Indian Institute of Technology Kharagpur

Query Processor and Optimizer

A Cost-Based Query Optimization System

Course: Database Management Systems (CS30202)

Instructors: Prof. Pabitra Mitra & Prof. K.S. Rao

Group Members

Name	Roll Number
Dadi Sasank Kumar	22CS10020
Jeevan Varma	22CS10038
Gurram Dhanunjay	22CS10029
Venkata Yaswanth	22CS30031
Nerella Trilochan	22CS10048

Submitted: April 15, 2025

Contents

1 Problem Statement

The objective of this project is to develop a query processor and optimizer for a relational database system. The system parses SQL SELECT queries, constructs an abstract syntax tree (AST), and optimizes the execution plan using selection pushdown and projection pushdown techniques. By leveraging table and column statistics, the optimizer estimates costs to select the most efficient plan, minimizing computational resources and execution time. The system supports SQL constructs such as SELECT, FROM, WHERE, JOIN, and aggregate functions (COUNT, MAX, MIN, AVG), providing detailed cost analyses for different execution plans.

This report includes a detailed analysis of the following SQL query:

```
SELECT employees.name, departments.dept_name
FROM employees
JOIN departments
ON employees.dept_id = departments.dept_id
WHERE departments.dept_name = 'Engineering';
```

The query leverage the available table and column statistics for cost-based optimization, constrained by the hardcoded metadata in `stats.cpp/stats.hpp`.

2 Methodology

The system is implemented in C, utilizing Flex for lexical analysis, Bison for parsing, and custom modules for optimization and statistics management. The processing pipeline consists of the following stages:

2.1 Lexical Analysis

The lexical analyzer (`lexer.l`) tokenizes SQL queries using Flex. It recognizes keywords (e.g., SELECT, FROM, WHERE, JOIN), identifiers, numbers, and operators in a case-insensitive manner. Debugging options enable token tracing for validation.

2.2 Parsing

The parser (`parser.y`), built with Bison, generates an AST from the tokenized query. Nodes represent operations (projection π, selection σ, join $\bowtie$), tables, conditions, and expressions. The parser handles column selections, table aliases, joins, conditions, and subqueries. The resulting AST, shown in Figure 1, captures the query's logical structure.

```
Original Abstract Syntax Tree:
π(employees.name,departments.dept_name)
  σ(departments.dept_name = Engineering)
    ⋈(employees.dept_id = departments.dept_id)
      table(employees)
      table(departments)
```

Figure 1: Original Abstract Syntax Tree

2.3 Statistics Management

The statistics module (`stats.cpp`, `stats.hpp`) initializes hardcoded metadata for four tables: `employees`, `salaries`, `departments`, and `projects`. Each table includes row counts, column counts, and column statistics (distinct values, min/max values, selectivity). Functions like `calculate_join_selectivity` and `calculate_condition_selectivity` estimate the impact of joins and conditions, supporting cost-based optimization.

2.4 Query Optimization

The optimizer (`optimizer.cpp`, `optimizer.hpp`) applies two optimization techniques:

- **Selection Pushdown**: Reorders selection operations (σ) closer to table scans to filter rows early, reducing intermediate result sizes.
- **Projection Pushdown**: Moves projection operations (π) toward table scans to limit column processing, minimizing data overhead.

The `estimate_cost` function calculates result size, column count, and cost (rows $\times$ columns), while `calculate_total_plan_cost` incorporates selectivity and column ratios. The optimizer evaluates multiple plans and selects the one with the lowest cost.

2.5 Main Program

The main program (`main.cpp`) reads a query from `query.sql`, parses it, optimizes the AST, and outputs execution plans with cost estimates. It integrates all modules to produce optimized query plans.

2.6 Key Data Structures

- **Node** (`parser.hpp`): Represents AST nodes with fields for operation, arguments, child, and sibling pointers.
- **JoinOrderNode** (`optimizer.hpp`): Supports join order optimization (partially implemented).
- **CostMetrics** (`optimizer.hpp`): Stores result size, column count, and cost.
- **TableStats** and **ColumnStats** (`stats.hpp`): Hold table and column metadata.

3 Results and Demonstration

The system was tested with a sample query selecting `employees.name` and `departments.dept_name` with a condition `departments.dept_name = 'Engineering'` and a join on `employees.dept_id = departments.dept_id`. The results demonstrate parsing and optimization effectiveness across multiple stages.

3.1 Original Execution Plan

The unoptimized execution plan, derived from the AST, includes all operations in their parsed order, with costs estimated using table statistics. The plan structure is as follows:

```
--- Original Plan ---
π(employees.name,departments.dept_name) [rows=500, cols=2, cost=1000.0]
  σ(departments.dept_name = Engineering) [rows=500, cols=7, cost=3500.0]
    ⋈(employees.dept_id = departments.dept_id) [rows=10000, cols=7, cost=70000.0]
      table(employees) [rows=10000, cols=4, cost=40000.0]
      table(departments) [rows=20, cols=3, cost=60.0]
```

Figure 2: Original Execution Plan

The cost breakup is detailed in Table 1.

Table 1: Cost Breakup for Original Plan

Node Type	Description	Node Cost	Cumulative Cost
π	employees.name,departments.dept_name	1000.0	45883.5
σ	departments.dept_name = Engineering	3500.0	49413.0
$\bowtie$	employees.dept_id = departments.dept_id	70000.0	47060.0
table	employees	40000.0	40000.0
table	departments	60.0	60.0
Total			45883.5

3.1.1 Cost Calculation for Original Plan

The cumulative cost is computed bottom-up using `calculate_total_plan_cost`:

- **Table (employees)**:
  - Node Cost: $10000 \times 4 = 40000.0$
  - Cumulative Cost: For a table, returns `estimate_cost.cost`.
  - Formula: return current.cost
  - Result: 40000.0
- **Table (departments)**:
  - Node Cost: $20 \times 3 = 60.0$
  - Cumulative Cost: 60.0
  - Formula: return current.cost
  - Result: 60.0
- **Join ($\bowtie$)**:
  - Node Cost: $10000 \times 7 = 70000.0$
  - Selectivity: `get_join_selectivity` returns 0.05 (default).
  - Child Costs: Left (employees): 40000.0, Right (departments): 60.0

- Cumulative Cost:
  - Formula: left_cost + right_cost + (result_size × num_columns × 0.1)
  - Calculation: 40000.0+60.0+(10000×7×0.1) = 40060.0+7000.0 = 47060.0
- Result: 47060.0

- **Selection (σ)**:
  - Node Cost: $500 \times 7 = 3500.0$
  - Selectivity: `get_condition_selectivity` returns 0.05.
  - Child Cost (join): 47060.0
  - Cumulative Cost:
    - Formula: child_cost × (1.0 + selectivity)
    - Calculation: $47060.0 \times (1.0 + 0.05) = 47060.0 \times 1.05 = 49413.0$
  - Result: 49413.0

- **Projection (π)**:
  - Node Cost: $500 \times 2 = 1000.0$
  - Child Cost (selection): 49413.0
  - Column Ratio: num_columns/child_num_columns = $2/7 \approx 0.2857$
  - Cumulative Cost:
    - Formula: child_cost × (0.9 + 0.1 × column_ratio)
    - Calculation: $49413.0 \times (0.9 + 0.1 \times 0.2857) = 49413.0 \times (0.9 + 0.02857) \approx 49413.0 \times 0.92857 \approx 45883.5$
  - Result: 45883.5

The total cost is the cumulative cost of the root node (π): 45883.5. Node costs ($1000.0 + 3500.0 + 70000.0 + 40000.0 + 60.0 = 114560.0$) don't sum to the total because cumulative costs incorporate selectivity (1.05 for σ, 0.05 for $\bowtie$) and column ratio (0.2857 for π), reflecting filtering and projection efficiencies. The table shows subtree costs, with the root's cost as the total plan cost.

3.2 Selection Pushdown Optimization

Selection pushdown reorders the selection to filter `departments` before the join, reducing the join's input size. The plan structure is:

Figure 3 illustrates this plan.

```
--- Selection Pushdown Plan ---
π(employees.name,departments.dept_name) [rows=500, cols=2, cost=1000.0]
  ⋈(employees.dept_id = departments.dept_id) [rows=500, cols=7, cost=3500.0]
    table(employees) [rows=10000, cols=4, cost=40000.0]
    σ(departments.dept_name = Engineering) [rows=1, cols=3, cost=3.0]
      table(departments) [rows=20, cols=3, cost=60.0]
```

Figure 3: Selection Pushdown Execution Plan

The cost breakup is provided in Table 2.

Table 2: Cost Breakup for Selection Pushdown Plan

Node Type	Description	Node Cost	Cumulative Cost
π	employees.name,departments.dept_name	1000.0	37526.4
$\bowtie$	employees.dept_id = departments.dept_id	3500.0	40413.0
table	employees	40000.0	40000.0
σ	departments.dept_name = Engineering	3.0	63.0
table	departments	60.0	60.0
Total			37526.4

3.2.1 Cost Calculation for Selection Pushdown Plan

The cumulative cost is computed bottom-up:

- **Table (departments)**:
  - Node Cost: $20 \times 3 = 60.0$
  - Cumulative Cost: 60.0
  - Formula: return current.cost
  - Result: 60.0
- **Selection (σ)**:
  - Node Cost: $1 \times 3 = 3.0$
  - Selectivity: 0.05
  - Child Cost (departments): 60.0
  - Cumulative Cost:
    - Formula: child_cost $\times$ (1.0 + selectivity)
    - Calculation: $60.0 \times (1.0 + 0.05) = 60.0 \times 1.05 = 63.0$
  - Result: 63.0

- **Table (employees)**:
  - Node Cost: $10000 \times 4 = 40000.0$
  - Cumulative Cost: 40000.0
  - Formula: return current.cost
  - Result: 40000.0
- **Join ($\bowtie$)**:
  - Node Cost: $500 \times 7 = 3500.0$
  - Selectivity: 0.05
  - Child Costs: Left (employees): 40000.0, Right (σ): 63.0
  - Cumulative Cost:
    - Formula: left_cost + right_cost + (result_size $\times$ num_columns $\times$ 0.1)
    - Calculation: $40000.0 + 63.0 + (500 \times 7 \times 0.1) = 40063.0 + 350.0 = 40413.0$
  - Result: 40413.0
- **Projection (π)**:
  - Node Cost: $500 \times 2 = 1000.0$
  - Child Cost (join): 40413.0
  - Column Ratio: $2/7 \approx 0.2857$
  - Cumulative Cost:
    - Formula: child_cost $\times$ (0.9 + 0.1 $\times$ column_ratio)
    - Calculation: $40413.0 \times (0.9 + 0.1 \times 0.2857) = 40413.0 \times (0.9 + 0.02857) \approx 40413.0 \times 0.92857 \approx 37526.4$
  - Result: 37526.4

The total cost is 37526.4. The selection reduces `departments` to 1 row before the join, lowering the join's result to 500 rows (vs. 10000), reducing the cost (40413.0 vs. 47060.0). Node costs ($1000.0 + 3500.0 + 40000.0 + 3.0 + 60.0 = 44563.0$) don't sum to the total due to selectivity (1.05 for σ, 0.05 for $\bowtie$) and column ratio (0.2857 for π). The table reflects subtree costs, with the root's cost as the total.

3.3 Projection Pushdown Optimization

Projection pushdown applies projections early to reduce columns before the join and selection. The plan structure is:

```
--- Projection Pushdown Plan ---
σ(departments.dept_name = Engineering) [rows=500, cols=2, cost=1000.0]
  π(employees.name,departments.dept_name) [rows=10000, cols=2, cost=20000.0]
    ⋈(employees.dept_id = departments.dept_id) [rows=10000, cols=2, cost=20000.0]
      π(employees.name,employees.dept_id) [rows=10000, cols=2, cost=20000.0]
        table(employees) [rows=10000, cols=4, cost=40000.0]
      π(departments.dept_name,departments.dept_id) [rows=20, cols=2, cost=40.0]
        table(departments) [rows=20, cols=3, cost=60.0]
```

Figure 4: Projection Pushdown Execution Plan

The cost breakup is detailed in Table 3.

Table 3: Cost Breakup for Projection Pushdown Plan

Node Type	Description	Node Cost	Cumulative Cost
σ	departments.dept_name = Engineering	1000.0	42060.9
π	employees.name,departments.dept_name	20000.0	40058.0
$\bowtie$	employees.dept_id = departments.dept_id	20000.0	40058.0
π	employees.name,employees.dept_id	20000.0	38000.0
table	employees	40000.0	40000.0
π	departments.dept_name,departments.dept_id	40.0	58.0
table	departments	60.0	60.0
Total			42060.9

3.3.1 Cost Calculation for Projection Pushdown Plan

The cumulative cost is computed bottom-up:

- **Table (departments)**:
  - Node Cost: $20 \times 3 = 60.0$
  - Cumulative Cost: 60.0
  - Formula: return current.cost
  - Result: 60.0
- **Projection (π, departments.dept_name,departments.dept_id)**:
  - Node Cost: $20 \times 2 = 40.0$
  - Child Cost (departments): 60.0
  - Column Ratio: $2/3 \approx 0.6667$
  - Cumulative Cost:
    - Formula: child_cost $\times$ $(0.9 + 0.1 \times$ column_ratio$)$
    - Calculation: $60.0 \times (0.9 + 0.1 \times 0.6667) = 60.0 \times (0.9 + 0.06667) \approx 60.0 \times 0.96667 \approx 58.0$

  - Result: 58.0

- **Table (employees)**:
  - Node Cost: $10000 \times 4 = 40000.0$
  - Cumulative Cost: 40000.0
  - Formula: return current.cost
  - Result: 40000.0

- **Projection (π, employees.name,employees.dept_id)**:
  - Node Cost: $10000 \times 2 = 20000.0$
  - Child Cost (employees): 40000.0
  - Column Ratio: $2/4 = 0.5$
  - Cumulative Cost:
    - Formula: child_cost $\times$ $(0.9 + 0.1 \times$ column_ratio$)$
    - Calculation: $40000.0 \times (0.9 + 0.1 \times 0.5) = 40000.0 \times (0.9 + 0.05) = 40000.0 \times 0.95 = 38000.0$
  - Result: 38000.0

- **Join ($\bowtie$)**:
  - Node Cost: $10000 \times 2 = 20000.0$
  - Selectivity: 0.05
  - Child Costs: Left (π employees): 38000.0, Right (π departments): 58.0
  - Cumulative Cost:
    - Formula: left_cost + right_cost + (result_size $\times$ num_columns $\times$ 0.1)
    - Calculation: $38000.0 + 58.0 + (10000 \times 2 \times 0.1) = 38058.0 + 2000.0 = 40058.0$
  - Result: 40058.0

- **Projection (π, employees.name,departments.dept_name)**:
  - Node Cost: $10000 \times 2 = 20000.0$
  - Child Cost (join): 40058.0
  - Column Ratio: $2/2 = 1.0$
  - Cumulative Cost:
    - Formula: child_cost $\times$ $(0.9 + 0.1 \times$ column_ratio$)$
    - Calculation: $40058.0 \times (0.9 + 0.1 \times 1.0) = 40058.0 \times (0.9 + 0.1) = 40058.0 \times 1.0 = 40058.0$
  - Result: 40058.0

- **Selection (σ)**:
  - Node Cost: $500 \times 2 = 1000.0$
  - Selectivity: 0.05

- Child Cost (π): 40058.0
- Cumulative Cost:
  - Formula: child_cost $\times$ (1.0 + selectivity)
  - Calculation: $40058.0 \times (1.0 + 0.05) = 40058.0 \times 1.05 \approx 42060.9$
- Result: 42060.9

The total cost is 42060.9. Projections reduce columns early (2 vs. 7 in join), but the join processes 10000 rows since selection is applied last. Node costs ($1000.0 + 20000.0 + 20000.0 + 20000.0 + 40000.0 + 40.0 + 60.0 = 101000.0$) don't sum to the total due to column ratio reductions (0.95, 0.96667) and selectivity (0.05). The table shows subtree costs, with the root's cost as the total.

3.4 Cost Comparison

The optimizer compares the three plans based on total cost, result size, and column count. The comparison is visualized in Figure 5 and summarized in Table 4.

```
Cost Comparison:
Metric          | Original      | Selection     | Projection    |
----------------|---------------|---------------|---------------|
Result Size     | 500           | 500           | 500           |
Columns         | 2             | 2             | 2             |
Total Cost      | 45883.5       | 37526.4       | 42060.9       |

Selection Pushdown is the best plan(lowest cost) with total cost 37526.4
```

Figure 5: Cost Comparison of Execution Plans

Table 4: Cost Comparison Across Plans

Metric	Original	Selection Pushdown	Projection Pushdown
Result Size	500	500	500
Columns	2	2	2
Total Cost	45883.5	37526.4	42060.9

The selection pushdown plan is the most efficient because it reduces the join's input size to 1 row for `departments`, yielding a join result of 500 rows, lowering the cost to 37526.4. The original plan (45883.5) processes 10000 rows in the join, while projection pushdown (42060.9) reduces columns but not rows until the final selection.

4 Conclusion and Future Scope

The query processor and optimizer successfully parse SQL queries, generate ASTs, and produce optimized execution plans. The selection pushdown plan achieves the lowest cost (37526.4), as shown in Table 2, by filtering rows early. Projection pushdown reduces column overhead but incurs a higher cost (42060.9) due to processing 10000 rows in the join, as seen in Table 3. The original plan is the least efficient (45883.5), confirming the value of optimization.

4.1 Future Scope

- **Join Reordering**: Implement `reorder_joins` and `find_optimal_join_order` to optimize join sequences.
- **Dynamic Statistics**: Replace hardcoded statistics with metadata file updates.
- **Subquery Optimization**: Enhance support for nested and correlated subqueries.
- **Parallel Execution**: Introduce parallel processing for joins and selections.

5 References

1. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill Education.
2. Flex & Bison documentation: `https://www.gnu.org/software/flex/`, `https://www.gnu.org/software/bison/`
3. Ramakrishnan, R., & Gehrke, J. (2002). *Database Management Systems* (3rd ed.). McGraw-Hill Education.