2IMD10: quickSilver Group 11

Sviatoslav Gladkykh and **Ivan Georgiev** and **Matteo G. Giordano** and **Nam Nguyen**

Slack leaderboard IDs

- namiknguyen1234
- m.g.giordano
- gladkykh.sviatoslav
- i.georgiev

Abstract

This report presents our work on optimizing cardinality estimation and query evaluation for the Quicksilver graph database. We explored various estimation techniques, including histograms, HyperLogLog (HLL), and sampling, with the Histogram v2 method proving the most effective, achieving a leaderboard score of 102.41 by balancing accuracy and efficiency.

For query evaluation, we introduced BFS-based transitive closure (TC), dynamic TC switching, and optimized join ordering, significantly improving execution speed. A custom caching system eliminated redundant computations, , while graph representation optimizations reduced peak memory usage.

Our final solution achieved a score of 2649.855, placing us in the top-10. Future work could further enhance performance through parallel execution, refined query planning, and more efficient graph representations.

Part 1

In the following sections we will be focusing on the first part of the project, which is the cardinality estimator.

1. Introduction

A database is a system that consists of interconnected data and the programs that process it. Databases have become an essential tool across many domains, with the primary goal of efficiently retrieving stored information. To achieve this, developers focus heavily on query processing, which involves several key stages, including query parsing, translation, optimization, and evaluation. Among these, cardinality estimation plays a critical role in optimizing query performance.

In the context of databases, cardinality refers to the number of distinct values or records within a column or combination of columns. When a query optimizer assesses the cost

of executing a query plan, cardinality serves as a crucial factor influencing its decisions. Cardinality estimation involves approximating the size of the result set that a query will return. While cardinality estimation techniques have been extensively studied for relational databases, many approaches face challenges related to accuracy, speed of inference, or storage efficiency.

For instance, histograms are one of the traditional techniques for cardinality estimation and offer valuable insights into data distribution. They achieve high computational efficiency by leveraging pre-computed statistical information, which is based on counting attribute occurrences within predefined ranges. However, this efficiency comes at the cost of additional memory usage, which can become substantial for large datasets requiring efficient histograms. Constructing such histograms also involves data sampling, which consumes time and introduces a trade-off between precision and speed. Alternatively, methods that rely on estimation formulas based on uniformity assumptions can reduce memory requirements and computational overhead, but this often comes at the expense of estimation accuracy.

The Quicksilver project is centered around a graph database, where the data is organized as triples: the first and last elements represent source and target nodes, while the middle element indicates the edge label. Compared to relational databases, graph databases pose additional challenges for cardinality estimation due to their nested and directed structure. Highly clustered nodes create dense networks, with connections often relying on join operations, further increasing the complexity of cardinality estimation. Additionally, collecting statistics in graph databases is more complicated because they lack a fixed schema. Instead of scanning columns or rows in a structured table, graph statistics must be derived through path traversal across nodes connected by various labels.

This traversal-based approach can hinder sampling efforts, as errors accumulate and propagate through consecutive join operations, ultimately degrading estimation accuracy. Moreover, the directional nature of edges presents another difficulty, particularly in queries involving alternating label directions. This report outlines the strategies and methods employed to address these challenges in the context of the Quicksilver project.

2. Literature discussion

In order to grasp the scope of the problem, we conduct a literature review of contemporary solutions to the problem. The problem of cardinality estimation in DBMS has existed for a long time. For example, Poosala and Ioannidis (1997) investigated an approach using multi-dimensional histograms. This approach provides a simple, yet efficient method for capturing data dependencies.

Other approaches (Galindo-Legaria et al. (2003)) feature methods based on the creation and utilization of statistics on views in SQL databases, providing the optimizer with statistical information on the result of relational queries. It opens a new dimension on the data available for cardinality estimation and enables arbitrary correction.

In another approach, Getoor et al. (2001) extends Bayesian Networks to the domain of relational databases. This allows for the modeling of joint probability distributions across multiple attributes and relations by leveraging conditional independence, enabling accurate selectivity estimation. Another example of a probabilistic approach is Flajolet et al. (2007),

which employs a hashing-based approach to map elements to binary values and observes the position of the leftmost 1-bit to infer the cardinality.

While the previous approaches have introduced methods grounded on attributes of the data, others work by instead working with a subset of data. An example of this is an approach shown by Gibbons (2001), Distinct Sampling builds a tailored sample during a single scan of the data. This sample captures a uniform representation of distinct values along with associated counts and optionally includes a subsample of rows for each value.

The majority of these methods focus on relational databases, as it is the most commonly used database type. As such, we also wanted to examine methods that operate on other types. Zhang et al. (2006) proposes a cardinality estimation method for XML data. Restructuring the recursive XML search tree as a graph allows for fast and memory-efficient initial cardinality estimates.

Lastly, van Leeuwen et al. (2022) propose an approach that specifically deals with graph databases. This approach provides a flexible framework that allows the combination of multiple cardinality estimation techniques with the aim of comparing and producing a superior solution.

3. Implementation plan

At the start of the project, our team begins by individually reviewing the assignment, exploring the provided code, reading supplementary materials (as outlined in Section 2), and researching various methods for cardinality estimation. This process helps us develop a solid understanding and intuition about the nature of the task.

We then come together to decide on four key methods to focus on: histogram-based, HyperLogLog, sampling, and sketch-based algorithms. Each team member takes responsibility for implementing specific methods, ensuring the workload is divided efficiently.

During the evaluation phase, we expect some methods to demonstrate greater accuracy than others, though challenges such as managing uneven data distributions or optimizing memory usage may arise. To address these issues, we refine the methods further by testing alternative data structures and fine-tuning parameters.

In the final phase, we combine elements from different methods to create a hybrid approach that balances accuracy and efficiency. While we anticipate improvements, we are prepared to tackle challenges such as computational inefficiencies or unexpected behaviors in the combined methods through iterative testing and optimization.

Further details on our methods, results, and challenges are discussed in the following sections.

4. Baseline histogram-based approach

As for the first "baseline" solution our team implemented a simple and straightforward algorithm by combining histograms, frequency data and the adjacency lists of the graph itself.

4.1 Preparation

In the preparation method of the SimpleEstimator class we precomputed the following fields:

- Label Frequencies – a vector in which an element at position i indicated the total number of edges labeled i.

- In/out degree histograms – a vector in which an element at position i indicated the number of vertices with in/out degree i.

- Average in/out degree of the vertex in a whole graph – two additional floating point fields.

4.2 Estimation

Our estimation algorithm was implemented in a similar way to the query evaluator. Specifically, we start from the simplest possible queries (just label with direction) and then combine them to get an estimate for possibly more complex requested query.

Here is more detailed breakdown into the algorithm we implemented:

1. Label with direction

   - If both source and target are not specified in a query, then we use the following estimate:

$$\begin{aligned}
\text{noOut} &= \sum_{i=1}^{\text{maxOutDegree}} \text{outDegreeHistogram}[i] \cdot P(\exists v \in V(G) \exists e \in E(G) : deg(v) = i, e = (v, l, *)) \\
&= \sum_{i=1}^{\text{maxOutDegree}} \text{outDegreeHistogram}[i] \cdot i \cdot P(\exists e : e = (*, l, *)) \\
&= \sum_{i=1}^{\text{maxOutDegree}} \text{outDegreeHistogram}[i] \cdot i \cdot \frac{\text{labelFreq}[l]}{\sum_{j=1}^{noLabels} \text{labelFreq}[j]}
\end{aligned}$$

$$\text{noPaths} = \text{labelFreq}[l]$$

$$\text{noIn} = \sum_{i=1}^{\text{maxInDegree}} \text{inDegreeHistogram}[i] \cdot i \cdot \frac{\text{labelFreq}[l]}{\sum_{j=1}^{noLabels} \text{labelFreq}[j]}$$

     In case of the reverse label direction we swap estimates for noOut and noIn.

   - If source is bound and label has forward direction OR target is bound and label has backward direction we compute noOut, noPaths and noIn exactly:

$$\text{noOut} = \begin{cases} 0 & \text{if noPaths} = 0, \\ 1 & \text{otherwise.} \end{cases}$$

$$\text{noPaths} = |\{e : e \in E(G), e = (src, l, *)\}|$$

$$\text{noIn} = \text{noPaths}$$

- If target is bound and label has forward direction OR source is bound and label has backward direction we also compute noOut, noPaths and noIn exactly:

$$\text{noOut} = \text{noPaths}$$

$$\text{noPaths} = |\{e : e \in E(G), e = (*, l, trg)\}|$$

$$\text{noIn} = \begin{cases} 0 & \text{if noPaths} = 0, \\ 1 & \text{otherwise.} \end{cases}$$

- If both source and target are bound we can easily check the existence of such edge:

$$\text{noOut} = \begin{cases} 1 & \text{if } \exists e \in E(G) : e = (src, l, trg) \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{noPaths} = \text{noOut}$$

$$\text{noIn} = \text{noOut}$$

2. Union of labels with direction

- To compute the union of simpler structures as labels with directions we can assume that paths generated by these structures are disjoint as well as input and output nodes. This (not always good) assumption allows us to estimate the result of the union in the following way:

$$\text{noOut} = \begin{cases} \text{noOut1} + \text{noOut2} & \text{if src is not bound} \\ 1 & \text{if src is bound and noPaths } > 0. \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{noPaths} = \text{noPaths1} + \text{noPaths2}$$

$$\text{noIn} = \begin{cases} \text{noIn1} + \text{noIn2} & \text{if trg is not bound} \\ 1 & \text{if trg is bound and noPaths } > 0. \\ 0 & \text{otherwise.} \end{cases}$$

3. Kleene's star of the union of labels with direction

   - Computing Kleene's start operation is quite challenging. Our estimate for paths is based on convergent geometric series which is quite weak since it does not use any information about the graph itself, we keep noOut estimate fixed and multiply noIn by average output degree of the graph:

$$\text{noOut} = \text{noOut1}$$

$$\text{noPaths} = \text{noPaths1} \cdot \frac{1}{1-q}$$

$$\text{noIn} = \text{noIn1} \cdot \text{avgOutDeg}$$

4. Path concatenation of Kleene's stars of the unions of labels with direction

   - For computing estimates of the noPaths after path concatenation we derived a formula in a similar way to the join for the tabular data and optimized for graph data. Thus, achieving the following estimates:

$$\text{noOut} = \text{noOut1}$$

$$\text{noPaths} = \min\{\frac{\text{noPaths1} \cdot \text{noPaths2}}{\min\{\text{noIn1}, \text{noIn2}\}}, \frac{\text{noPaths1} \cdot \text{noPaths2}}{\min\{\text{noOut1}, \text{noOut2}\}}\}$$

$$\text{noIn} = \text{noIn2}$$

4.3 Results

Although this approach makes several unrealistic assumptions and uses weak methods for estimating cardinality, it still achieves relatively high accuracy on real data (around 25), though lower accuracy on synthetic data (approximately 164). A positive aspect of this algorithm is that it requires little preparation time and helps avoid worst-case query plans. All in all, resulting in final score of 241.

5. HyperLogLog-Based Approach

This section explains the second approach our team implemented. HyperLogLog (HLL) is a sketch-based approach estimator in the context of data summarization and approximate computations. The estimator includes a probabilistic counting to approximate the number of distinct elements in the graph queries.

5.1 Mathematical Foundations

The estimator uses the *HyperLogLog (HLL)* algorithm to approximate the number of distinct values. HLL uses hashing and leading-zero counts to estimate cardinalities. The main formula for the raw estimate is:

$$\text{Estimate} = \alpha \cdot m^2 \cdot \left(\sum_{i=1}^{m} 2^{-b_i} \right)^{-1}$$

where:

- m is the number of buckets (1024 in our implementation),
- b_i is the value in the i-th bucket, representing the maximum number of leading zeros seen in that bucket,
- α is a bias correction factor dependent on m ($\alpha = 0.7213/(1 + 1.079/m)$).

5.2 Preparation Phase

The `prepare` method precomputes statistics and structures to help with the estimation of the queries. This includes initializing the HyperLogLog buckets and computing degree histograms for the graph.

1. **HyperLogLog Buckets:**
    - For each edge in the graph, a unique hash is computed using the source node, label, and target node. This ensures that each edge contributes uniquely to the HLL buckets.
    - The bucket index is determined by the lower-order bits of the hash value:

$$\text{BucketIndex} = \text{Hash}(v, l, t) \mod m$$

- The number of leading zeros in the remaining bits is computed to update the bucket:

$$b_i = \max(b_i, \text{LeadingZeros}(\text{Hash}(v, l, t)))$$

This value represents the maximum trailing zero pattern observed for a bucket.

2. **Degree Statistics:**

- The out-degree and in-degree of each vertex are used to build histograms:

$$\text{OutDegreeHistogram}[d] = \text{Count of vertices with out-degree } d$$

$$\text{InDegreeHistogram}[d] = \text{Count of vertices with in-degree } d$$

These histograms help with cardinality estimation when nodes are bounded.

3. **Label Frequencies:**

- The frequency of each label l is precomputed to estimate the number of paths associated with the label:

$$\text{LabelFreq}[l] = \sum_{(v,t) \in E(G)} \delta(e = (v, l, t))$$

where δ is an indicator function that evaluates to 1 if the edge matches the label.

5.3 Estimation

Similarly to the previous approach - we start from the simplest possible queries and then combine them to get an estimate for possibly more complex requested queries.

5.4 Label with direction

5.4.1 Free Source and Target

When both the source and target nodes are free, the cardinality estimation is based on label frequencies and the HLL buckets:

$$\text{noPaths} = \text{LabelFreq}[l]$$

$$\text{noOut} = \text{HLL Estimate for distinct source nodes}$$

$$\text{noIn} = \text{HLL Estimate for distinct target nodes}$$

Here, the HLL provides approximate counts for distinct nodes, avoiding the need for explicit enumeration.

5.4.2 Bounded Source or Target

For queries where the source node is fixed (src $\neq *$) and the target is free, the estimator performs exact counting using the graph's adjacency list:

$$\text{noPaths} = |\{e \in E(G) \mid e = (\text{src}, l, *)\}|$$

$$\text{noOut} = \text{Distinct outgoing edges from src}$$

$$\text{noIn} = 1$$

Similarly, for a fixed target (trg $\neq *$) and free source:

$$\text{noPaths} = |\{e \in E(G) \mid e = (*, l, \text{trg})\}|$$

$$\text{noIn} = \text{Distinct incoming edges to trg}$$

$$\text{noOut} = 1$$

5.5 Union of Labels

For unions of labels $(l_1, l_2, \ldots, l_k)$, the estimator aggregates paths across all labels:

$$\text{noPaths} = \sum_{i=1}^{k} \text{LabelFreq}[l_i]$$

Distinct source and target nodes are estimated using the HLL:

$$\text{noOut} = \text{HLL Estimate for distinct sources across all labels}$$

$$\text{noIn} = \text{HLL Estimate for distinct targets across all labels}$$

5.6 Kleene Star

The Kleene star operator represents repeated path traversal. We approximate this, using a geometric decay factor:

$$\text{noPaths} = \sum_{i=1}^{\text{iterations}} \text{noPaths}_{i-1} \cdot \text{discountFactor}$$

The discount factor reduces the contribution of each additional repetition, ensuring that the estimate converges.

5.7 Concatenation

For concatenation of two paths (P_1 and P_2), the estimator combines their cardinalities while accounting for overlaps:

$$\text{noPaths} = \frac{\text{noPaths}_1 \cdot \text{noPaths}_2}{\max(\text{noIn}_1, \text{noOut}_2)}$$

$$\text{noOut} = \max(\text{noOut}_1, \text{noOut}_2)$$

$$\text{noIn} = \text{noIn}_2$$

5.8 Results

Even though the HLL approach seemed to have high potential while researching the different methods - we reached the conclusion that the other approaches have better performance based on the different query operations which with some of them HLL struggles. While the synthetic data result was decent (95), the real data result was quite poor (1476) so we decided to put more focus on the other approaches. Even with rather low preparation and estimation time, the final score (3080) was not good enough for us.

6. Histogram v2

In the final phase of the project we decided to focus our efforts into improving the approach with the most promise, which was the baseline histogram approach. We tried to improve on the algorithm's weaknesses, which appeared to be the synthetic workload. Upon further inspection we found out that for bounded queries, the algorithm always produced 0 paths. Upon closer inspection, we found that this was due to the computation of noPaths only considering direct paths. This fails to take into consideration the combination of labels. We therefore decided to collect more statistics to address this problem.

6.1 Preparation

In addition to the fields from the baseline Histogram approach, we collect additional statistics:

- NodeIn – Stores the in degree of node i
- NodeOut – Stores the out degree of node i

6.2 Estimation

The estimation phase is identical to the baseline algorithm, except for constructing the estimates of the bound source/bound target. Using our now pre-computed statistics of node in/out paths, we are able to better estimate noPaths for single labels with direction:

- If source is bound and label has forward direction OR target is bound and label has backward direction:

$$\text{noPaths} = \text{NodeOut}_i * \text{average out degree}$$

- If target is bound and label has forward direction OR source is bound and label has backward direction:

$$\text{noPaths} = \text{NodeIn}_i * \text{average in degree}$$

This addresses our previous issue with only finding direct paths. By using a precomputed statistic for the node degree, we are able to leverage the characteristics of the actual data. Multiplying by the average degree scales up the estimate for possible future paths, giving

a reasonable estimate. We also change the aggregation method for path concatenation, opting for a simpler formula:

$$\text{noPaths} = \text{sum}\{\text{noPaths1}, \text{noPaths2}\}$$

The aggregation formula for when both variables are free remains the same

6.3 Results

Using this refined approach, we were able to achieve significantly better results, with an accuracy score of 23.97 for the synthetic workload, and 26.54 for the real workload. This gave us an overall score of 102.41. What is note worthy about this solution is that, compared to other entries dominating the top of the leaderboard, our solution requires significantly less preprocessing time. This makes it computationally light, as well as decently performative.

7. Sampling

In the initial phase of the project, our team decided to implement sampling as a safer backup option to have a working solution, although not so efficient, if the others had problems. The basic idea is to actually execute the query on a smaller, sampled graph, evaluate it, and scale up the results so they are meaningful for the original graph.
Such sampled graph is constructed choosing random edges from the original (with replacement) with a uniform distribution. We also tried sampling nodes instead of edges, with worse results.

7.1 Preparation Phase and Sampling

The main aim of the preparation phase is to create the sampled graph we will use to evaluate the query on.
The sampled graph has a pre-established ratio between its size and the original graph's: in our implementation, the former has **30%** of the latter's number of edges. This ratio has proven to be a good compromise between speeding up the evaluation time and not losing too much accuracy.

The `prepare` function calls `edgeSample`, which returns the actual sampled graph, according to the following:

1. A new empty graph is initialized with the same number of vertices and labels as the original.

2. The `edgeOffsets` vector, required for `accessEdge`, is computed. This contains the starting position for each vertex's adjacency list if we were to flatten all of them into a single array.

3. Until the sampled graph is 30% in size compared to the original, a random index is generated, then used to access the respective edge (using the `accessEdge` function) and add it to the sampled graph.

7.1.1 accessEdge

This function provides a way to directly access a specific edge in the original graph by its index, assuming a flattened version of its adjacency list representation. Since this function is called $(0.3)E$ times, it was imperative to have it run as efficiently as possible.
Its inputs are the graph and the global (randomly generated) index of the edge we want to access. Through a binary search, the smallest value in `edgeOffsets` greater than the index is found: this determines which vertex (`vertex`)'s adjacency list contains the edge at the given global index. Then `localIndex = index - edgeOffsets[vertex]` gives the position of the edge in that vertex's adjacency list, thus enabling us to access `graph->adj[vertex][localIndex]` to add it in the sampled graph. This approach entails that the time to construct the sampled graph is $O((0.3)E \cdot logN)$, but we save on memory that would be required if we were to actually copy the original graph into a flattened structure.

7.2 Estimation

The first step in the `estimate` function is to check whether one between the source and the target of the query are specific identifiers, and whether the query is complex enough (path size > 2). Given the nature of sampling as an estimation method, queries that are this specific usually are not estimated properly because they entirely depend on the random chance that their required solution still exists in the sampled graph. Thus, instead of wasting estimation time on this kind of queries, we hardcoded estimates in the following fashion:

$$q.src \neq NO\_ID \implies noOut = 1, noIn = noPaths = 120$$

$$q.dst \neq NO\_ID \implies noIn = 1, noOut = noPaths = 120$$

where 120 is a value that empirically produced the least amount of accuracy loss in the experiments.

For all other kinds of query, the approach is simply running it on the sampled graph using the already provided evaluator infrastructure in order to get its result's cardinalities. Of course, this carries a sensible overhead that is inherent to the sampling estimation method. After obtaining our evaluation, we scale up the cardinalities by multiplying them by the inverse of the size ratio between the original and sampled graphs. In our implementation, that is $\frac{1}{0.3}$.

7.2.1 Results with naive approach

This simple approach clearly did not have the best performance among the ones we tried; scores we got were around the ~ 730 mark, with more and less lucky instances depending on the random sampling process.

As we can see from table 1, the main problem with this solution is the high estimation time caused by the need to actually run the query and evaluate its results. Accuracy for real data is unsurprisingly not the best, while the accuracy for the synthetic counterpart is significantly better, mostly because there are more specific queries where the hardcoded values provide a good enough estimate.

7.3 Combined approach

Given the best results provided by the improved histogram approach, we thought it would be interesting to use it in combination with sampling to check whether that leads to a improvement in the scores. The idea is to first sample the graph as previously described, then, instead of running the query, estimating it via histogram method.
The `prepare` method therefore is the same as the one discussed in the histogram approach, with the addition of the creation of the sampled graph. `estimate` is the same as in the sampling approach, the only difference being that instead of instantiating an evaluator and running the query, we estimate the result.

7.3.1 Results

Results for this combined approach are significantly better than the plain sampling. The most significant improvement is the estimation time, which drops to the same levels as the improved histograms plain approach. Preparation time is higher due to the merging of two different methods and their respective preparation steps. Synthetic and real accuracy respectively slightly worsen and improve compared to the histogram v2 approach due to sampling having better performance on the former and worse on the latter. Overall, the way sampling and histogram-based estimation are combined causes a overhead, especially for what concerns the estimation time, which results in a slightly worse performance than the improved histogram approach alone.
Future developments could aim at reducing such preparation time by trading off memory usage, and carrying out the 2 preparation phases required for sampling and histograms independently and in parallel.

8. Discussion

approach	acc_syn	acc_real	prep_time	est_time	peak_memory	score
Base Histogram	164.98	25.48	77.71	5.7	24.07	241.22
Hyper-LogLog	95.75	1476.65	219.50	35.40	24.07	3080.42
Basic sampling	8.06	91.04	362.38	2600.96	24.84	735.53
Histogram v2	23.97	26.54	99.04	5.61	24.14	102.41
Sampl. + Hist.v2	8.81	37	351.05	5.82	24.84	109.16

Table 1: Results for all of our approaches.

In our implementation plan, the first phase consisted of working on multiple implementations in parallel. This way, we were able to explore a sufficiently wide array of solutions and determine which one worked best for our use case.

Ultimately, we have decided on four methods to focus on, each of which was based on the provided literature overview. Given that a majority of the literature concerned relational databases instead of graph databases, we used our own intuition and experience to decide which approaches would be feasible to implement. Out of the four, we have successfully implemented three approaches: Histogram-based, Hyper-LogLog, and Sampling.

The sketch-based approach was discarded due to difficulty with implementation, as well as overall similarity with the Hyper-LogLog approach.

During our evaluation phase, we decided to focus on the Histogram approach, addressing the weak points in our implementation in hopes of providing a better estimate. We noticed that the main issue of our implementation was low accuracy for synthetic workloads. After focusing our attention there, were able to address these weak points, giving rise to our histogram v2 estimator. which allowed us to obtain the highest score of 102.41.

In the final phase, we made an attempt to combine elements from different methods in an attempt to create a superior solution. We combined our sampling and histogram v2 solutions into a single estimator. This did achieve an increase in accuracy for the synthetic workload, but at the cost of lower accuracy in the real workload. In the end, our combined approach achieved a decent score, albeit not better than our previous best.

Overall, we can say that we have had considerable success in implementing a cardinality estimator for the quickSilver graph database. We have investigated a wide array of different approaches with varying degrees of success. We have also conducted a thorough investigation of promising approaches, being able to achieve incremental improvements. The leaderboard scores of all implementations are shown together in Table 1.

Part 2

In the following sections, we will focus on our implementation of an efficient query evaluator.

9. Introduction

Efficient query evaluation is a fundamental challenge in database systems, particularly for graph databases where queries often involve complex operations like transitive closure and path traversal. These computations can be computationally expensive, especially as graph size and query complexity increase.

In Quicksilver, the default query evaluation system faces inefficiencies due to the high cost of transitive closure operations, redundant computations for repeated subqueries, and limited flexibility in query planning. These limitations result in slower execution times and higher resource consumption, particularly for large and intricate queries.

Part 2 focuses on addressing these challenges by improving key aspects of the query evaluation pipeline, including transitive closure, caching mechanisms, and query planning.

10. Strategy

In order to improve Quicksilver and our leaderboard score, our team focused on the following components of the query pipeline:

- Database engine: We want our implementation to be **Quick**, so we decided to improve the transitive closure and implement a caching system. We chose these kinds of optimizations as priorities since they benefit all queries, regardless of their specific structure, and evaluation time plays a big role in the final score.

- Parser and query planner: Quicksilver needs to be **Smart**, so we need a more flexible representation of queries in order to be able to fully leverage our Estimator and allow for choosing the order of operations that result in better query plans.

- Lastly, database (data representation): Improving the representation of both the graph and the cached intermediate results aims to make the project **Frugal** by optimizing memory usage. We decided to focus more on the caching aspect and leave some refinements for the later stages of the project.

11. Literature Overview

In order to understand the scope of possible solutions, we decided to look at an overview of the available techniques. This paper by Graefe (1993) gives an overview of the foundation of query evaluation systems. Despite coming from 1993, it still provided great inspiration to our implementation.

After analyzing the given default implementation, we immediately notice that the Transitive Closure (TC) algorithm is very inefficient. This is especially due to it also relying on the join and union implementations. Our focus on improving this implementation leads us to Demetrescu and Italiano (2006). Based on this, we implement our BFS TC implementation, which has much better theoretical guarantees. We also explore alternative versions of TC, which may work better in certain scenarios (Hougardy (2010)).

Another thing we notice is that in the query evaluation process, certain subgraphs may be computed many times. This would naturally lead to a large overhead, especially for long queries. We therefore look into approaches that would alleviate this issue. Drawing inspiration from Wang et al. (2017), we implement a modified caching system that allows us to reuse components that have already been computed, removing the overhead from recomputation.

12. Implementation Plan

At the start of the project, our team reviewed the provided codebase and supplementary materials to understand the challenges in query evaluation. This phase involved exploring different techniques for optimizing transitive closure computation, caching, and query planning.

We identified three key areas to focus on:

- **Transitive Closure Optimization:** Since transitive closure is a critical operation, we planned to evaluate and implement multiple algorithms, including breadth-first search (BFS)-based and iterative methods. Our goal was to improve efficiency across both shallow and deep graph queries.

- **Caching Mechanism:** To avoid redundant computations, we aimed to develop a flexible caching system that would store intermediate results. This would reduce query evaluation time without significantly increasing memory usage.

- **Query Planner Enhancements:** We planned to refine the query parsing and planning stages, enabling more efficient operation ordering and better integration with our cardinality estimator.

Our approach involved dividing these tasks among team members to ensure parallel progress. Each team member took responsibility for researching and implementing specific components. During development, we anticipated challenges such as balancing memory usage in caching and ensuring correctness across diverse query types.

In the final phase, we planned to integrate these components into a cohesive query evaluation pipeline, iteratively testing and optimizing for both synthetic and real workloads. This structured plan allowed us to systematically address the key bottlenecks in query evaluation.

Towards the end of the project, when the major implementations are done, we would also focus on minor improvements that would allow us to squeeze out maximum performance from the leaderboard.

13. Implementation

13.1 Caching

One of the key improvements implemented to speed up query execution is caching query results. In our assignment, the graph remains fixed and does not change over time. Additionally, during benchmarking, a fixed set of queries is executed. Given these conditions, we devised the following approach:

We utilize a hash table to store all intermediate results during query execution. At each step, we first attempt to retrieve the result from the cache. If the result exists in the cache, we use it directly; if not, we evaluate the query segment, store the new result in the cache, and then proceed.

More concretely, we store various components of the query along with their corresponding resulting graphs. Specifically, the cache holds:

- The query result with both endpoints (Triple).
- The query result with unbound endpoints (a vector of PathEntry elements).
- The resulting graph of each PathEntry (both with and without Kleene star).
- The resulting graph of each LabelDir.

To make all the above-mentioned objects hashable, we implemented a separate class with query extensions. This class consists of static methods that convert the objects in focus into strings. These strings are then used as keys in our hash table.

After implementing the baseline approach, we turned our focus to optimizing caching. Initially, we observed that storing the full query with both endpoints did not significantly speed up evaluation but did consume substantial memory. This was due to the fact that selecting source and target from an unbound query is a relatively fast operation. Storing both the bound and unbound versions essentially doubled memory consumption in the worst case without a corresponding performance benefit.

Consequently, we removed the storage of the full query with both endpoints. This change alone improved our leaderboard score by approximately 150 points.

After that, we investigated whether other caching components offered a worthwhile execution speed-up relative to their additional memory cost. It turned out that these components were crucial for performance, so we maintained them as they were.

Uncertain about the number of queries that would be used to test our evaluator, we considered further reducing memory consumption for scenarios involving many queries. We contemplated setting a maximum cache size and removing older results when approaching that limit (similar to a queue mechanism). However, this strategy ultimately proved unnecessary for our task.

To conclude, we achieved an improvement of more than 1500 leaderboard points solely due to caching, which is an excellent result. Unfortunately, more advanced optimization techniques were unnecessary for this assignment because of the fixed graph and the small number of queries used in benchmarking.

In a different scenario, strategies like limiting cache size, caching execution plans instead of entire query results, developing more sophisticated logic for prioritizing certain components in the cache, and clearing cache entries based on the lowest priority would make more sense. However, for this task, the simpler version proved to be the best-performing approach.

13.2 Transitive Closure (TC) Improvements

Looking at the performance of the queries, we quickly noticed that the largest overhead comes from the transitive closure computation. We therefore focused a lot of our effort into improving this component.

13.2.1 SMART TC

At first, our team attempted to implement the SMART Transitive Closure (TC) Evaluation from the slides. This approach aims to "batch" new edges in each iteration, effectively doubling the path length captured so far. The idea is that if $\delta R*$ represents all newly discovered edges of path length k, then in the next iteration we can uncover edges of path length $2k$ by chaining $\delta R*$ with itself, and so on.

While it yielded promising performance results in some scenarios, our particular implementation did not fully handle certain deeper queries, leading to suboptimal overall efficiency. We invested significant effort trying to address edge cases with longer paths, but ultimately shifted our focus to a different approaches described in later sections.

Nevertheless, the core SMART Evaluation algorithm itself is outlined below in 1. It highlights how $\delta R*$ (the "delta" set of newly discovered edges) evolves, and how the closure set $R*$ is incrementally expanded until no further edges can be found.

Algorithm 1 SMART Transitive Closure Evaluation

Require: A binary relation $R \subseteq V \times V$
Ensure: R^*, the transitive closure of R

$R^* \leftarrow \{(x, x) \mid \exists\, y\colon (x, y) \in R\}$ ▷ Include reflexive edges for nodes with an outgoing edge
$\delta R^* \leftarrow R$ ▷ Initial "delta" set is the original relation
while $\delta R^* \neq \emptyset$ **do**
    $\Delta R^* \leftarrow \{(x, y) \mid \exists\, z\colon (x, z) \in \delta R^* \wedge (z, y) \in R^*\}$ ▷ All newly formed edges from combining δR^* and R^*
    $R^* \leftarrow R^* \cup \Delta R^*$ ▷ Add these new edges to the transitive closure
    $\delta R^* \leftarrow \{(x, y) \mid \exists\, z\colon (x, z) \in \delta R^* \wedge (z, y) \in \delta R^*\} \setminus R^*$ ▷ Edges newly spanning two steps of δR^*, minus those already in R^*
end while
return R^*

13.2.2 BFS TC

The breadth-first search-based transitive closure yielded our most significant performance improvement, effectively halving our score from 13,000 to 6,500 in the leaderboard benchmarks. The idea behind this method is straightforward:

1. For each vertex s, we run a breadth-first search over the original graph G.
2. Whenever we discover a new vertex u that is reachable from s, we add an edge $\big(s, u\big)$ into a separate transitive closure graph G'.

Repeating this for each $s \in \{0, \ldots, V-1\}$ ensures that G' ultimately has edges capturing all reachability relationships in the original graph—if there is a path of any length from s to u, then G' has an edge $\big(s, u\big)$. Because BFS has a linear complexity in terms of the number of vertices and edges $\mathcal{O}(|V| + |E|)$, performing it from every vertex leads to a total complexity of $\mathcal{O}\big(|V|(|V| + |E|)\big)$. This proved both simple to implement and efficient in practice.

Algorithm 2 BFS-Based Transitive Closure

Require: A directed graph $G = (V, E)$ with adjacency lists, where $V = \{0, \ldots, V\text{-}1\}$
Ensure: Transitive closure graph $G' = (V, E')$

```
G' ← CreateEmptyGraph(V)          ▷ Initialize G' with same vertex set, no edges
for s ← 0 to V − 1 do
    visited ← [false, ..., false]                          ▷ Array of size V
    queue ← EmptyQueue()
    Enqueue(queue, s)
    while queue is not empty do
        u ← Front(queue)
        Dequeue(queue)
        if not visited[u] then
            visited[u] ← true
            if s ≠ u then
                AddEdge(G', s, u)                          ▷ s → u is in the closure
            end if
            for each edge (u, w) in E do
                if not visited[w] then
                    Enqueue(queue, w)
                end if
            end for
        end if
    end while
end for
return G'
```

13.2.3 Iterative TC

Although our BFS TC implementation yields a significant improvement over the base implementation, we still observe that the transitive closure computation takes significantly

longer for some queries. This is mostly the case for those where the subgraph is quite large. In such cases, BFS must be run repeatedly from each node, which can be computationally expensive when there is a large number of nodes.

We therefore implement a method based on the Floyd-Warshall algorithm Hougardy (2010), which computes transitive closure by iteratively expanding the reachability. The pseudocode for the algorithm can be seen in 3.

Algorithm 3 Iterative Transitive Closure Algorithm

Require: Graph $G = (V, E)$ represented by adjacency lists.
Ensure: Transitive closure graph $G' = (V, E')$.

```
G' ← G                                   ▷ Initialize output graph with edges from G
updated ← true
iteration ← 0
maxIterations ← |V|
while updated and iteration < maxIterations do
    for each node u ∈ V do
        for each reachable vertex v from u in G' do
            for each vertex w reachable from v in G do
                if w is not already reachable from u in G' then
                    Add edge (u, w) to G'
                end if
            end for
        end for
    end for
end while
return G'
```

This achieved an increase in performance for the queries in question. However, we found that the algorithm does not work for queries with reverse edge directions, which cause the algorithm to never terminate. Furthermore, we have found that the previous BFS implementation still has a slight edge for cases where the graphs are shallow.

For this reason, we decided to implement dynamic TC handling, switching to BFS in cases where it performs better, and defaulting to the iterative algorithm for the remaining cases. After submitting this version, we achieved a solid improvement of our leaderboard score by around 800 points.

13.3 Join Order Improvement

Looking at the performance of the queries, it became pretty apparent that the join method needed to be improved. A left-deep join plan executes concatenations sequentially, which can be inefficient when dealing with large and complex queries. To optimize query execution, we implemented a bushy join plan that balances the evaluation order of concatenations based on estimated costs. This implementation utilizes dynamic programming (DP) to determine the optimal execution order, reducing unnecessary computations and minimizing intermediate results.

Our algorithm consists multiple steps:

1. We first initialize two tables. The first table `dp[i][j]`, is used to store the estimated cost of evaluating a subpath from i to j. The second table, `plan[i][j]` is used to store the best split point for the same subpath. This information will then be used to actually perform the merge. To obtain the estimate, we make use of our estimator from the previous part of the assignment.

2. The main part of the algorithm consists of iterating over the increasing length of the subpath until we reach the full length of the path query. With each component, we find the most optimal splits according to the following cost function, which approximates the cost of concatenating two components:

$$\texttt{estimateCost} = \texttt{leftEst.noPaths} * \texttt{rightEst.noIn}$$

   This ensures that we find the most balanced execution order, minimizing intermediate result sizes.

3. The last part is where we take the optimal split points for the entire path query and perform the joins. Starting from the top, we recursively join the components in the most optimal order, until we are left with the final graph.

This implementation yielded us significant improvements. By joining the components in the most optimal order, we significantly reduce the intermediate results. This yielded us an improvement of our score by about 1000 points.

13.4 Improved Graph Representation

After submitting our solution, we have found that although our implementation performed quite well, it had a much worse memory score than other solutions in a similar range around 26, while other solutions had as low as 3. Since memory plays an important factor, that means we got a worse score than these solutions despite our query times being a slight bit faster. This memory overhead comes from the graph implementation. Since both the graphs for the synthetic and the real workloads are sparse, the adjacency list will remain mostly empty.

13.4.1 Compressed Sparse Row

We investigated alternative methods of graph representation. Given the sparsity of our graph, the Compressed Sparse Row (CSR) representation appeared like a good fit. As described in Borštnik et al. (2014), it has a low memory footprint and also allows for fast traversal. The general approach consists of storing all the adjacency information of the nodes together instead of in separate adjacency lists. Since the graphs are sparse, this should lead to large memory savings.

In our implementation, the graph is represented using three arrays:

- offsets: This array stores the indexes where the adjacency list for each node will start. With n nodes, there will be n indexes in the offsets array.

- targets: This array stores the end nodes for the directed edges.

- labels: The standard implementation of CSR does not consider labels. We therefore needed to add an extra array to store label information for each edge. Its size will be the same as that of the target array.

This implementation is very efficient, yet it has a major disadvantage in the sense that it is a static representation. It can therefore not be extended. This is not an issue for operations like transitive closure, but for operations like join, it means the entire representation has to be rebuilt. We were hoping that this overhead would be offset by the increase in performance, but after submitting our solution, we found that it performed a fair bit worse than our previous best solution, worsening the score by almost 2500 points.

It seemed that in practice, the overhead from rebuilding the representation was much larger than we anticipated. We tried to address this using approaches such as a hybrid representation with both adjacency lists and CSR, which did improve the score a little bit, but defeated the purpose of using a memory-efficient representation in the first place. In the end, we did not manage to implement this successfully, leading us to abandoning the approach.

13.4.2 Forward adjacency with reverse index

Although we did not manage to change the graph representation, we noticed that the labels with reverse directions rarely come up in any queries. This lead us to discarding the `reverse_adj` adjacency list from the basic implementation. Instead, we compute a similar structure (`reverseIndex`) from the forward adjacency list whenever it is needed. By only storing the necessary reverse edges for the current computation, `reverseIndex` avoids duplicating the entire graph's adjacency list in reverse form. This resulted in an improvement of around 15 points in `peak_mem` and a modest improvement of about 500 points in the total score.

13.5 Miscellaneous Improvements

In addition to the major improvements previously mentioned, we have also made a series of miscellaneous improvements that lead to minor score improvements.

13.5.1 Graph reading

The function `readFromContiguousFile`, used to read the edges to add to the graph from a given file, has been reworked from using regular expressions to simply scanning for integer tokens with `operator>>`. This avoids the overhead of regex matching, leveraging the fact that the file structure is predictable. Moreover, in order to optimize memory allocations, before reading the edges the function attempts to reserve enough capacity in each adjacency list so that it minimizes the number of times those lists have to expand. Since the file header provides the total edge count and the number of nodes, the code computes an approximate average out-degree using the formula:

$$\texttt{approxOutDegree} = \left\lceil \frac{\texttt{noEdges}}{\texttt{noNodes}} \right\rceil + 1$$

This heuristic is used to call `reserve(approxOutDegree)` for each node's edge list, reducing the amount of expansions needed for the lists. Overall these changes resulted in a `load_time` of 864.0, which is approximately a quarter of the default scores, and an improvement of approximately 100 points for `prep_time`.

13.5.2 Edge deduplication

We have noticed that for the real workload, the graph contains a lot of duplicate edges. Given that the basic implementation does not take this into account, it creates a large overhead for the query evaluation. We therefore introduced a simple deduplication step when reading in the graph. This yielded a pleasantly surprising improvement of around 400 points in the final score.

13.5.3 Precomputing label selections

Another thing we did was precomputing all the label selections in the preparation phase of the estimator. This was beneficial due to our cache implementation. Pre-computing the label subgraphs allows us to cache those before the actual queries are evaluated so that when a query comes, we can just retrieve the label subgraphs from the cache. This saves some computational overhead from having to compute each label subgraph for the first time. This is only possible due to the low number of labels for both the real and synthetic workload graphs. We therefore leveraged this property of both graphs to squeeze out about 500 points worth of score improvement in the leaderboard.

14. Results

Our project demonstrated substantial performance improvements through various optimizations. We have improved the **Smart** aspect of query evaluation through the usage of the estimator to dynamically reorder joins for concatenation queries. Then, we have made major improvements to the **Quick** area by implementing a caching system and improving the transitive closure computation via multiple akgorithms. Lastly, we have also managed to reach minor improvements in the **Frugal** domain, by modifying the adjacency list representation. This together, led to us reaching a leaderboard score of 2649.855. This places us in the top-10 of the leaderboard (at the time of writing), which is a solid performance. Yet, there remain several promising directions for further enhancement.

- **Smart improvements**: Although we have managed to implement query planning for the evaluation of concatenations, that does not affect, for example, queries with a lot of union operations. We believe there are significant performance gains to be had from implementing a more thorough system of query planning. Furthermore, we did not focus on improving evaluation for bound source/target queries, as they did not appear often. However, there are improvements like selection pushdown that would most likely result in better performance for those particular queries.

- **Frugal improvements**: Improve graph representation, especially for sparse graph. The need for a more compact representation is amplified by our use of caching, which

scales the memory overhead required. We will try to investigate solutions which are compatible with the current set of optimizations we already made on the project. Looking back, we should have done it the other way around; first improving the graph representation and then building our implementation on top, as it would make integration much easier.

- **Quick improvements**: implementing parallel execution of independent subqueries could significantly reduce overall query evaluation time. Techniques such as partitioning the graph and parallelizing transitive closure computations could be explored. This is particularly apparent when we look at the following two queries evaluated by our implementation:

```
Processing query: *,(0>|1>|2>|3>|4>),*
Actual (noOut, noPaths, noIn) : (4122, 8972, 1391)
Time to evaluate: 18 ms

Processing query: *,(0>|1>|2>|3>|4>)+,*
Actual (noOut, noPaths, noIn) : (4122, 9020, 1391)
Time to evaluate: 30 ms
```

It is clear that the main bottleneck still lies within the transitive closure. This should therefore be the main focus of future improvement.

References

U. Borštnik, J. VandeVondele, V. Weber, and J. Hutter. Sparse matrix multiplication: The distributed block-compressed sparse row library. *Parallel Computing*, 40(5-6):47–58, 2014.

C. Demetrescu and G. F. Italiano. Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *Journal of Discrete Algorithms*, 4(3):353–383, 2006.

P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science*, (Proceedings), 2007.

C. A. Galindo-Legaria, M. M. Joshi, F. Waas, and M.-C. Wu. Statistics on views. In *Proceedings 2003 VLDB Conference*, pages 952–962. Elsevier, 2003.

L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 461–472, 2001.

P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, volume 1, pages 541–550, 2001.

G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

S. Hougardy. The floyd–warshall algorithm on graphs with negative cycles. *Information Processing Letters*, 110(8-9):279–281, 2010.

V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Very Large Data Bases Conference*, 1997. URL `https://api.semanticscholar.org/CorpusID:8408623`.

W. van Leeuwen, G. Fletcher, and N. Yakovets. A general cardinality estimation framework for subgraph matching in property graphs. *IEEE Transactions on Knowledge and Data Engineering*, 35(6):5485–5505, 2022.

J. Wang, N. Ntarmos, and P. Triantafillou. Graphcache: A caching system for graph queries. 2017.

N. Zhang, M. T. Ozsu, A. Aboulnaga, and I. F. Ilyas. Xseed: Accurate and fast cardinality estimation for xpath queries. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 61–61. IEEE, 2006.