Database Management Systems Laboratory

Assignment 5

Simple Query Optimizer

Team Celestial DB

Aritra Chakraborty
21CS10009

Bratin Mondal
21CS10016

Sarika Bishnoi
21CS10058

Anish Datta
21CS30006

Somya Kumar
21CS30050

Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur

Contents

1 Objective

Simple Query Optimizer - Develop a cost-based query planner and optimizer aimed at enhancing the performance of database queries. The objective is to transform rudimentary, unoptimized execution plans into efficient ones. This involves crafting a suitable representation of query plan nodes and devising a mechanism for estimating costs associated with different plan configurations. Additionally, the optimizer should integrate techniques for managing computational resources and limiting search efforts.

In this project, we take a query as input and produce an optimized query as output. By analyzing the structure and characteristics of the input query, our optimizer aims to generate execution plans that minimize resource consumption and execution time while maximizing performance. Such optimization is crucial for database management systems as it improves overall query processing efficiency, leading to faster response times and better utilization of system resources.

2 Methodology

The methodology employed in this project involves utilizing Lex (lexical analyzer), Bison, and Yacc (parser generators) for parsing queries. Lex is used to generate lexical analyzers that tokenize input queries into meaningful tokens, while Bison and Yacc are employed to construct the parser responsible for interpreting the syntactic structure of the queries. By leveraging these powerful tools, we establish a robust foundation for processing and analyzing the input queries, facilitating subsequent optimization steps.

2.1 Grammar

Grammar Rules:

- **result** → **result** *table*
  This rule indicates that a result consists of a previous result followed by a table.
- **result** → **result** *table* `\n`
  Similar to the previous rule, but the table is followed by a newline character.
- **result** → **result** *error* `\n`
  Indicates that if an error occurs during the parsing of a result, it will be followed by a newline character.
- **exp** → *term*
  Expression consists of a single term.
- **exp** → **exp** `AND` *term*
  Expression can be combined with 'AND' operator and another term.
- **term** → *WORD*
  A term is a single word.
- **term** → **term** `OR` *WORD*
  Terms can be combined with 'OR' operator and another word.
- **table** → *term*
  A table can be represented by a single term.
- **table** → { *table* }
  A table can be enclosed within curly braces, indicating a nested table.
- **table** → `SELECT [` *exp* `]` { *table* `X` *table* }
  A table can be a Cartesian product (`X`) of two other tables, with an optional selection condition.
- **table** → `SELECT [` *exp* `]` { *table* `JOIN [` *exp* `]` *table* }
  A table can be the result of a join operation between two other tables, with an optional selection condition.
- **table** → `SELECT [` *exp* `]` { *table* - *table* }
  A table can be the result of set difference operation (-) between two other tables, with an optional selection condition.
- **table** → `SELECT [` *exp* `]` { *table* }
  A table can be simply selected with an optional selection condition.
- **table** → `PROJECT [` *exp* `]` { *table* }
  A table can be projected onto specific attributes, with an optional selection condition.

- **table** → `PROJECT [` *exp* `] {` *table* `U` *table* `}`
  A table can be the union (`U`) of two other tables, with an optional projection and selection condition.
- **table** → *table* ∩ *table*
  A table can be the intersection (∩) of two other tables.
- **table** → *table* `U` *table*
  A table can be the union (`U`) of two other tables.
- **table** → *table* `-` *table*
  A table can be the set difference (`-`) between two other tables.
- **table** → *table* `JOIN [` *exp* `]` *table*
  A table can be the result of a join operation between two other tables, with an optional selection condition.

2.2 Syntax for Relational Algebra

The following symbols represent various operations in relational algebra:

- σ: **SELECT** [condition] - Selects tuples from a relation that satisfy the given condition.
- Π: **PROJECT** [attributes] - Projects specified attributes from a relation.
- $\bowtie$: **JOIN** [condition] - Performs a join operation between two relations based on the specified condition.
- $\times$: **PRODUCT** - Computes the Cartesian product of two relations.
- $\cup$: **UNION** - Computes the union of two relations.
- $\cap$: **INTERSECTION** - Computes the intersection of two relations.
- $-$: **DIFF** - Computes the set difference between two relations.
- $\wedge$: **AND** - Represents logical AND operation.
- $\vee$: **OR** - Represents logical OR operation.

Consider the following query transformation:

Actual Query: $\Pi(a)(\Pi(b)(\sigma(c \wedge d)(T)))$

Our Query: PROJECT [a] (PROJECT [b] (SELECT [c AND d] (T)))

In this example, the original query is transformed into our query format. The original query involves projection of attributes a and b from a relation T, after applying a selection condition $c \wedge d$. Our query follows the same structure, using relational algebra operations to achieve the same result.

2.3 Optimization

2.3.1 Selection Optimization

$$\sigma_{(a \wedge b)}(T) = \sigma_{(a)}(\sigma_{(b)}(T))$$

Figure 1: Selection Optimization

Explanation: Applying a selection on a table followed by another selection generally results in faster performance compared to applying both selections together. This is because each selection operation reduces the number of tuples earlier in the query plan, potentially reducing the size of the intermediate result set and subsequent operations. By applying selections sequentially, the optimizer can exploit indexes efficiently, leading to optimized query execution plans.

2.3.2 Projection Optimization

$$\prod_{(a_1)}(\prod_{(a_2)}(...(\prod_{(a_n)}(T)...)) = \prod_{(a_1)}(T)$$

Figure 2: Projection Optimization

Explanation: A cascade or series of projections is often meaningless as only the columns specified in the last, or outermost projection are selected. Therefore, collapsing all projections into just one, i.e., the outermost projection, is more efficient. This optimization reduces the overhead of unnecessary projection operations and minimizes the data transferred between query operators, resulting in improved query performance.

2.3.3 Theta Join Optimization with Intersection

$$\sigma_\theta(T_1 \times T_2) = T_1 \bowtie_\theta T_2$$

Figure 3: Theta Join Optimization with Intersection

Explanation: Theta Join drastically reduces the number of resulting tuples. By applying an intersection of the join conditions into the Theta Join itself, the number of scans required is significantly reduced, leading to improved efficiency. This optimization leverages the properties of the join conditions to minimize the size of the intermediate result set, resulting in faster query execution.

2.3.4 Theta Join Optimization with Intersection of Join Conditions

$$\sigma_{\theta_1}(T_1 \bowtie_{\theta_2} T_2) = T_1 \bowtie_{(\theta_1 \wedge \theta_2)} T_2$$

Figure 4: Theta Join Optimization with Intersection of Join Conditions

Explanation: Similar to the previous optimization, by intersecting both join conditions within the Theta Join itself, the number of resulting tuples is further reduced, leading to fewer scans and improved performance. This optimization is particularly effective when the join conditions have common predicates, allowing the optimizer to exploit shared computation and minimize redundant processing.

2.3.5 Selection Optimization with Set Difference

$$\sigma_\theta(T_1 - T_2) = \sigma_\theta(T_1) - \sigma_\theta(T_2)$$

Figure 5: Selection Optimization with Set Difference

Explanation: Applying a selection condition on the entire set difference is equivalent to applying the selection condition on the individual tables first and then performing the set difference operation. This reduces the number of comparisons required in the set difference step, enhancing efficiency. By filtering the input tables before the set difference operation, the optimizer reduces the size of the intermediate result set, leading to faster query processing.

2.3.6 Projection Optimization before Union

$$\prod_{\theta}(T_1 \bigcup T_2) = \prod_{\theta}(T_1) \bigcup \prod_{\theta}(T_2)$$

Figure 6: Projection Optimization before Union

Explanation: Applying individual projections before computing the union of two expressions is more optimal than applying projection after the union step. This optimization reduces the size of the intermediate result set before the union operation, leading to improved performance. By projecting only the necessary attributes early in the query plan, the optimizer minimizes data transfer and processing overhead, resulting in faster query execution.

2.3.7 Join Optimization

Explanation: For n join operations in a query, determining the optimal order of execution is crucial for minimizing computational cost. Dynamic Programming (DP) techniques are commonly employed to select the most efficient join order. The goal is to minimize the total cost of executing all joins, which typically includes factors such as disk I/O, memory usage, and processing time.

Formulation: Let $R_1, R_2, ..., R_n$ be the relations involved in the join operations, and let JoinCost(i, j) represent the cost of joining relations R_i and R_j. The optimal join order π^* is determined by solving the following recursive equation:

$$\text{JoinCost}(i,j) = \min_{i \le k < j} \{\text{JoinCost}(i,k) + \text{JoinCost}(k+1,j) + \text{Cost}(R_i \bowtie R_k \bowtie R_{k+1} \bowtie ... \bowtie R_j)\}$$

where Cost$(R_i \bowtie R_k \bowtie R_{k+1} \bowtie ... \bowtie R_j)$ represents the cost of joining relations $R_i, R_{i+1}, ..., R_j$ in a specific order.

Dynamic Programming: The optimal join order problem can be efficiently solved using dynamic programming. We construct a DP table to store intermediate results, where each entry corresponds to the cost of joining a subset of relations. By filling the DP table in a bottom-up manner, we can determine the optimal join order and minimize the overall cost.

Time Complexity: The time complexity of solving the optimal join order problem using dynamic programming is $O(3^n)$, where n is the number of join operations.

Space Complexity: The space complexity is $O(2^n)$ due to the storage of intermediate results in the DP table.

R_1	R_2	R_3	R_4	R_5
JoinCost$(1,1)$	JoinCost$(1,2)$	JoinCost$(1,3)$	JoinCost$(1,4)$	JoinCost$(1,5)$
-	JoinCost$(2,2)$	JoinCost$(2,3)$	JoinCost$(2,4)$	JoinCost$(2,5)$
-	-	JoinCost$(3,3)$	JoinCost$(3,4)$	JoinCost$(3,5)$
-	-	-	JoinCost$(4,4)$	JoinCost$(4,5)$
-	-	-	-	JoinCost$(5,5)$

Figure 7: Illustration of Dynamic Programming Table for Join Optimization

In Figure 7, we illustrate how the DP table is filled to determine the optimal join order. Each cell in the table represents the cost of joining a subset of relations, and the optimal join order is determined by selecting the minimum cost path from the top-left corner to the bottom-right corner.

Although we coded this DP algorithm, it is worth noting that the time complexity grows exponentially with the number of join operations, making it infeasible for large queries. In practice, database systems employ sophisticated optimization techniques, such as cost-based query optimizers and query rewrite rules, to efficiently process complex queries.

3 Results

below we provide some input-output pairs for the query optimizer:
Input-Output Pairs:

- **Selection Optimization:**
  - Input: `SELECT [ A AND B ] ( T )`
  - Output: `SELECT [ A ] ( SELECT [ B ] ( T ) )`
  - Input: `SELECT [ A AND B AND C AND D AND E ] ( T )`
  - Output: `SELECT [ A ] (SELECT [ B ] (SELECT [ C ] (SELECT [ D ] (SELECT [ E ] (T)))))`
- **Projection Optimization:**
  - Input: `PROJECT [ A1 ] ( PROJECT [ A2 ] (T))`
  - Output: `PROJECT [ A1 ] (T)`
  - Input: `PROJECT [ A1 ] ( PROJECT [ A2 ] ( PROJECT [ A3 ] ( PROJECT [ A4 ] (T))))`
  - Output: `PROJECT [ A1 ] (T)`
- **Theta Join Optimization with Intersection:**
  - Input: `SELECT [ A1 ] (T1 PROD T2)`
  - Output: `T1 JOIN [ A1 ] T2`
  - Input: `SELECT [ A1 ] (T1 PROD ( SELECT [ A2 ] ( T2 PROD T3 )))`
  - Output: `T1 JOIN [ A1 ] T2 JOIN [ A2 ] T3`
- **Theta Join Optimization with Intersection of Join Conditions:**
  - Input: `SELECT [ A1 ] (T1 JOIN [ B1 ] T2)`
  - Output: `T1 JOIN [ A1 AND B1 ] T2`
  - Input: `SELECT [ A1 ] ( SELECT [ A2 ] ( T1 JOIN [ B1] T2 ))`
  - Output: `T1 JOIN [ A1 AND A2 AND B1 ] T2`
- **Selection Optimization with Set Difference:**
  - Input: `SELECT [ A1 ] (T1 DIFF T2)`
  - Output: `SELECT [ A1 ] (T1) DIFF SELECT [ A1 ] (T2)`
  - Input: `SELECT [A2] ( (SELECT [ A1 ] (T1 DIFF T2) ) DIFF T3)`
  - Output: `SELECT [ A2 ] (SELECT [ A1 ] (T1) DIFF SELECT [ A1 ] (T2)) DIFF SELECT [ A2 ] (T3)`
- **Projection Optimization before Union:**
  - Input: `PROJECT [ A1 ] (T1 UNION T2)`
  - Output: `PROJECT [ A1 ] (T1) UNION PROJECT [ A1 ] (T2)`
  - Input: `PROJECT [A2] ( ( PROJECT [ A1 ] (T1 UNION T2)) UNION T3)`
  - Output: `PROJECT [ A2 ] (T1) UNION PROJECT [ A2 ] (T2) UNION PROJECT [ A2 ] (T3)`
- **Join Optimization:**
  - Input: `T1 JOIN [ A1 ] T2 JOIN [ A2 ] T3 JOIN [ A3 ] T4 JOIN [ A4 ] T5`
  - Output: `(( T1 JOIN [ A1 ] T2 ) JOIN [ A2 ] T3 ) JOIN [ A3 ] ( T4 JOIN [ A4 ] T5 )`
  - Input: `T1 JOIN [ A1 ] T2 JOIN [ A2 ] T3 JOIN [ A3 ] T4 JOIN [ A4 ] T5 JOIN [ A5 ] T6`
  - Output: `(( T1 JOIN [ A1 ] T2 ) JOIN [ A2 ] T3 ) JOIN [ A3 ] ( (T4 JOIN [ A4 ] T5 ) JOIN [ A5 ] T6 )`
  - Input: `T1 JOIN [ A1 ] T2 JOIN [ A2 ] T3 JOIN [ A3 ] T4 JOIN [ A4 ] T5 JOIN [ A5 ] T6 JOIN [ A6 ] T7`
  - Output: `((( T1 JOIN [ A1 ] T2 ) JOIN [ A2 ] T3 ) JOIN [ A3 ] ( (T4 JOIN [ A4 ] T5 )) JOIN [ A5 ] (T6 JOIN [ A6 ] T7))`

We further test our code on mixed queries and queries with multiple operations. The optimizer successfully processes these queries, applying the appropriate optimizations to generate efficient execution plans. The optimizer's ability to handle complex queries demonstrates its robustness and effectiveness in enhancing query performance. Here are some input-output pairs for mixed queries and queries with multiple operations:

- Input: `SELECT [ A1 AND B5 AND C5 ] ((T))`
- Output: `SELECT [ A1 ] (SELECT [ B5 ] (SELECT [ C5 ] (T)))`
- Input: `SELECT [ * ] (SELECT [ B1 AND C7] (T))`
- Output: `SELECT [ * ] (SELECT [ B1 ] (SELECT [ C7 ] (T)))`
- Input: `PROJECT [ B1 ] (PROJECT [ A2 ] (T))`
- Output: `PROJECT [ B1 ] (T)`
- Input: `SELECT [ A1 AND B5 ] (PROJECT [ B1 ] (PROJECT [A2] (T)))`
- Output: `SELECT [ A1 ] (SELECT [ B5 ] (PROJECT [ B1 ] (PROJECT [ A2 ] (T))))`
- Input: `PROJECT [ B1 ] (PROJECT [ A2 ] (SELECT [ A1 AND B5 ] (T)))`
- Output: `PROJECT [ B1 ] (PROJECT [ A2 ] (SELECT [ A1 ] (SELECT [ B5 ] (T))))`
- Input: `SELECT [ A3 ] (T1 DIFF T2)`
- Output: `SELECT [ A3 ] (T1) DIFF SELECT [ A3 ] (T2)`
- Input: `PROJECT [ A5 ] (T1 UNION T2)`
- Output: `PROJECT [ A5 ] (T1) UNION PROJECT [ A5 ] (T2)`
- Input: `SELECT [ A3 ] (PROJECT [ A5 ] (T1 UNION T2) DIFF T2)`
- Output: `SELECT [ A3 ] (PROJECT [ A5 ] (T1 UNION T2)) DIFF SELECT [ A3 ] (T2)`
- Input: `SELECT [ A4 ] (T1 PROD T2)`
- Output: `T1 JOIN [ A4 ] T2`
- Input: `SELECT [ A7 ] (T1 JOIN [ B3 ] T2)`
- Output: `T1 JOIN [ A7 AND B3 ] T2`
- Input: `PROJECT [ A5 ] (PROJECT [ B7 ] (SELECT [ A3 ] (T1 DIFF T2) UNION T2))`
- Output: `PROJECT [ A5 ] (SELECT [ A3 ] (T1) DIFF SELECT [ A3 ] (T2)) UNION PROJECT [ A5 ] (T2)`
- Input: `SELECT [ A3 AND B9 ] (SELECT [ A1 AND B5 ] (T) DIFF T2)`
- Output: `SELECT [ A3 ] (SELECT [ B9 ] (SELECT [ A1 ] (SELECT [ B5 ] (T)))) DIFF SELECT [ A3 ] (SELECT [ B9 ] (T2))`
- Input: `SELECT [ C4 AND M4] (SELECT [ A3 ] (SELECT [ A1 AND B5 ] (T) DIFF T2) INTERSECT PROJECT [ B1 ] (PROJECT [ A2 ] (T))`
- Output: `SELECT [ C4 ] (SELECT [ M4 ] (SELECT [ A3 ] (SELECT [ A1 ] (SELECT [ B5 ] (T))) DIFF SELECT [ A3 ] (T2) INTERSECT PROJECT [ B1 ] (T))`
- Input: `SELECT [ A8 AND M4 ] (T1 UNION T2 INTERSECT T3)`
- Output: `SELECT [ A8 ] (SELECT [ M4 ] (T1 UNION T2 INTERSECT T3))`
- Input: `SELECT [ C5 ] (SELECT [ A7 AND M2 ] (T1 PROD T2))`
- Output: `T1 JOIN [ C5 AND A7 AND M2 ] T2`

A whole demo of the code can be found at link

4 References

1. Database Management Systems by Raghu Ramakrishnan and Johannes Gehrke
2. Database System Concepts by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan
3. A Detailed Guide on SQL Query Optimization
4. A Comprehensive Guide to SQL Query Optimization Techniques
5. Database performance and query optimization
6. Awesome Database Testing

5 GitHub Repository

The code for the Simple Query Optimizer can be found at the following GitHub repository: Query Optimiser