OptiQuery: Query Processing and Optimization in SQL

Anaswar K B	220138	anaswar22@iitk.ac.in
Amogh Bhagwat	220288	bhagwata22@iitk.ac.in
Khushi Gupta	220531	khushig22@iitk.ac.in
Pathe Nevish Ashok	220757	pathena22@iitk.ac.in
Wadkar Srujan Nitin	221212	nsrujan22@iitk.ac.in

Abstract

SQL Query Optimization is a hot research area in the field of database systems as the scale of data continues to grow and inefficient SQL queries can cause significant bottlenecks in database performance. In this project, we implement a basic SQL query optimization system by integrating query parsing, conversion to relational algebra and optimization strategies like predicate pushdown and join reordering. A visualizer is also provided to view the changes in query plan and the estimated cost of the plan.

1 Motivation and Problem Statement

Modern applications—from e-commerce platforms to scientific data warehouses—generate and consume massive volumes of data. As datasets grow into the terabyte and petabyte scale, even seemingly simple SQL queries can become performance bottlenecks, leading to slow response times and increased infrastructure costs. Database systems rely on sophisticated query optimizers to transform high-level SQL statements into efficient execution plans; without such optimizations, joins and filters might be applied in suboptimal order, unnecessarily processing large intermediate results.

Despite the central role of query optimization in database engines, building a fully general optimizer is a complex undertaking, involving cost models, plan enumeration, and integration with catalog statistics. This project illustrates the core ideas—parsing SQL into relational algebra, pushing down predicates to reduce intermediate row counts, and reordering joins based on simple cost estimates.

In this project, we address the following problem statement:

- **Input:** An SQL `SELECT FROM WHERE JOIN` query (with optional nested subqueries) over known table schemas.
- **Objective:** Automatically transform the input into an optimized relational algebra plan that minimizes a cost metric based on estimated row counts.
- **Outcome:** A proof-of-concept optimizer that demonstrates significant cost reductions on representative queries, illustrating the practical benefits of basic query rewriting techniques.

2 Methodology

The system is designed in a modular approach, having modules for parsing the SQL query, estimating cost of a query plan and separate modules for each optimization technique. The general flow begins with the

parser converting the SQL query to a relational algebra tree. Then, the desired optimizations are applied on the tree. Finally, the estimated cost is calculated. The costs and relational algebra tree at each stage can be visualized in the web interface provided.

2.1 SQL Query Parsing

The SQL parser takes input an SQL query and builds a relational algebra tree for it. Note that there are some limitations in our parser - we support only a small subset of queries having `SELECT`, `FROM` `WHERE`, `JOIN` and nested subqueries. We do not support aggregations or other SQL constructs due to the complexity of building such a parser. Example SQL query and its output:

```
SELECT P.PARTKEY, P.SUPPKEY
FROM PARTSUPP P
JOIN SUPPLIER S ON P.SUPPKEY = S.SUPPKEY
WHERE P.AVAILQTY > 10 OR P.SUPPLYCOST < 500 AND S.ACCTBAL > 1000;
```

```
Projection(["P.PARTKEY", "P.SUPPKEY"],
    Selection("WHERE P.AVAILQTY > 10 OR P.SUPPLYCOST < 500
               AND S.ACCTBAL > 1000",
        Join(
            Relation("PARTSUPP AS P"), Relation("SUPPLIER AS S"),
            "P.SUPPKEY = S.SUPPKEY"
        )
    )
)
```

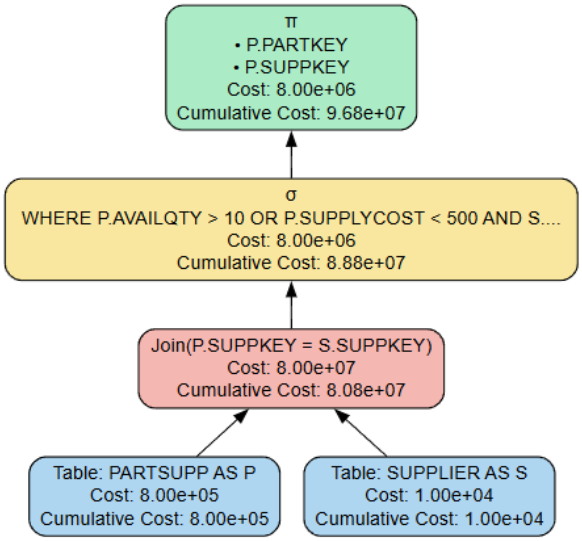


Figure 1: Visualization of Relational Algebra Tree

2.2 Predicate Pushdown

Predicate pushdown is a very effective optimization rule in SQL queries. This is because the in the relational algebra tree obtained after parsing queries, the `Selection` operations generally occur at the top of the tree. If these can be done early on during table scanning, the cost of the subsequent operations like joins can be reduced significantly. The tree after predicate pushdown is as shown.

PushdownSelections

Function `PushdownSelections`(node)

- If node is a SELECTION:
  - Normalize the selection predicate (strip leading "WHERE")
  - Split out any top-level conjuncts and push each one recursively
  - If the (possibly split) predicate sits above a JOIN:
    - Determine which table aliases the predicate actually references
    - If all references belong to the join's left input:
      - Push the selection into the left child and rebuild the join
    - If all references belong to the join's right input:
      - Push the selection into the right child and rebuild the join
    - Else:
      - Leave the selection in place (cannot push further)
- Return the (possibly moved) selection node
- If node is a PROJECTION:
  - Recursively optimize its child
  - Return a projection over the optimized child
- If node is a JOIN:
  - Recursively optimize both left and right inputs
  - Return a join of the two optimized children, using the same join condition
- If node is a SUBQUERY:
  - Recursively optimize the inner RA tree
  - Return a subquery node over the optimized child
- Else:
  - (node is a RELATION leaf) Return node unchanged

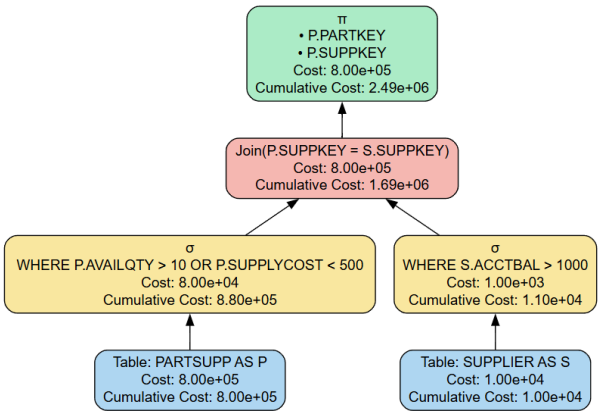
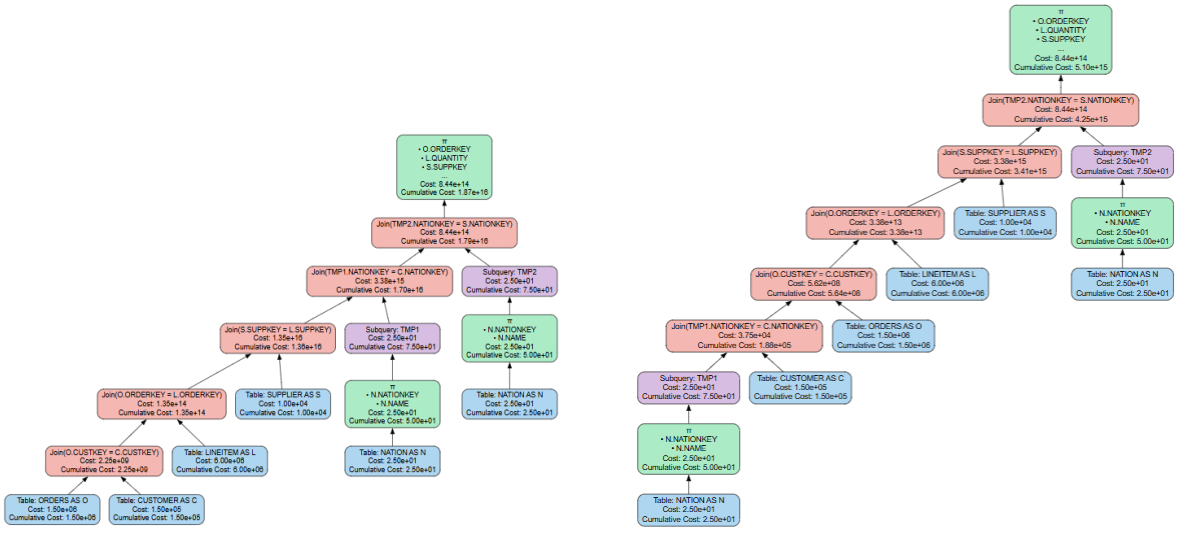


Figure 2: Tree after Predicate Pushdown

2.3 Join Optimization

The join optimizer is the most computation-intensive phase in the query optimization pipeline. It takes as input the relational algebra tree and outputs a reordered join plan that minimizes query execution cost using cost-based optimization. This can offer significant benefits on performance, especially in cases where multiple tables are joined together.

The following figures show the query plan before and after join optimization.



(a) Before applying Join Optimization

(b) After applying Join Optimization

Figure 3: Result of Join Optimization

Part 1: Traversal & Extraction of Joins

1. **Initialize state:**
   - Create empty list `edges`.
   - Create empty map `aliasMap`.
   - Set flag `foundFirst` to `false`.
   - Reserve variable `tempRoot`.
2. **Recursive traversal:**
   (a) If `foundFirst` is `false`:
      - If current node is a Join with condition $\neq$ `TRUE`:
        - Set `tempRoot` to parent.
        - Set `foundFirst` to `true`.
      - Else, if node has a single child, recurse into that child.

   (b) If `foundFirst` is `true`:
      - Extract tables (t_1, t_2) from the join condition.
      - Append $(t_1, t_2, \text{condition})$ to `edges`.
      - For each side (left, right):
        - If side is a Join, recurse into it.
        - Else, record `aliasMap[alias]` = subtree node.
3. **Outcome:** `tempRoot`, `edges`, and `aliasMap` ready.

Part 2: Permutation & Rebuilding of Optimal Join Order

1. **Setup:**
   - Let $N = |\texttt{edges}| + 1$. If $N < 2$, return original tree.
   - Initialize `bestCost` $= \infty$, `bestOrder = null`.
2. **Evaluate permutations:**
   (a) For each permutation P of `edges`:
   - Seed cost with first edge (A, B): $\max(50, cost(A) \times cost(B) \times 0.01)$.
   - Mark A, B visited; set `cumulativeCost`.
   - For each subsequent edge (X, Y):
     - If exactly one of X, Y is visited, extend join, compute new cost, add to `cumulativeCost`.
     - Else, permutation invalid—abort.
   - If valid and `cumulativeCost < bestCost`, update `bestCost,bestOrder`.
3. **Rebuild subtree:**
   - From `bestOrder`, create initial join node.
   - Iteratively attach each next relation to the growing join tree.
   - Assign resulting subtree to `tempRoot.child`.
4. **Return:** Optimized root.

2.4 Cost Estimation

The cost for each operation is estimated by the row counts of the result. The row counts of each table is obtained from the PostgreSQL API. Estimation heuristics are used to estimate the sizes after selection, projection and joining. The total cost of the query is the sum of the costs of all operations. This is maintained by storing the cumulative cost of a single operation, which is the sum of costs of all its children. This estimated cost is used to perform cost-based optimization techniques like join optimization.

3 Implementation and Results

We have used Flask for integrating the user interface with the query optimizer framework. For parsing the queries we have used `sqlglot` library. To test the effectiveness of our query optimization scheme, we test it on the **TPC-H Benchmark** database. The code and setting up of TPC-H database is present in our GitHub repository `https://github.com/AmoghBhagwat/OptiQuery`. Since we support only a small subset of queries, we created a set of our own queries aimed at testing all aspects of our system. Overall, we achieved an average reduction of **85.93%** in cost. The database schema and queries are given in the appendix.

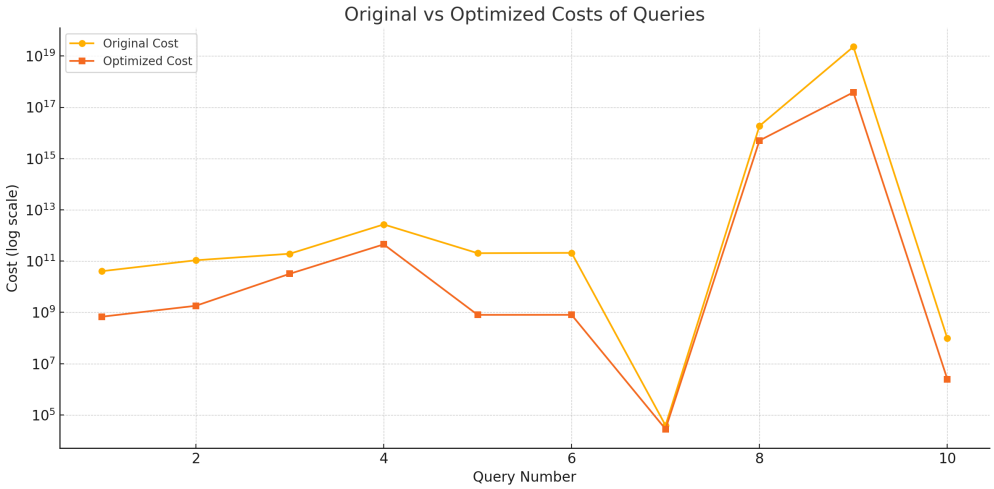


Figure 4: Original vs Optimized Query Costs

Query No.	Original Cost	Optimized Cost	% Improvement
1	4.05e+10	6.79e+08	98.32%
2	1.08e+11	1.81e+09	98.32%
3	1.94e+11	3.21e+10	83.47%
4	2.70e+12	4.50e+11	83.33%
5	2.04e+11	8.01e+08	99.61%
6	2.10e+11	8.03e+08	99.62%
7	3.90e+04	2.81e+04	27.95%
8	1.87e+16	5.06e+15	72.94%
9	2.30e+19	3.84e+17	98.33%
10	9.68e+07	2.49e+06	97.43%

Table 1: Original vs Optimized Query Costs

4 Conclusion

In this work, we presented *OptiQuery*, a lightweight SQL query optimizer that demonstrates the core techniques of parsing, predicate pushdown, and join reordering within a unified, modular framework. By converting SQL into relational algebra, pushing filters as close to the base relations as possible, and exhaustively exploring join orders under a simple cost model, our prototype consistently produces execution plans with substantially lower estimated costs compared to the original, unoptimized plans.

Evaluation on a set of queries shows that even basic optimizations—when applied systematically—can reduce intermediate result sizes and overall plan cost by a significant margin. The visualizer component further aids understanding by displaying each transformation step and the associated cost estimates.

Future extensions include support for aggregations and `GROUP BY` and more sophisticated cost models. Overall, *OptiQuery* serves as an educational foundation and a starting point for experimenting with advanced query optimizer features.

5 Discussion and Limitations

- **Join Optimization Scope:**
  - Supports only *equality* join predicates (e.g., `A.id = B.id`); arbitrary non-equi or range joins are not handled.
  - Enumerates only *left-deep* join trees. Bushy plans, which can be optimal in some cases, are not considered.
  - Assumes a single join condition per pair of relations; queries requiring multi-condition joins or cross-products (Cartesian joins) are not optimized and will remain in their original order.
- **SQL Feature Coverage:**
  - The parser accepts only basic `SELECT FROM WHERE JOIN` with simple nested subqueries; it does not support `GROUP BY`, `HAVING`, `UNION`, or `EXISTS`.
  - Aggregations, `DISTINCT` projections, and outer joins `(LEFT/RIGHT/FULL)` are outside the current scope.
- **Cost Model Simplifications:**
  - Relies on cardinality estimates derived from PostgreSQL's row counts and uniform-distribution heuristics; it does not incorporate histograms, correlation statistics, or selectivity sampling.
  - Assumes cost is proportional to intermediate result size; CPU, I/O, and network costs are not differentiated.

6 Contributions

Here are the contributions of each group member -

- **Anaswar K B** - Join Optimization, Cost Estimation
- **Amogh Bhagwat** - Parser, User Interface, Testing
- **Khushi Gupta** - Parser, Predicate Pushdown
- **Pathe Nevish Ashok** - User Interface, Cost Estimation
- **Wadkar Srujan Nitin** - Join Optimization, Testing

Everyone contributed equally in preparing the report.

A TPC-H Benchmark schema

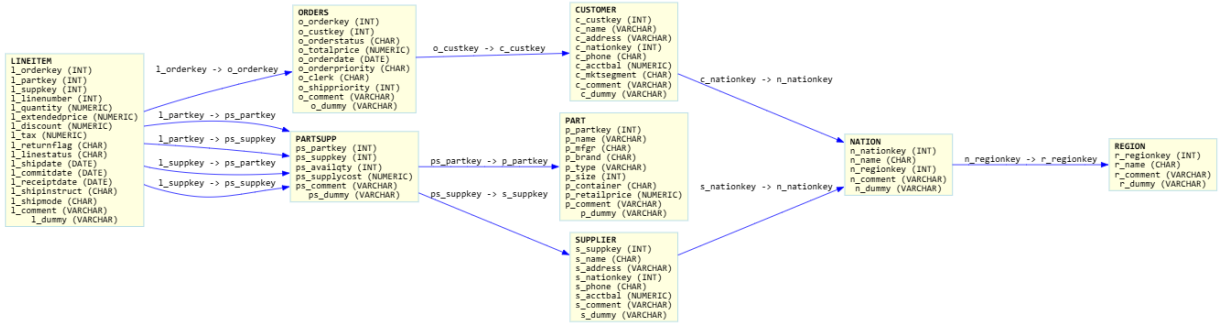


Figure 5: TPC-H Schema

B Testing Queries

```
-- Query 1 ---------------------------------------------------------------------

SELECT H.C_NAME, O.O_ORDERKEY, O.O_TOTALPRICE
FROM (
  SELECT CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME, CUSTOMER.C_ACCTBAL
  FROM CUSTOMER
  WHERE CUSTOMER.C_ACCTBAL > 1000
) H
JOIN (
  SELECT CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME, CUSTOMER.C_ACCTBAL
  FROM CUSTOMER
  WHERE CUSTOMER.C_ACCTBAL > 1000
) I ON H.CUSTOMER.C_NATIONKEY = I.CUSTOMER.C_NATIONKEY
JOIN ORDERS O ON H.CUSTOMER.C_CUSTKEY = O.O_CUSTKEY
WHERE O.O_TOTALPRICE > 50000 AND O.O_ORDERSTATUS = 'O' OR O.O_SHIPPRIORITY = 0;

-- Query 2 ---------------------------------------------------------------------

SELECT L.L_ORDERKEY, L.L_PARTKEY
FROM LINEITEM L
JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
WHERE L.L_SHIPDATE >= '1994-01-01' AND O.O_ORDERSTATUS = 'F';

-- Query 3 ---------------------------------------------------------------------

SELECT P.P_NAME, P.P_BRAND
```

```
FROM PART P
JOIN PARTSUPP PS ON P.P_PARTKEY = PS.PS_PARTKEY
JOIN SUPPLIER S ON PS.PS_SUPPKEY = S.S_SUPPKEY
WHERE P.P_SIZE >= 25;

-- Query 4 -----------------------------------------------------------------

SELECT S.S_NAME, N.N_NAME, L.L_EXTENDEDPRICE, O.O_ORDERDATE
FROM SUPPLIER S
JOIN NATION N ON S.S_NATIONKEY = N.N_NATIONKEY
JOIN LINEITEM L ON S.S_SUPPKEY = L.L_SUPPKEY
JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
WHERE L.L_DISCOUNT > 0.05;

-- Query 5 -----------------------------------------------------------------

SELECT
    P.P_NAME,
    PS.PS_SUPPLYCOST,
    S.S_NAME,
    N.N_NAME,
    R.R_NAME
FROM PART P
JOIN PARTSUPP PS ON P.P_PARTKEY = PS.PS_PARTKEY
JOIN SUPPLIER S ON PS.PS_SUPPKEY = S.S_SUPPKEY
JOIN NATION N ON S.S_NATIONKEY = N.N_NATIONKEY
JOIN REGION R ON N.N_REGIONKEY = R.R_REGIONKEY
WHERE PS.PS_SUPPLYCOST < 300
  AND R.R_NAME = 'EUROPE';

-- Query 6 -----------------------------------------------------------------

SELECT P.P_NAME, S.S_NAME, N.N_NAME, PS.PS_SUPPLYCOST
FROM PART P
JOIN PARTSUPP PS ON P.P_PARTKEY = PS.PS_PARTKEY
JOIN SUPPLIER S ON PS.PS_SUPPKEY = S.S_SUPPKEY
JOIN NATION N ON S.S_NATIONKEY = N.N_NATIONKEY
WHERE P.P_TYPE = 'COPPER'
  AND PS.PS_SUPPLYCOST < 500;

-- Query 7 -----------------------------------------------------------------

SELECT H.S_NAME, I.S_PHONE, N.N_NAME
FROM (
  SELECT SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY
```

```
  FROM SUPPLIER
  WHERE SUPPLIER.S_ACCTBAL > 5000
) H
JOIN (
  SELECT SUPPLIER.S_SUPPKEY, SUPPLIER.S_PHONE, SUPPLIER.S_NATIONKEY
  FROM SUPPLIER
  WHERE SUPPLIER.S_ACCTBAL > 5000
) I ON H.SUPPLIER.S_NATIONKEY = I.SUPPLIER.S_NATIONKEY
JOIN NATION N ON H.SUPPLIER.S_NATIONKEY = N.N_NATIONKEY
WHERE N.N_NAME = 'UNITED STATES';

-- Query 8 ---------------------------------------------------------------

SELECT O.ORDERKEY, L.QUANTITY, S.SUPPKEY, C.CUSTKEY
FROM ORDERS O
JOIN CUSTOMER C ON O.CUSTKEY = C.CUSTKEY
JOIN LINEITEM L ON O.ORDERKEY = L.ORDERKEY
JOIN SUPPLIER S ON S.SUPPKEY = L.SUPPKEY
JOIN (SELECT N.NATIONKEY, N.NAME FROM NATION N) TMP1
    ON TMP1.NATIONKEY = C.NATIONKEY
JOIN (SELECT N.NATIONKEY, N.NAME FROM NATION N) TMP2
    ON TMP2.NATIONKEY = S.NATIONKEY;

-- Query 9 ---------------------------------------------------------------

SELECT PS.PARTKEY, PS.SUPPLYKEY
FROM PARTSUPP PS
JOIN SUPPLIER S ON PS.SUPPLYKEY = S.SUPPLYKEY
JOIN LINEITEM L ON PS.SUPPLYKEY = L.SUPPLYKEY
JOIN (SELECT P.NAME, P.BRAND FROM PART P) TMP1 ON TMP1.PARTKEY = PS.PARTKEY
JOIN (SELECT P.NAME, P.BRAND FROM PART P) TMP2 ON TMP2.PARTKEY = L.PARTKEY
WHERE PS.AVAILQTY > 10 AND S.ACCTBAL > 1000;

-- Query 10 --------------------------------------------------------------

SELECT P.PARTKEY, P.SUPPKEY
FROM PARTSUPP P
JOIN SUPPLIER S ON P.SUPPKEY = S.SUPPKEY
WHERE P.AVAILQTY > 10 OR P.SUPPLYCOST < 500 AND S.ACCTBAL > 1000;

-------------------------------------------------------------------------
```