CS6999 : MASTER'S PROJECT FINAL REPORT

TECHNICAL REPORT: FALL 2024

Hersh Dhillon
Master of Science in Computer Science
Georgia Institute of Technology
hdhillon30@gatech.edu

1 Introduction : Query Optimization

For CS6999 Master's Project course under Prof. Joy Arulraj, I am working on Query Optimization. In this report I provide details about the POC's I built for Learned Indices and Join Ordering along with details about the Literature Review. I then explain about the test bench I am currently working on which will help us test for various cost functions, selectivity, join algorithms and join ordering algorithms. This report is divided into the following parts

1. **POCs for learned indices**
2. **Join ordering simulator**
3. **SimpleDBMS and creating a C++ codebase inspired from that**

The github repositories with the code and the data are present here

- Contains readings POCs and src for SimpleDBMS inspired code simulator `https://github.com/hershd23/query_optimization`
- Java code for Simple DBMS `https://github.com/hershd23/SimpleDBMS`

2 Literature Review

To get more context about the Query Optimization I read the following papers

- AI driven Query Optimization : `https://github.com/hershd23/query_optimization/blob/main/readings/IJCTT-V72I3P103.pdf`
- Neo, a learned query optimizer : `https://github.com/hershd23/query_optimization/blob/main/readings/p1705-marcus.pdf`
- How good are query optimizers really? : `https://github.com/hershd23/query_optimization/blob/main/readings/p204-leis.pdf`

Along with the papers I also have used the following courses to supplement my foundations for Query Optimization

- `https://db.in.tum.de/teaching/ws1819/queryopt/?lang=en`
- `https://15721.courses.cs.cmu.edu/spring2024/`

3 POC for Learned Indices

I first experimented with Learned Indices. For that I created a POC for learned indices present here.

`https://github.com/hershd23/query_optimization/tree/main/poc`

NOTE: The paper "How good are query optimizers, really?" discusses how modern data workloads can be processed fully in RAM, hence reading from the disk is becoming less and less often in modern workloads, hence the experiments (in other sections as well) done are mostly in-memory

I obtained for following results for the three strategies for in-memory searching.

Experiment:

- Created a dataset of size 1000000 and range [0, 2000000]
- Did 1000 searches for random numbers in this range.
- Compared the search metrics in the below table

Metric	LI + Linear Search	LI + Binary Search	Binary Search
Total Operations	3655481	9267	19482
Total Time	595ms	0.14ms	0.34ms

We see that Learned Indices plus Linear search around the search radius is multiple magnitudes slower than binary searching. While binary searching is possible in an in memory setting it is not feasible for on disk settings. To further verify this I tested on the z-regression learned indices and z-neural learned index in the Buzz DB repo which runs a similar experiment

Metric	LI (Regression)	LI (NN)	Binary Search
Total Time	19us	11us	1us

We see here as well that there is a magnitude difference in the search time between Learned Indices and Binary Searching. Due to this I started exploring Join Ordering.

4 Join Ordering Simulator

I also created a Join Ordering Simulator comparing various strategies for joins here `https://github.com/hershd23/query_optimization/tree/main/join_ordering`. This creates a dummy dataset with 6 tables and employs multiple strategies to figure out the best way to join those tables.

4.1 Dataset Generation

We generated a synthetic dataset comprising six relations (A through F) with varying cardinalities:

- Relation A: 10,000 records
- Relation B: 15,000 records
- Relation C: 20,000 records
- Relation D: 5,000 records
- Relation E: 25,000 records
- Relation F: 8,000 records

To ensure meaningful join operations, we implemented the following data generation characteristics:

- Created a pool of 1,000 shared IDs to guarantee join matches between relations
- Each record consists of an ID field and a data field with a relation-specific prefix
- Maintained a match ratio of 0.1 (10%) for join compatibility between relations
- Utilized the `generateRecordsWithJoinMatch` function to ensure controlled distribution of shared IDs across relations

4.2 Join Graph Structure

The experiment implements a chain-like join graph with five join conditions:

$$\begin{aligned} A &\leftrightarrow B \quad \text{(selectivity: 0.1)} \\ B &\leftrightarrow C \quad \text{(selectivity: 0.05)} \\ C &\leftrightarrow D \quad \text{(selectivity: 0.2)} \\ D &\leftrightarrow E \quad \text{(selectivity: 0.15)} \\ E &\leftrightarrow F \quad \text{(selectivity: 0.1)} \end{aligned}$$

4.3 Optimization Algorithms

We evaluated four distinct join optimization strategies:

4.3.1 Random Optimizer

- Generates random permutations of join orders
- Serves as a baseline for comparative analysis

4.3.2 Greedy Optimizer

- Initializes with the smallest relation
- Selects subsequent relations based on optimal local join selectivity
- Employs size-based selection as a fallback mechanism when no direct join exists

4.3.3 Dynamic Programming Optimizer

- Implements a bottom-up approach considering all possible relation subsets
- Bounded by a maximum of 16 relations (defined by `MAX_RELATIONS` constant)
- Incorporates both join selectivity and intermediate result sizes in cost calculations

4.3.4 IKKBZ Optimizer

- Implements the IKKBZ algorithm specifically designed for acyclic join graphs
- Requires the join graph to maintain acyclicity
- Utilizes cost estimation based on relation cardinalities and join selectivity factors

4.4 Performance Metrics

For each optimization strategy, we measured and analyzed the following metrics:

1. **Optimization Time (ms)**: Duration required to compute the optimal join order
2. **Join Order**: Sequence of relations in the final execution plan
3. **Estimated Join Cost**: Calculated based on selectivity factors and relation sizes
4. **Join Execution Time (ms)**: Actual duration of join execution in the determined order
5. **Final Result Size**: Cardinality of the final joined result set

4.5 Implementation Details

The implementation incorporates several key features:

- Utilizes hash-based join implementation for efficient execution
- Implements comprehensive cost estimation considering:
  - Relation cardinalities
  - Join selectivity factors

  - Attribute count
  - Cross-product penalties for non-direct joins
- All joins are implemented as inner joins using a hash-based algorithm

4.6 Results

Metric	Random (B1)	Greedy (B2)	IKKBZ	DP
Join Order	FAEDCB	DEFABC	ABCDEF	FEDCBA
Join Execution Time	65ms	24ms	27ms	30ms
Optimization Time	0.001ms	0.006ms	0.010ms	0.028ms
Estimated Cost	0.0015	7.5e-05	1.5e-05	1.5e-05

5 SimpleDBMS

I came across SimpleDBMS, a simple database used by MIT and UW to teach their database courses. The codebase had very neat separations between the various components of the database and tests queries and joins on a real dataset (IMDb dataset).

5.1 Experiments on SimpleDBMS

5.1.1 Test Queries

We evaluated three distinct SQL queries of varying complexity:

1. **Actor's First Movie Year Query**

```
SELECT MIN(m.year)
FROM actor a, casts c, movie m
WHERE a.id=c.pid AND c.mid=m.id
AND a.fname='Tom' AND a.lname='Cruise';
```

2. **Actor-Director Relationship Query**

```
SELECT d.fname, d.lname
FROM actor a, casts c, movie_director m, director d
WHERE a.id=c.pid AND c.mid=m.mid AND m.did=d.id
AND a.lname='Spicer';
```

3. **Director Movie Count Query**

```
SELECT d.lname, count(m1.id)
FROM director d, movie_director m, movie m1
WHERE d.id=m.did AND m.mid=m1.id
GROUP BY d.lname;
```

5.2 Cost Model Configurations

We implemented and compared two different cost model configurations:

5.2.1 Configuration 1 (C1)

- **Scan Cost Calculation**:
  - Cost = Page Cost (static 1000) $\times$ Number of Pages
- **Table Cardinality Estimation**:
  - Cardinality = Number of Tuples $\times$ Selectivity Factor

- **Join Cardinality Estimation**:
  - For Equality Predicates:
    - If neither table has PK: $\max(card_1, card_2)$
    - If one table has PK: cardinality of the PK table
  - For Other Predicates:
    - 30% of cross product
- **Join Cost Model**:

$$Cost = card_1 + card_1 \times cost_2 + card_2 \times cost_1 \tag{1}$$

5.2.2 Configuration 2 (C2)

- **Scan Cost Calculation**:
  - Same as C1: Page Cost (static 1000) $\times$ Number of Pages
- **Table Cardinality Estimation**:
  - Same as C1: Number of Tuples $\times$ Selectivity Factor
- **Join Cardinality Estimation**:
  - For Equality Predicates:
    - If neither table has PK: $\min(card_1, card_2)$
    - If one table has PK: cardinality of the PK table
  - For Other Predicates:
    - $\sqrt{\text{cross product}}$
- **Join Cost Model**:

$$Cost = \log(card_1) + \log(card_1) \times cost_2 + \log(card_2) \times cost_1 \tag{2}$$

Both configurations:

- Used histogram-based selectivity calculation
- Implemented Selinger optimization algorithm
- Maintained same page cost (1000)

5.3 Experimental Results

Table 1: Performance Comparison of Cost Model Configurations

Query	C1 Exec Time	C1 Join Op Time	C2 Exec Time	C2 Join Op Time
1	0.14s	406.4μs	0.11s	409.5μs
2	0.11s	347.0μs	0.10s	306.1μs
3	0.04s	250.5μs	0.04s	271.9μs

5.4 Key Observations

5.4.1 Query Performance

- Configuration C2 consistently performed slightly better or equal to C1 in terms of total execution time
- The difference was most noticeable for Query 1 (0.03s improvement)
- Query 3 showed identical performance across both configurations

5.4.2 Join Operation Times

- Join operation times remained relatively consistent between configurations
- Query 1 showed similar join times ($\sim$400μs) for both configurations
- Query 2 showed slightly better join performance in C2
- Query 3 had marginally better join performance in C1

5.4.3 Cost Model Impact

- The logarithmic cost model in C2 appeared to produce marginally better execution plans
- The min() vs max() cardinality estimation choice for non-PK joins showed impact on optimization decisions
- The sqrt() vs 30% approach for non-equality predicates didn't show significant impact in these queries

6 Creating my own C++ query test bench

Working with SimpleDBMS and Prof's feedback motivated me to create a query analysis test bench in C++ myself. The test bench in located here `https://github.com/hershd23/query_optimization/tree/main/cpp_src`

I go into the created query test bench in more detail.

6.1 Parser

The parser supports a limited SQL grammar and converts SQL-like queries into an Abstract Syntax Tree (AST). The AST consists of nodes that represent different clauses, such as `SELECT`, `FROM`, and `WHERE`. The following are the key components of the grammar:

- **SELECT Clause:** The `SelectNode` class is responsible for parsing the `SELECT` clause. It supports both table-qualified columns (e.g., `table.column`) and unqualified columns (e.g., `column`). Parsed columns are stored in a vector of `Column` structures. The `print()` method outputs the parsed columns for debugging.
- **FROM Clause:** The `FromNode` class parses the `FROM` clause, handling tables and their optional aliases. Tables and aliases are stored in a vector of `TableRef` structures. The `print()` method outputs the parsed tables and aliases.
- **Conditions (WHERE and JOIN):** Conditions are handled by the `Condition` class, supporting comparison operators (e.g., =, >, <) between columns, integers, or strings. The `isJoinCondition` flag distinguishes between filter and join conditions. Conditions are printed for debugging using the `print()` method.
- **WHERE Clause:** The `WhereNode` class parses the `WHERE` clause. It stores conditions in a vector of `Condition` objects, allowing filtering of rows based on column comparisons or constant values.
- **JOIN Clause:** The `JoinNode` class handles the parsing of `JOIN` clauses, supporting joins between tables with conditions. Table aliases are also managed within the join.
- **GROUP BY and HAVING Clauses:** The `GroupByNode` class parses the `GROUP BY` clause, storing the columns by which results are grouped. The `HavingNode` class handles the `HAVING` clause, applying conditions to the groups of results.
- **SQLParser Class:** The `SQLParser` class is the main interface for parsing queries. It processes the query string, skips whitespace, manages table aliases, and constructs the AST using various node classes.

6.2 Executor

The executor is responsible for interpreting and executing the AST generated by the parser. It performs operations such as column selection, row filtering, and table joining based on the parsed query. The following details the key components of query execution:

- **Executing the SELECT Clause:** The executor processes the columns specified by the `SelectNode`. It validates each column against the schema and retrieves the relevant data for output. If the columns are table-qualified, the executor ensures proper reference to the table.
- **Executing the FROM Clause:** The executor loads the tables specified in the `FROM` clause into memory. Aliases are applied as needed, and in cases of multiple tables, the executor handles joins accordingly.
- **Handling Conditions (WHERE and JOIN):** Conditions specified in the `WhereNode` are evaluated, and the executor filters rows based on these conditions. Join conditions, as specified in the `JoinNode`, are used to combine rows from different tables.
- **Debugging and Output:** Throughout execution, the system uses the `print()` methods in each AST node to output the parsed query structure, providing debugging information for the query execution process.

6.3 Queries run and results

The queries are inspired from Join Order Benchmark by Leis et al. `https://github.com/gregrahn/join-order-benchmark/blob/master/10a.sql`

6.4 About the dataset

I work with 10% of the IMDb dataset. Here are the table schema and sizes

Schema

```
actor(id int,fname string,lname string,gender string)
movie(id int,name string,year int)
director(id int,fname string,lname string)
casts(pid int,mid int,role string)
movie_director(did int,mid int)
genre(mid int,genre string)
```

Row sizes

- **actor**: 199,195 rows
- **movie**: 26,412 rows
- **director**: 21,257 rows
- **casts**: 323,818 rows
- **movie_director**: 29,762 rows
- **genre**: 47,512 rows

6.4.1 Query example

```
SELECT director.fname, director.lname, movie.name
FROM movie, movie_director, director
WHERE movie.id=8854 AND movie.id=movie_director.mid
AND movie_director.did=director.id;
```

Multiple plans are generated from this query, we keep a track of the times and mention them in the results

Plan	Planning Time (µs)	Execution Time (µs)
Filtering After Joining	45	59,914,378
Filtering in the Middle	88	32,566,594
Filtering Before Joining	50	7,275

Table 2: Comparison of Execution Plans

We can see that in general filtering before joining gives much better performance, which is known. Hence we take filtering before joining as a general rule before proceeding to joins.

Now we perform join ordering

Plan	Planning Time (µs)	Execution Time (µs)
director-movie_director-movie	39	26820421
movie-movie_director-director	50	7,275

Table 3: Comparison of Execution Plans

Since the filtering happens on the movie table, it makes sense to take the movie table first and then move forward, since movie_director to director joins both tables fully without any filtering

7 Final Test Bench and Multiple Plan Generation

One major problem in the previous design is code manageability due to a lot of parsing logic. To solve for that, I completely revamped the parser to create a different type query language which is much more explicit.

```
query_start
tables: table1, table2, ...
scalar_filters: table.column=value, ...
dynamic_filters: table1.column=table2.column, ...
joins: table1.column=table2.column, ...
query_end
```

This new design enabled me to scale the code better and for each query create and execute 5 different plans.

The plans are the following

- Joins First: Executes all joins before filters
- Filters First: Executes all filters before joins
- Try All Join Orders: Exhaustively tries all possible join orders (with filters first)
- Greedy Join Order: Uses a greedy approach to select join orders
- Dynamic Programming: Uses DP to build optimal plans from smaller subplans

7.1 Test queries

These are the queries on which we test benchmark our solutions. 1. Find director for a movie

```
query_start
tables: movie, director, movie_director
scalar_filters: movie.id=8854
joins: movie_director.did=director.id, movie.id=movie_director.mid
query_end
```

2. Find all male actors who acted in movies directed by Spielberg after 2000

```
query_start
tables: movie, director, movie_director, actor, casts
scalar_filters: director.lname=Spielberg, movie.year>2000, actor.gender=M
joins: movie.id=movie_director.mid, movie_director.did=director.id, movie.id=casts.mid, casts.pid=ac
query_end
```

3. Find all movies directed by Nolan in the Drama genre

```
query_start
tables: movie, director, movie_director, genre
scalar_filters: director.lname=Nolan, genre.genre=Drama
joins: movie.id=movie_director.mid, movie_director.did=director.id, movie.id=genre.mid
query_end
```

7.2 Cost based optimization

The workbench uses several metrics for cost estimation:

- Table sizes and selectivity factors from histograms
- I/O costs for scanning tables
- CPU costs for comparisons
- Join output size estimation

For joins, the output size is estimated as:

```
output_size = min(left_table_size, right_table_size)
```

7.3 Results

7.3.1 Plan Generation Times

Query	Joins First	Filters First	Greedy Joins	DP Joins	Try All
1	0.075	0.046	0.101	0.454	0.225
2	0.086	0.058	0.093	7.507	0.651
3	0.035	0.020	0.039	0.0574	0.075

Table 4: Performance comparison of different query strategies.

7.3.2 Execution times

Query	Joins First	Filters First	Greedy Joins	DP Joins	Try All
1	28158.6	12688.9	3.827	2.386	3.677
2	Too long	481337	881.971	124.587	873.538
3	49368.7	21963.1	18.66	13.943	18.947

Table 5: Performance comparison of query strategies with updated data.

7.4 Analysis

- **Plan Generation Times Variability:** *Greedy Joins* consistently has the lowest plan generation times across all queries, while *DP Joins* has the highest times, particularly for Query 2, which takes over 7 seconds. *Joins First* and *Filters First* are comparable, with *Filters First* slightly faster in most cases.
- **Execution Times Dominance:** *Greedy Joins* and *DP Joins* show significantly faster execution times compared to *Joins First* and *Filters First*. However, *Try All* is competitive with *Greedy Joins*, particularly for Query 3. *Joins First* exhibits extremely high execution times, especially for Query 2, where the time is noted as "Too long," likely due to inefficient intermediate results.
- **Filters vs. Joins Preference in Planning:** *Filters First* demonstrates faster plan generation than *Joins First*, indicating that filtering early reduces the complexity of join planning.
- **Query Complexity Impact:** Query 2 shows a significant increase in both planning and execution times for almost all strategies. This may indicate a more complex query structure or larger intermediate datasets.
- **Performance of Greedy Joins:** Although *Greedy Joins* has a slightly higher plan generation time than *Filters First*, it consistently achieves the fastest execution times, demonstrating the effectiveness of its heuristic-based approach.
- **Execution Time Bottleneck in Complex Queries:** *DP Joins*, while slow in plan generation, maintains competitive execution times across Queries 1 and 3. However, for Query 2, its execution time is significantly slower, indicating it may struggle with very complex or large datasets.
- **Trade-offs Between Strategies:** The *Try All* strategy, while slower in plan generation compared to *Greedy Joins*, performs well during execution, making it a viable alternative for balancing robustness and performance.
- **Choosing the Right Strategy:** *Greedy Joins* appears to be the optimal choice for most scenarios due to its balance of reasonable planning times and fast execution.
- **Handling Complex Queries:** For complex queries like Query 2, it might be worth exploring a hybrid approach, combining the strengths of *DP Joins* and *Greedy Joins* to optimize both planning and execution times.
- **Further Tuning of Joins First:** Significant inefficiencies in the *Joins First* strategy, particularly in execution times, suggest a need for further optimization or relegating it to simpler queries.
- **Impact of Early Filtering:** The faster planning times of *Filters First* highlight the importance of pruning irrelevant data early, which can significantly reduce downstream processing costs.

8 Conclusion and Future work

In this semester, I have learned a lot about query optimization and have learnt to put my coding skills into use by creating the POCs and simulators discussed in this report.

Future work directions

- Add more features to the simulator like the indices
- Experiment with cost models and selectivity estimator functions
- Try and study the impact of model changes and addition of queries like INSERT

9 Learnings and Challenges

- Understanding where the limits of AI generated code are.
- I have learnt to utilize courses and slides available online to quickly ramp up. This can be applied across multiple subjects.
- Taking feedback and turning it around into meaningful progress.
- Understanding and implementing ideas from systems papers.
- Benchmarking multiple query plan generators