on a different block.
  - Cost = $(h_i + n) \times (t_T + t_S)$, where h_i is the height of the index.
    For large number of blocks n with matching records, this can be very expensive and cost even more than a linear scan!

Selections Involving Comparisons

We assume that the relation is sorted on attribute A. Consider a selection of the form $\sigma_{A \leq v}(r)$. We can implement the selection either by using linear search, binary search or by using indices in one of the following ways:

- **A6 (primary index, comparison):** A primary ordered index (for example, a primary B+-tree index) can be used when selection condition is a comparison.
  - For $\sigma_{A \geq v}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there.
  - For $\sigma_{A \leq v}(r)$ just scans relation sequentially till first tuple $> v$ without using any index.
- **A7 (secondary index, comparison):** We can use a secondary ordered index to guide retrieval for comparison conditions involving $<, \leq, \geq$, or $>$.
  - For $\sigma_{A \geq v}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq v}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$.

The secondary index provides pointers to the records, but to get the actual records we have to fetch the records by using the pointers. This step may require an I/O operation for each record fetched, since consecutive records may be on different disk blocks; as before, each I/O operation requires a disk seek and a block transfer. If the number of retrieved records is large, using the secondary index may be even more expensive than using linear search. Therefore, the secondary index should be used only if very few records are selected.

Selections of Complex Selections

So far, we have considered only simple selection conditions of the form A *op* B, where *op* is an equality or comparison operation. We now consider more complex selection predicates.

- **Conjunction:** A *conjunctive selection* is a selection of the form:

$$\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$$

- **Disjunction:** A *disjunctive selection* is a selection of the form: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$

  A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θi.
- **Negation:** The result of a selection $\sigma_{\neg\theta}(r)$ is the set of tuples of r for which the condition θ evaluates to false. In the absence of nulls, this set is simply the set of tuples in r that are not in $\sigma_{\theta}(r)$.
- **A8 (conjunctive selection using one index):** We first check if there is an access path available for an attribute in one of the simple conditions θ_i. to reduce the costs, we choose a θ_i and one of algorithms A1 through A8 for which the combination results in the least cost for $\sigma_{\theta i}(r)$. The cost of algorithm A8 is given by the cost of the chosen algorithm.
- **A9 (conjunctive selection using composite index):** An appropriate composite (multiple-key) index may be available for some conjunctive selections. If composite index exists on the combined attribute fields, then the index can be searched directly.
- **A10 (conjunctive selection by interesting of identifiers):** This algorithm requires indices with record pointers, on the fields involved in the individual conditions. The algorithm uses corresponding index for each condition, and take intersection of all the obtained sets of record pointers. Then fetch records from the file and if some conditions do not have appropriate indices, apply test in memory.
- **A11 (disjunctive selection by union of identifiers):** Indices can only be used if there is an index for all conditions; otherwise, a linear scan of the relation has to be performed any way. Uses corresponding index for each condition, and take union of all the obtained sets of record pointers. Then fetch records from file.

SORTING

Sorting in database system is important for two reasons:

1. A query may specify that the output should be sorted
2. The processing of some relational query operations can be implemented more efficiently based on sorted relations e.g., join operation.

For relations that fit in memory, techniques like quick-sort can be used and for relations that do not fit in memory an external sort-merge algorithm can be used.

External Sort-Merge Algorithm

Sorting of relations that do not fit in memory is called **external sorting**. The most commonly used technique for external sorting is the **external sort–merge** algorithm. Let M denote memory size (in pages).

1. Create sorted **runs**. Initialize i=0.
   Repeat the following **till** the end of the relation (Let the final value of i be N)
   a) Read M blocks of relation into memory
   b) Sort the in-memory blocks
   c) Write sorted data to run R_i
   d) $i = i + 1$

2. **Merge the runs (N-way merge).** We assume that $N < M$.
   a. Use N blocks of memory to buffer input runs (one block per run), and one block to buffer output.
   b. **Repeat** the following steps **until** all input buffer pages are empty:
      i. Select the first record (in sort order) among all buffer pages
      ii. Write the record to the output buffer. If the output buffer is full write it to disk.
      iii. Delete the record from its input buffer page. If the buffer page becomes empty then read the next block (if any) of the run into the buffer.

If $N \geq M$, several merge passes are required. In each pass, contiguous groups of $M - 1$ runs are merged. A pass reduces the number of runs by a factor of $M - 1$, and creates runs longer by the same factor. Repeated passes are performed till all runs have been merged into one.

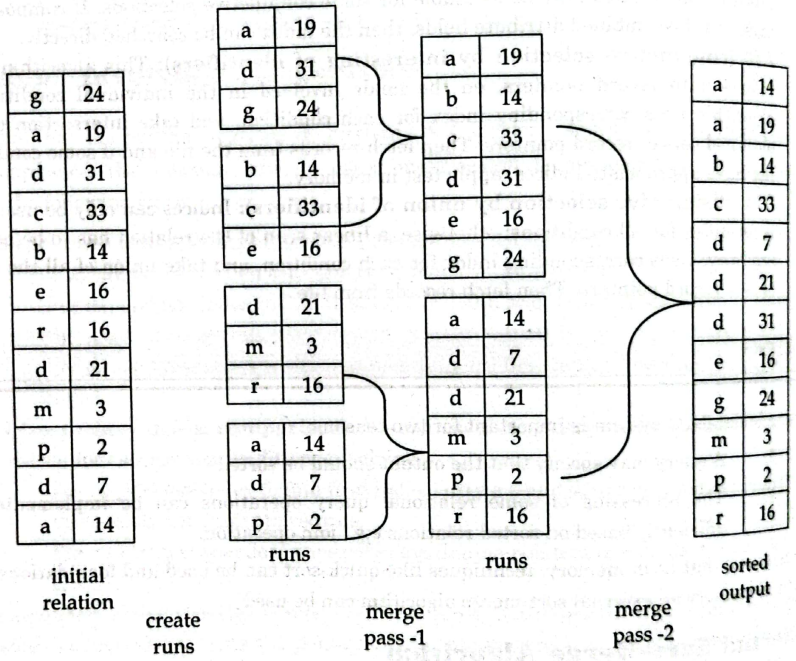

Figure 3.3: External sorting using sort-merge.

Figure 3.3 illustrates the steps of the external sort–merge for an example relation. For illustration purposes, we assume that only one tuple fits in a block ($f_r = 1$), and we assume that memory holds at most three blocks. During the merge stage, two blocks are used for input and one for output.

Cost Analysis of external Sort-Merge

Let b_r denote the number of blocks containing records of relation r.

The initial number of runs = $\lceil b_r / M \rceil$.

Since the number of runs decrease by a factor of $M - 1$ in each merge pass;

The total number of merge passes required = $\lceil \log_{M-1}(b_r / M) \rceil$.

The first stage reads every block of the relation and writes them out again, giving a total of $2b_r$ block transfers. Each of these passes reads every block of the relation once and writes it out once, with two exceptions.

- **First,** the final pass can produce the sorted output without writing its result to disk.
- **Second,** there may be runs that are not read in or written out during a pass

Total number of block transfers for external sorting of the relation = $b_r \times (2 \times \lceil \log_{M-1}(b_r / M) \rceil + 1)$.

JOINING

Like selection, the join operation (joining) can be implemented in a variety of ways. In terms of disk accesses, the joining can be very expensive, so implementing and utilizing efficient join algorithms is critical in minimizing a query's execution time. The following are 5 well-known types of join algorithms are:

- Nested-Loop Join
- Block Nested-Loop Join
- Indexed Nested-Loop Join
- Sort-Merge Join
- Hash Join

Nested-Loop Join

This algorithm consists of an inner for loop nested within an outer for loop. To illustrate this algorithm, we will use the following notations:

r, s	Relations r and s
t_r	Tuple (record) in relation r
t_s	Tuple (record) in relation s
n_r	Number of records in relation r
n_s	Number of records in relation s
b_r	Number of blocks with records in relation r
b_s	Number of blocks with records in relation s

Here is a sample pseudo-code listing for joining the two relations r and s utilizing the nested-for loop:

```
for each tuple t_r in r {
    for each tuple t_s in s do begin
        if join condition is true for (t_r, t_s)
        add tuple t_r×t_s to the result;
    }
}
```

Figure 3.4: Nested-loop join

In the algorithm, t_r and t_s are the tuples of relations r and s, respectively. The notation $t_r \times t_s$ is a tuple constructed by concatenating the attribute values of tuples t_r and t_s.

With the help of the algorithm, we understood the following points:

- The nested-loop join does not need any indexing similar to a linear file scan for accessing the data.
- Nested-loop join does not care about the given join condition. It is suitable for each given join condition.
- The nested-loop join algorithm is expensive in nature. It is because it computes and examines each pair of tuples in the given two relations.

Block Nested-Loop Join:

If the buffer is too small to hold either relation entirely in memory, we can still obtain a major saving in block accesses if we process the relations on a per-block basis, rather than on a per-tuple basis. Figure 3.5 shows block nested-loop join, which is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, ever tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples. As before, all pairs of tuples that satisfy the join condition are added to the result.

```
for each block b_r of r {
    for each block b_s of s {
        for each tuple t_r in b_r {
            for each tuple t_s in b_s {
                if join condition is true for (t_r, t_s)
                add tuple t_r×t_s to the result;
            }
        }
    }
}
```

Figure 3.5: Block nested-loop join

The primary difference in cost between the block nested-loop join and the basic nested-loop join is that, in the worst case, each block in the inner relation s is read only once for each block in the outer relation, instead of once for each tuple in the outer relation. Clearly, it is more efficient to use the smaller relation as the outer relation, in case neither of the relations fits in memory.

Index Nested-Loop Join

This algorithm is the same as the Nested-Loop Join, except an index file on the inner relation's (s) join attribute is used versus a data-file scan on s - each index lookup in the inner loop is essentially an equality selection on s utilizing one of the selection algorithms.

Sort-Merge Join

This algorithm can be used to perform natural joins and equi-joins and requires that each relation (r and s) be sorted by the common attributes between them ($R \cap S$). The details for how this algorithm works will not be presented here. However, it is notable to point out that each

record in r and s is only scanned once, thus producing a worst and best-case cost of $b_r + b_s$. Variations of the Sort-Merge Join algorithm are used, for instance, when the data files are in unsorted order, but there exist secondary indices for the two relations.

Hash Join

Like the sort-merge join, the hash join algorithm can be used to perform natural joins and equi-joins. The concept behind the Hash join algorithm is to partition the tuples of each given relation into sets. The partition is done on the basis of the same hash value on the join attributes. The hash function provides the hash value. The main goal of using the hash function in the algorithm is to reduce the number of comparisons and increase the efficiency to complete the join operation on the relations.

For example, suppose there are two tuples **a** and **b** where both of them satisfy the join condition. It means they have the same value for the join attributes. Suppose that both **a** and **b** tuples consist of a hash value as **i**. It implies that tuple a should be in $\mathbf{a_i}$, and tuple **b** should be in $\mathbf{b_i}$. Thus, only compare **a** tuples in $\mathbf{a_i}$ with **b** tuples of $\mathbf{b_i}$. There is no need to compare the **b** tuples in any other partition. Therefore, in this way, the hash join operation works.

EVALUATION OF EXPRESSION

We have studied how individual relational operations are carried out. The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order. Now we consider how to evaluate an expression containing multiple operations. There are two approaches how a query execution tree can be evaluated:

- **Materialization:** Compute the result of an evaluation primitive and **materialize** (store) the new relation on the disk.
- **Pipelining:** Pass on tuples to parent operations even while an operation is still being executed.

Materialization

It is easiest to understand intuitively how to evaluate an expression by looking at a pictorial representation of the expression in an **operator tree**.

Example 3.2: Consider the expression:

$\Pi_{Staff\_name}(\sigma_{Dept\_id=3}$ (Department) $\bowtie$ Staff)

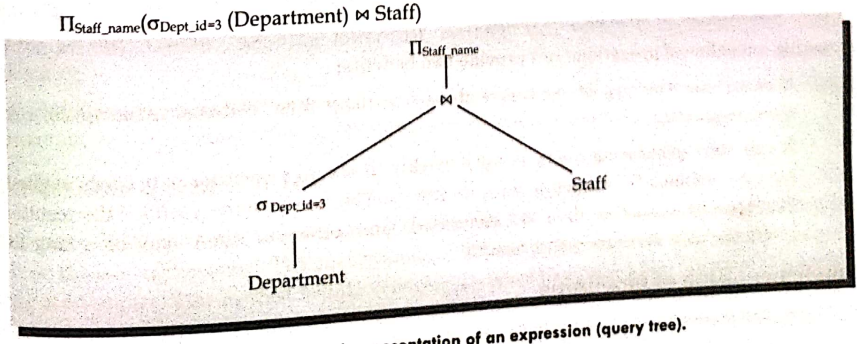

Figure 3.6: Pictorial representation of an expression (query tree).

In this method, the given expression evaluates one relational operation at a time. Also, each operation is evaluated in an appropriate sequence or order. After evaluating all the operations, the outputs are materialized in a temporary relation for their subsequent uses. The example of figure 3.6 is computed as following:

1. Compute $\sigma_{Dept\_id=3}$ (Department) and store relation1
2. Compute Staff $\bowtie$ materialized relation1 and store relation2
3. Compute Π_{Staff_name} on materialized relation2

By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression. In our example, we get the final result by executing the projection operation at the root of the tree, using as input the temporary relation created by the join.

The cost of this type of evaluation is always more leading to a disadvantage. The disadvantage is that it needs to construct those temporary relations for materializing the results of the evaluated operations, respectively. These temporary relations are written on the disks unless they are small in size.

Double buffering (using two buffers, with one continuing execution of the algorithm while the other is being written out) allows the algorithm to execute more quickly by performing CPU activity in parallel with I/O activity. The number of seeks can be reduced by allocating extra blocks to the output buffer, and writing out multiple blocks at once.

Pipelining

In this method, DBMS do not store the records into temporary tables. Instead, it queries each query and result of which will be passed to next query to process and so on. It will process the query one after the other and each will use the result of previous query for its processing. **Pipelining** evaluates multiple operations simultaneously by-passing results of one operation to the next one without storing the tuples on the disk.

In the example of figure 3.6, all three operations can be placed in a pipeline, which passes the results of the selection to the join as they are generated. In turn, it passes the results of the join to the projection as they are generated. The memory requirements are low, since results of an operation are not stored for long. However, as a result of pipelining, the inputs to the operations are not available all at once for processing.

Creating a pipeline of operations can provide two benefits:

- It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation.
- It can start generating query results quickly, if the root operator of a query evaluation plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated, since otherwise there may be a long delay before the user sees any query results.

Implementation of pipelining

Pipelines can be executed in either of two ways:

1. **Demand-driven (or Lazy evaluation) pipelining:** In this method, the result of lower-level queries is not passed to the higher level automatically. It will be passed to higher level only when it is requested by the higher level. In this method, it retains the result value and state with it and it will be transferred to the next level only when it is requested.

2. **Producer-driven (or Eager) pipelining:** In this method, the lower-level queries eagerly pass the results to higher level queries. It does not wait for the higher-level queries to request for the results. In this method, lower-level query creates a buffer to store the results and the higher-level queries pulls the results for its use. If the buffer is full, then the lower-level query waits for the higher-level query to empty it. Hence it is also called as PULL and PUSH pipelining.

QUERY OPTIMIZATION

The function of query optimization engine is to find an evaluation plan that reduces the overall execution cost of a query. We have seen in the previous sections that the costs for performing particular operations such as select and join can vary quite dramatically.

Example 3.3: Consider 2 relations r and s, with the following characteristics:

10,000 = n_r = Number of tuples in r

1,000 = n_s = Number of tuples in s

1,000 = b_r = Number of blocks with tuples in r

100 = b_s = Number of blocks with tuples in s

Selecting a single record from r on a non-key attribute can have,

- a cost of $\lceil log_2(b_r) \rceil = 10$ (binary search) or
- a cost of $b_r/2 = 5{,}000$ (linear search).

Joining r and s can have,

- a cost of $n_r \times b_s + b_r = 1{,}001{,}000$ (nested-loop join) or
- a cost of $3(b_r + b_s) + 4n_h = 73{,}000$ (hash-join where $n_h = 10{,}000$).

Notice that the cost difference between the 2 selects differs by a factor of 500, and the 2 joins by a factor of ~14. Clearly, selecting lower-cost methods can result in substantially better performance.

Query optimization strategies for lowering the execution time of queries includes: *cost-based optimization, heuristic-based optimization* and *semantic-based optimization*.

Cost-based Optimization

This process of selecting a lower-cost mechanism is known as cost-based optimization. This is based on the cost of the query. The query can use different paths based on indexes, constraints, sorting methods etc. This method mainly uses the statistics like record size, number of records, number of records per block, number of blocks, table size, whether whole table fits in a block, organization of tables, uniqueness of column values, size of columns etc. Some of the features of the cost-based optimization are as follows:

- It is based on the cost of the query that to be optimized.
- The query can use a lot of paths based on the value of indexes, available sorting methods, constraints, etc.
- The aim of query optimization is to choose the most efficient path of implementing the query at the possible lowest minimum cost in the form of an algorithm.
- The cost of executing the algorithm needs to be provided by the query Optimizer so that the most suitable query can be selected for an operation.
- The cost of an algorithm also depends upon the cardinality of the input.

Heuristic-based Optimization

Heuristic optimization transforms the query-tree by using a set of rules (Heuristics) that typically (but not in all cases) improve execution performance. Some common the common heuristic rules are:

- Perform selection early (reduces the number of tuples)
- Perform projection early (reduces the number of attributes)
- Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations

Initially query tree from SQL statement is generated. Query tree is transformed into more efficient query tree, via a series of tree modifications, each of which hopefully reduces the execution time. A single query tree is involved at last.

Semantic-based Optimization

This strategy uses constraints specified on the database schema—such as unique attributes and other more complex constraints—in order to modify one query into another query that is more efficient to execute.

Example 3.4: Consider the following SQL query:

```
SELECT e.lname, m.lname
FROM EMPLOYEE as e, EMPLOYEE as m
WHERE e.super_ssn=m.ssn and e.salary>m.salary;
```

This query retrieves the names of employees who earn more than their supervisors. Suppose that we had a constraint on the database schema that stated that no employee can earn more than his or her direct supervisor. If the semantic query optimizer checks for the existence of this constraint, it does not need to execute the query at all because it knows that the result of the query will be empty. This may save considerable time if the constraint checking can be done efficiently. However, searching through many constraints to find those that are applicable to a given query and that may semantically optimize it can also be quite time-consuming. With the inclusion of active rules and additional metadata in database systems, semantic query optimization techniques are being gradually incorporated into the DBMSs.

Expression and Tree Transformations

After a high-level query (i.e., SQL statement) has been parsed into an equivalent relational algebra expression, the query optimizer can perform heuristic rules on the expression and tree to transform the expression and tree into equivalent, but optimized forms.

Example 3.5: Consider the following SQL query:

```
SELECT Stu_name, Marks_Obtained
FROM  Student, Marks
Where Stu_id=10 and Sub_id=20;
```

A corresponding relational algebra expression is:

$\Pi_{Stu\_name,\ Marks\_Obtained}(\sigma_{Stu\_id=10}\ (\sigma_{Sub\_id=20}\ (Student \bowtie Marks)))$

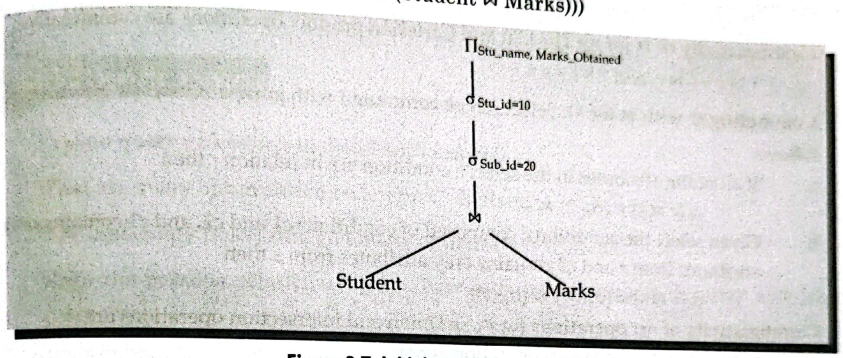

Figure 3.7: Initial expression tree

Suppose the Student and Marks relations both have 100 records each and the number of Stu_id=10 is 50. Note that the Cartesian product resulting in 10,000 records can be reduced by 50% if the σ Stu_id=10 operation is performed first. We can also combine the σ Sub_id=20 and Cartesian product operations into a more efficient join operation, as well as eliminating any unneeded columns before the expensive join is performed. The diagram below shows this better, "optimized" version of the tree:

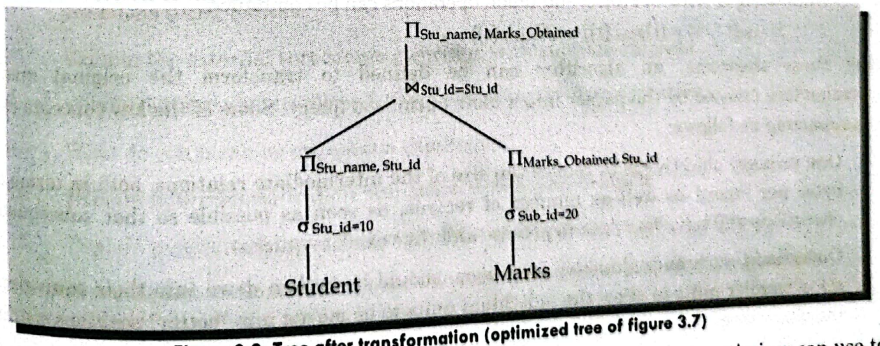

Figure 3.8: Tree after transformation (optimized tree of figure 3.7)

In relational algebra, there are several definitions and theorems the query optimizer can use to transform the query. For instance, the definition of **equivalent relations** states that the set of attributes (domain) of each relation must be the same—because they are sets, the order does not matter. Here is a partial list of relational algebra theorems:

1. **Cascade of σ:** A select with conjunctive conditions on the attribute list is equivalent to a cascade of selects upon selects:
   $\sigma_{A1 \wedge A2 \wedge \ldots \wedge An}(r) \equiv \sigma_{A1}(\sigma_{A2}(\ldots(\sigma_{An}(r))\ldots))$
2. **Commutativity of σ:** The select operation is commutative:
   $\sigma_{A1}(\sigma_{A2}(r)) \equiv \sigma_{A2}(\sigma_{A1}(r))$
3. **Cascade of Π:** A cascade of project operations is equivalent to the last project operation of the cascade:
   $\Pi_{AList1}(\Pi_{AList2}(\ldots(\Pi_{AListn}(r))\ldots)) \equiv \Pi_{AList1}(r)$
4. **Commutating σ with Π:** Given a Π's and σ's attribute of $A_1, A_2, \ldots\ldots, A_n$, the Π and σ operations can be commuted:
   $\Pi_{A1,A2,\ldots\ldots,An}(\sigma_c(r)) \equiv \sigma_c(\Pi_{A1,A2,\ldots\ldots,An}(r))$
5. **Commutativity of ⋈ (or ×):** The join and Cartesian product operations are commutative:
   $r \bowtie s \equiv s \bowtie r$ and $r \times s \equiv s \times r$
6. **Commuting σ with ⋈ (or ×):** Select can be commuted with join (or Cartesian product) as follows:
   a. If all of the attributes in the select's condition are in relation *r* then
      $\sigma_c(r \bowtie s) \equiv (\sigma_c(r)) \bowtie s$
   b. Given select the condition c composed of conditions c1 and c2, and c1 contains only attributes from r and c2 contains only attributes from s then
      $\sigma_c(r \bowtie s) \equiv (\sigma_{c1}(r)) \bowtie (\sigma_{c2}(s))$
7. **Commutativity of set operations (∪, ∩, -):** Union and intersection operations are commutative; but the difference operation is not:
   $r \cup s \equiv s \cup r$, $r \cap s \equiv s \cap r$, $r\text{-}s \neq s\text{-}r$
8. **Associativity of ⋈, ×, ∪ and ∩:** All four of these operations are individually associative. Let θ be any one of these operators, then:
   $(r \theta s) \theta t \equiv r \theta (s \theta t)$
9. **Commuting σ with set operations (∪, ∩, -):** Let θ be any one of the three set operations, then:
   $\sigma_c(r \theta s) \equiv (\sigma_c(r)) \theta (\sigma_c(s))$
10. **Commuting Π with ∪:** Project and union operations can be commuted:
    $\Pi_{AList}(r \cup s) \equiv (\Pi_{AList}(r)) \cup (\Pi_{AList}(s))$

Using these theorems, an algorithm can be defined to transform the original query expression/tree created by the parser into a more optimized query. Some of the key concepts can be summarized as follows:

1. One primary objective is to reduce the size of the intermediate relations, both in terms of bytes per record as well as number of records, as soon as possible so that subsequent operations will have less data to process and thus execute quicker.
2. Operations, such as conjunctive selections, should be broken down into their equivalent set of smaller units to allow the individual units to be moved into "better" positions within the query tree.
3. Combine Cartesian products with corresponding selects to create joins—utilizing optimized join algorithms like the sort-merge join and hash join can be orders of magnitude more efficient.

4. Move selects and projects as far down the tree as possible, as these operations will produce smaller intermediate relations that can be processed more quickly by the operations above.

Choice of Evaluation Plans

The query optimization engine typically generates a set of candidate evaluation plans. Some will, in heuristic theory, produce a faster, more efficient execution. Others may, by prior historical results, be more efficient than the theoretical models—this can very well be the case for queries dependent on the semantic nature of the data to be processed. Still others can be more efficient due to "outside agencies" such as network congestion, competing applications on the same CPU, etc. Thus, excess of data can exist from which the query execution engine can probe for the best evaluation plan to execute at any given time.

1. Explain query processing in detail with example.
2. What are query optimization techniques? Explain.
3. How does query processing and query optimization related?
4. Write the possible relational-algebra expression of following SQL query and draw their query tree.

   SELECT Stu_name, Dept_id

   FROM Student

   WHERE Dept_id <=2;
5. How do you measure the cost of query? Explain.
6. Define access path. Write the formula to calculate the cost of searching algorithm selections using indices.
7. Explain the external sort-merge algorithm with suitable example.
8. What are the main strategies for implementing the Join operation?
9. What do you mean by evaluation expression?
10. Explain how materialization evaluation works with example?
11. Explain pipelining approach of evaluation of expression in detail.
12. Contrast cost estimation and heuristic rules with regard to query optimization.
13. Discuss semi-join and anti-join as operations to which nested queries may be mapped; provide an example of each.
14. How outer join and non–equi are join implemented?

15. How does a query tree represent a relational algebra expression? What is meant by an execution of a query tree? Discuss the rules for transformation of query trees, and identify when each rule should be applied during optimization.
16. What is meant by semantic query optimization? How does it differ from other query optimization techniques?
17. What is the difference between pipelining and materialization?
18. What are the problems associated with keeping views materialized?
19. What do you mean by query processing? What are the various steps involved in query processing? Explain with the help of a block diagram.
20. Discuss the cost components for a cost function that is used to estimate query execution cost. Which cost components are used most often as the basis for cost functions?

❑❑❑