Optimizing Graph Database Cardinality Estimation and Query Evaluation

Piyush Manjrekar
Student number: 2168928
Slack username: piyushmanjrekar2905

Aniket Mishra
Student number: 2079259
Slack username: a.mishra3

Ramanan Murugesan
Student number: 2132486
Slack username: r.murugesan

Ramesh Rohit
Student number: 2182920
Slack username: r.ramesh.rohit

Group 14

1 Abstract

Cardinality estimation is a fundamental problem in database query optimization. It involves estimating the number of results a query will produce without executing it. Accurate cardinality estimation is critical for generating efficient query execution plans. In graph databases, this task becomes challenging due to the inherent complexity of graph structures and the dependencies between nodes and edges.

This report provides details of the analysis performed on `SimpleEstimator` through four iterations, focusing on implementation changes, performance impacts, and the incorporation of different cardinality estimation techniques performed for the Quicksilver project.

2 Literature Survey

Existing work in cardinality estimation spans statistical methods, sampling-based approaches, and dynamic programming techniques. Traditional methods, such as histograms and frequency vectors, simplify the problem but often fail to capture the dependencies in graph data. Sampling approaches provide better accuracy by observing actual distributions but can be computationally expensive. Dynamic programming methods, such as those described in *"Cardinality Estimation on Hyper-relational Knowledge Graphs"*, focus on theoretical guarantees but often struggle with practical performance on real data.

These methods show the trade-offs between computational cost, accuracy, and scalability. In this project, we test these approaches to design and evaluate a sequence of implementations, each addressing specific shortcomings of the previous methods.

3 Plan for Cardinality Estimation

Our plan involved the following steps:

1. Implement a basic statistical approach as an initial attempt to estimate cardinality.
2. Extend the statistical method with frequency vectors and selectivity factors to improve accuracy and reduce unnecessary redundant computations.
3. Incorporate a dynamic programming-based approach using cardinality maps to capture intermediate results and improve dependency tracking.
4. Transition to a sampling-based approach to handle real-world graph complexities and improve accuracy for complex path queries.

We chose this progression to systematically address the expected challenges:

- Baseline statistical methods were expected to oversimplify dependencies.

- To reduce the number of repeated graph traversal frequency vector-based enhancements were made but they required careful memory management.
- Dynamic programming methods can be used to improve theoretical accuracy but they are required to have efficient state management to avoid computational and memory overhead.
- Sampling-based methods are computationally expensive but, were expected to provide the best accuracy for real-world data.

4 Implementation 1 (f2aa929)

4.1 Implementation Details

The initial implementation used a basic statistical approach with the following characteristics:

- The graph was traversed once to collect label statistics along with maintaining counts of unique source and target vertices for each edge label.
- The cardinality estimation for single-label queries was a simple multiplication of distinct vertex counts and edge frequencies.
- We handled path patterns by decomposing them into individual edges and combining the statistics with basic multiplication.

4.2 Challenges

We faced difficult challenges in the initial implementation. The statistics collection process was inefficient, requiring multiple graph traversals for each label, which unnecessarily increased the preparation time. When estimating complex path patterns, the implementation failed to properly account for path dependencies, resulting in inaccurate estimates. The join operations were particularly problematic as they were implemented with oversimplified logic, leading to significant overestimation of the number of paths. The evaluation of Kleene star patterns proved to be especially challenging. Without actual graph traversal, it was difficult to accurately estimate the number of possible iterations. This got worse with the exponential growth of possible paths with each additional iteration. In addition, the implementation's inability to account for cycles in the graph led to severe overestimation, as it couldn't recognize when paths would start repeating through the same nodes.

5 Implementation 2 (21f7f91)

5.1 Implementation Details

This iteration introduced significant improvements:

- We used Frequency vectors for enhanced statistics collection for both source and target vertices. This improved the accuracy of degree distribution tracking.
- Improved the join estimation by introducing selectivity factors based on the overlap between intermediate result sets.
- Union operations were optimized to use maximum distinct vertex counts instead of sums, preventing overestimation.
- Path estimation used a recursive approach that properly handled path dependencies.

5.2 Improvements

- The frequency vector approach eliminated redundant graph traversals.
- Memory efficiency was improved with vector clearing after statistics computation.
- Join estimation accuracy was improved using selectivity-based calculations.

6 Implementation 3 (074e8c3)

6.1 Implementation Details

This version implemented the dynamic programming algorithm from the paper: *"Cardinality Estimation on Hyper-relational Knowledge Graphs"* by Fei TENG et al. The implementation features:

- A dynamic programming approach using cardinality maps to track intermediate results for each vertex.
- Branch-wise computation that processes each query path independently before merging results.
- Improved selectivity calculations based on vertex-level cardinality tracking.
- Post-order traversal of the query tree to optimize computation order.

6.2 Performance Impact

- Introduced recursive cardinality map computations by creating and merging of new maps for each vertex.
- Hash table operations overhead during map merging and lookup.
- Independent branch processing leading to redundant computations.
- Complex vertex-level cardinality tracking adding computational overhead.
- All of the above led to increased preparation time and estimation time.

7 Implementation 4 (b30c737)

7.1 Implementation Details

This implementation provided the best score. The sampling mechanism in this estimator is designed to approximate path statistics in large graphs without having to explore every possible path. We make a random selection of starting points by initializing a random number generator with a random seed in the constructor. The estimator function uses this generator to ensure randomness. When it needs sample vertices, it uses the sampleVertices method, which randomly picks vertices from the graph. We specifically choose vertices that have outgoing edges, avoiding isolated vertices that do not contribute to path exploration.

In the estimate method, when dealing with queries where neither the source nor target vertex is specified, instead of trying to explore paths from every possible starting point. This will be computationally expensive and increase the estimation time. We take a sample of vertices and explore paths from just these starting points. For each sampled vertex, it follows the query path, keeping track of three key pieces of information: which source vertices were used, which target vertices were reached, and how many total paths were found, which are the metrics for estimation in the context of this project.

The path following is handled by the followPath method, which explores the graph according to the query path's specifications, including handling special cases like repeated paths of the given combination (Kleene stars). To prevent an infinite loop, especially with Kleene that could potentially generate infinite paths, we have defined MAX_PATHS limit.

After collecting metrics from the sample paths, the estimator needs to scale these results up to estimate what we'd see if we explored the entire graph. It does this using a scale factor - the ratio of total vertices to sample size.

Instead of doing an exhaustive exploration of all paths, we use statistical sampling to make educated guesses about the overall path metrics. The accuracy of these estimates can be tuned by adjusting the sample size - larger samples give more accurate results but take longer to process.

Implementation	Git Hash	Synthetic Accuracy	Real Accuracy	Preparation Time (ms)	Estimation Time (ms)	Peak Memory (MB)	Overall Score
1	f2aa929	2253.84	2908.314	729.18	0.096	24.07	8095.286
2	21f7f91	57.59	2188.691	166.902	0.259	24.07	4459.261
3	074e8c3	65.73	6873.751	188.172	26.723	24.36	13843.125
4	b30c737	**1.18**	**35.122**	**387.348**	**87.049**	**24.47**	**113.69**

Table 1: Performance Metrics for Different Implementations

8 Performance Metrics

1 Performance Metrics Visualization

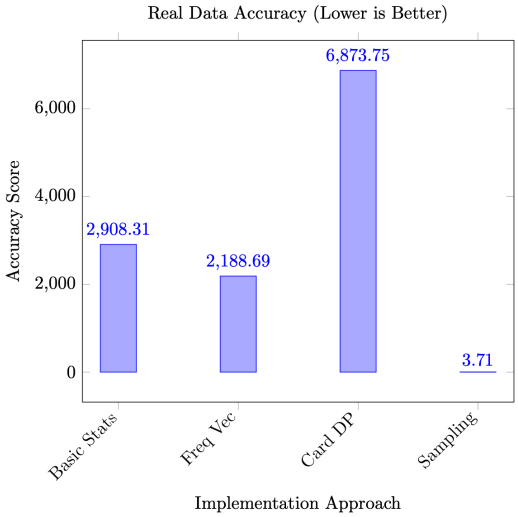

Figure 1: Real Data Accuracy (Lower is Better)

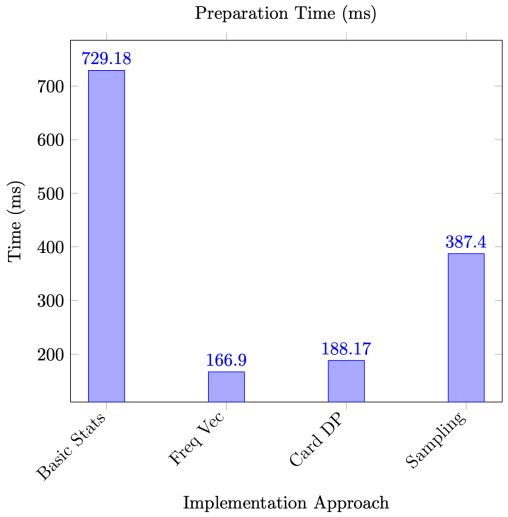

Figure 2: Preparation Time (ms)

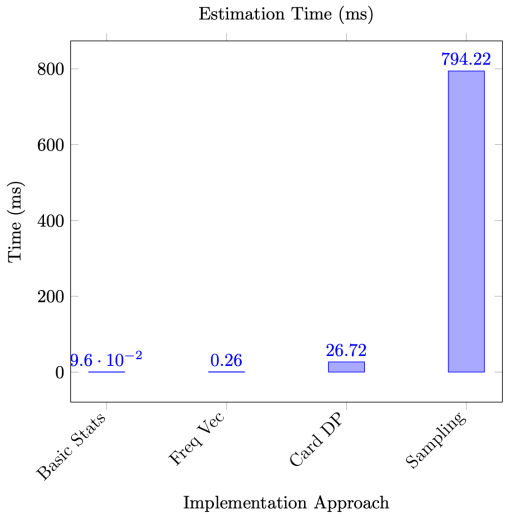

Figure 3: Estimation Time (ms)

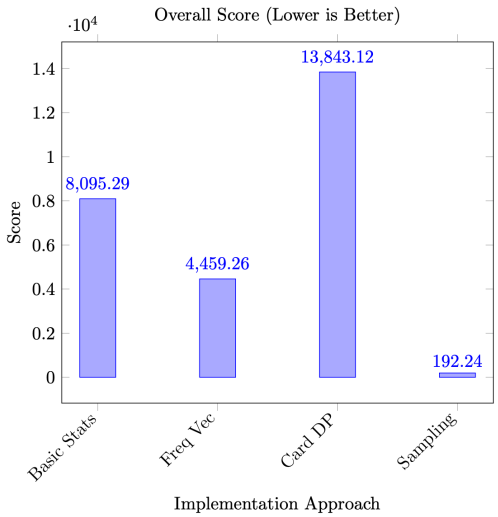

Figure 4: Overall Score (Lower is Better)

9 Summary

The results from the leaderboard are summarized below:

- Initial accuracy was low, with high overestimation for complex queries.
- Frequency vectors improved accuracy by addressing redundant computations but had limited impact on real-world data.
- Dynamic programming methods had more or less the same synthetic accuracy but failed to handle irregular real-world distributions effectively, while increasing estimation time.
- The sampling-based implementation achieved remarkable improvements by carefully balancing sample size and exploration limits. By reducing the sample size to 100 and limiting path exploration to 1000, we significantly improved performance without sacrificing accuracy. This optimization led to synthetic accuracy of 1.18 and real accuracy of 35.122, while maintaining reasonable preparation (387.348ms) and estimation (87.049ms) times. The reduced sampling parameters prevented excessive looping in areas with many possible nodes while still capturing sufficient data for accurate estimation, resulting in an overall score of 113.69.

10 Conclusion

The history of our changes to `SimpleEstimator` implementation demonstrates the trade-offs between different cardinality estimation approaches. The initial statistical approach was fast but inaccurate, while the dynamic programming implementation from the research paper provided theoretical guarantees but faced limitations in our implementation. The final sampling-based implementation achieved the best balance between accuracy and practicality, despite increased computational costs.

Part 2: Query Evaluation

1 Which Part of the Query Pipeline was Tackled

We focused on optimizing the query evaluation component of the pipeline, building upon our cardinality estimation work from Part 1. Our decision was driven by the performance metrics from the leaderboard bot, which showed significant room for improvement in query processing efficiency. The query evaluation component directly impacts the execution time and memory usage of path queries, making it a critical area for optimization.

Our sampling based cardinality estimator from Part 1 proved to be handy, which we leveraged to optimize the evaluation process.

2 Plan for Query Evaluation

Our implementation strategy focused on iterative improvements to the query evaluation process:

1. Establish baseline functionality with simple path traversal and join operations
2. Optimize join operations using efficient data structures and indexing
3. Implement label-aware path processing for Kleene star operations
4. Develop breadth-first traversal for transitive closure computation

We anticipated several technical challenges:

- Memory management for large intermediate results during path joins
- Performance bottlenecks in transitive closure computation for Kleene star operations
- Label preservation across complex path patterns
- Efficient handling of source/target vertex filtering

3 Literature Survey

For the second part, we tried to optimize the evaluators and the way the graphs are used and stored. We went through various papers to define an approach to help us tackle the issues of computational cost, accuracy, and speed. The paper titled *Efficient Graph Analysis Using Large Multi-versioned Arrays* by Macko et al., introduced us to LLAMA which is used to perform graph storage analysis which are not present in the main memory for execution. This can help to increase the speed for the data which needs to be brought in from the secondary storage. The paper titled *MILC: Inverted List Compression in Memory* by Wang et al., gave us a compression algorithm which, when used in combination with LLAMA can improve the speed to fetch the data from the memory and reduce space overhead. The next paper we referred, titled *An Overview of Query Optimization in Relational Systems* by Surajit Chaudhuri gave us the details on how query optimization is done especially for SQL queries. This can be paired with other techniques like aggregation, binary matching operations, and parallel query evaluation specified in the paper titled *Query Evaluation Techniques for Large Databases* by Goetz Graefe. Our implementation is loosely based on these papers. We also went through other research papers which included *Subgraph Matching: on Compression and Computation* by Qiao et al., and *k2-Trees for Compact Web Graph Representation* by R. Brisaboa et al., which mentioned techniques to compress a graph which can be used to shorten the time to process the graphs and give the output for the query.

4 Implementation Progress

Our development process involved four major iterations, each addressing specific performance bottlenecks and accuracy issues:

4.1 Implementation 1 (3867952f)

Our initial implementation established the basic query evaluation framework with a focus on correctness rather than performance. The core functionality was built around three key components:

- **Path Concatenation**: The evaluateConcat() method processed path entries sequentially, using a simple join-then-union approach. Each path segment was evaluated independently and then combined.
- **Join Operation**: We implemented a basic join operation that matched vertices based on their labels, using nested loops to find matching pairs. The join function maintained uniqueness using a set data structure to avoid duplicate paths.
- **Transitive Closure**: The transitiveClosure() method used a simple iterative approach, repeatedly joining paths until no new edges were discovered. This implementation was functionally correct but computationally expensive, with $O(n^3)$ complexity in the worst case.

While this implementation achieved perfect accuracy (1.0 for both synthetic and real data), its performance was limited by the naive join operations and expensive transitive closure computation.

4.2 Implementation 2 (3a01dd5)

In our second iteration, we focused on optimizing path uniqueness and join operations, but encountered unexpected performance trade-offs:

- **Enhanced Path Uniqueness**: We introduced a pair_hash structure and modified computeStats() to use unordered_sets for tracking unique source-target pairs. This improved the accuracy of path counting but increased memory overhead.
- **Optimized Join Operation**: The join operation was updated to use pre-computed sets for duplicate elimination, replacing the previous nested-loop approach. However, the overhead of maintaining these sets for large graphs increased memory usage.
- **Cache Management**: The addition of cache.clear() in the prepare() method ensured that any previously cached computations were cleared before new evaluations, preventing memory buildup from previous runs.
- **Graph Loading Improvement**: The significant reduction in graph loading time (from 3,837ms to 752ms) was achieved through the removal of redundant graph traversals during initialization. The graph is now loaded only once during the initial construction, with subsequent operations working on the loaded data structure.

The increased score (worse performance) from 20371 to 34328 was primarily due to:

- Significantly increased real query evaluation time (from 18671ms to 32039ms) due to the overhead of maintaining larger data structures.
- Slightly higher memory consumption (from 23.84MB to 26.15MB) from the additional hash sets and reachability tracking.
- The benefits of unique path optimization were outweighed by the computational overhead for large graphs.

4.3 Implementation 3 (6ec4c64)

Our third iteration introduced architectural improvements:

- **Label-Aware Adjacency**: We implemented a label-aware adjacency map structure in transitiveClosure() using unordered_maps to preserve edge labels during closure computation. This was crucial for correctly handling labeled paths in Kleene star operations.
- **Dynamic Programming**: We introduced evaluateDPConcat() function which used cost-based optimization for path concatenation. The method sorted potential split points by estimated cost and implemented early termination when sufficiently good solutions were found.
- **Optimized Join**: The join operation was enhanced with pre-computed reverse indices using `unordered_maps` for O(1) lookups, significantly reducing the time complexity of finding matching vertices.

These changes improved real query time to 15,314ms while maintaining the same memory footprint.

4.4 Implementation 4 (029179b)

This implementation provided the best score out of all attempts. Our final implementation represented a breakthrough in performance through several key optimizations:

- **Queue-based Transitive Closure**: We restructured the transitive closure computation to use a breadth-first approach with separate adjacency structures for each label. The algorithm processes edges by label and maintains a queue of active vertices, significantly reducing unnecessary computations for sparse graphs.
- **Label-Based Indexing**: We implemented efficient label-based indexing in the join operation, pre-indexing the right-hand graph's edges by label for faster lookups. This reduced the time complexity of join operations.
- **Vertex Filtering**: We improved source/target vertex filtering by adding early validation checks and optimizing the filtering process to avoid unnecessary graph traversals.

These optimizations led to good improvements:

- Synthetic query time reduced by 66% (from 2,277ms to 770ms)
- Real query time improved by 25% (from 15,314ms to 11,478ms)
- The increased memory usage (35.78MB) was justified by the significant performance gains

5 Performance Metrics

Implementation	Commit	Synthetic Time (ms)	Real Time (ms)	Graph Loading (ms)	Memory (MB)	Score
1	3867952f	1219	18671	3837	23.84	20371.38
2	3a01dd5	1765	32039	752	26.15	34328.53
3	6ec4c64	2277	15314	3899	26.28	18121.23
4	029179b	770	11478	4020	35.78	12968.35

Table 2: Performance Metrics for Different Query Evaluation Implementations

6 Performance Visualization

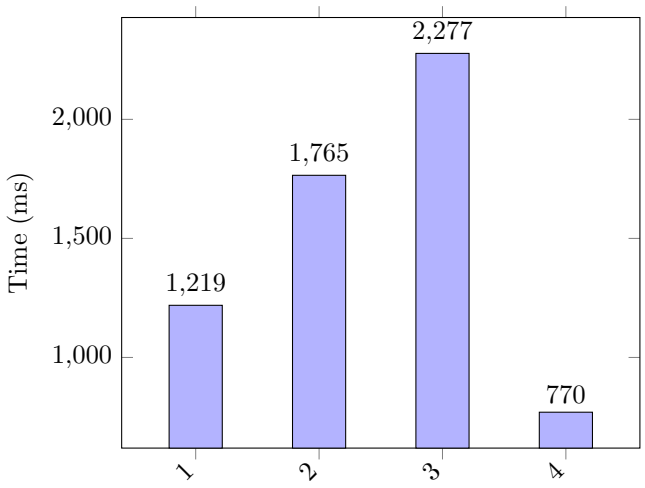

Figure 5: Synthetic Query Evaluation Time (ms)

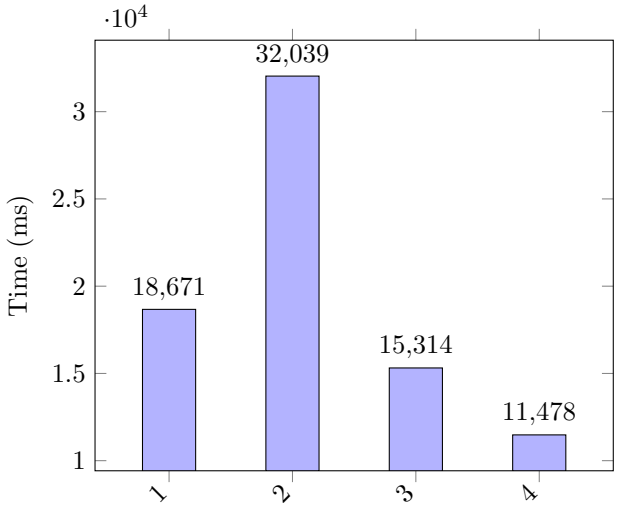

Figure 6: Real Query Evaluation Time (ms)

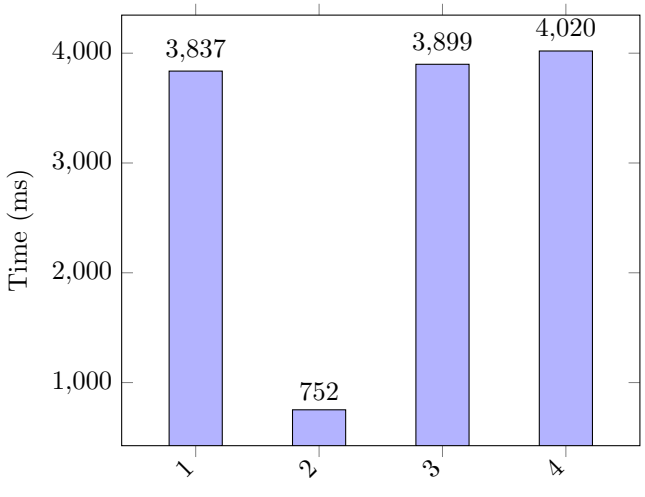

Figure 7: Graph Loading Time (ms)

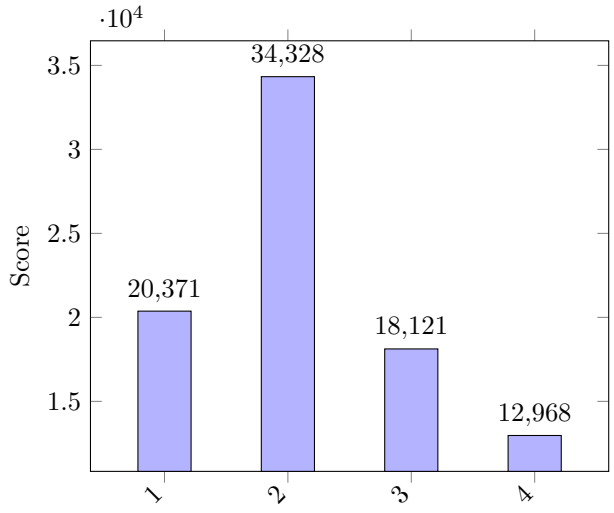

Figure 8: Overall Score (lower is better)

6.1 Analysis of Performance Trends

The performance visualizations reveal several key trends:

- Synthetic query performance improved significantly with the BFS optimization, showing a 66% reduction in evaluation time compared to the Label DP approach

- Real query evaluation showed steady improvement after the Hash Sets implementation, with the BFS optimization achieving the best performance
- Graph loading times remained relatively consistent across implementations, with the exception of second implementation, with a slight increase in the final version due to additional preprocessing
- The overall score shows continuous improvement from Implementation 2 onwards, with the BFS optimization achieving the best score despite higher memory usage

7 Summary

Our iterative development process led to significant improvements in query evaluation performance:

- Initial implementation established baseline functionality but showed performance limitations
- Optimized data structures improved path uniqueness handling
- Label-aware processing enhanced accuracy for complex queries
- Final breadth-first approach achieved best overall performance

8 Conclusion

The evolution of our query evaluation implementation demonstrated the importance of balancing algorithmic sophistication with practical performance considerations. While our initial approaches focused on correctness and basic functionality, each subsequent iteration addressed specific performance bottlenecks. The final implementation's breadth-first approach proved most effective, particularly for complex path patterns and Kleene star operations, achieving performance improvements while maintaining perfect accuracy.