Engineering Data Intensive Systems - 2IMD10

EDS - Project Report

Team Number - 16

Full Name	Discord Username	Email
Divyansh Purohit	wah_shampy	d.purohit@student.tue.nl
Likhit Vesalapu	likhit7.	l.vesalapu@student.tue.nl
Prathamesh Samal	viper__101	p.samal@student.tue.nl
Elena Terzieva	ellie218388	e.e.terzieva@student.tue.nl

Eindhoven, February 1, 2026

1 ABSTRACT

Accurate cardinality estimation is fundamental to query optimization in graph databases, enabling the selection of efficient execution plans for regular path queries. In this report, we present a hybrid cardinality estimator that combines multiple statistical synopses including per-label statistics, pairwise label correlations, and characteristic sets with a weighted and stratified sampling strategy for complex queries. Our approach balances estimation accuracy against preparation time and memory overhead, achieving competitive performance on both synthetic and real-world workloads.

2 INTRODUCTION

The efficiency of query processing is an important topic in the domain of databases, where cardinality estimation plays a central role. Modern graph and XML database systems rely on cardinality estimators as a core component of their cost based query optimizers. A cardinality estimator is a statistical module that, given an abstract description of a query (a labeled path pattern starting from some node and ending at another), predicts how many results this query is likely to produce. These predictions are crucial for choosing efficient execution plans, selecting join orders, and deciding which physical operators to use. Without an accurate cardinality estimator, an optimizer can easily choose plans that are orders of magnitude slower than necessary.

Traditional cardinality estimators for path queries are based on compact summaries of structural synopses that record per-label statistics, degree distributions, and limited information about how labels coincide. Although such synopses can be very effective for short, simple path patterns, their accuracy rapidly degrades for long paths, patterns with repetition, or data with strong structural correlations. In these cases, naive independence assumptions break down and purely synopsis based estimators can produce large errors.

To address these challenges, our work investigates a sampling based cardinality estimator that involves four complementary approaches: Stratified reservoir sampling, End-biased sampling, Correlated sampling, XSEED based Simulation.

3 LITERATURE SURVEY

The topic of cardinality estimation continues to be one of the most important in query optimization. There are different techniques that have been developed for estimating the size of the results produced by a query. In this project, we are dealing with RDF databases that differ from the typical relational schemas because of their structure, which leads to different assumptions and decisions made when estimating the cardinality of a query. In query optimisation, synopses are precomputed summaries of database data (like histograms, sketches, samples) that database systems use to quickly estimate the cost of different query execution plans. Bonifati et al. survey cardinality estimation techniques specifically designed for graph query languages. The two approaches discussed are cardinality of paths and cardinality on patterns. With cardinality on paths, the query path is split into simpler segments and predefined formulas are used for concentration, union and inversion are combined to estimate the result. Cardinality on patterns, on the other hand focuses on graph patterns with characteristic sets or graph summaries. With characteristic sets, each node is summarised by the set of outgoing edge labels it has, and these "characteristic sets" are used to estimate how many nodes (and joins) participate in a given pattern. They are discussed in a study by Neumann and Moerkotte, where they observe that in RDF databases, many subjects share the same set of outgoing predicates, and define a characteristic set as exactly this set of predicates attached to a subject [7]. With graph summarizers, the graph is compressed into a smaller summary (by grouping similar nodes/edges), and cardinalities for complex path or subgraph patterns are estimated by evaluating the query on this summary instead of on the full graph.

Two statistical methods for dealing with that are sampling and histograms. Estan and Naughton propose end-biased samples for join cardinality estimation in relational databases. The method retains tuples with probability proportional to their frequency in join columns, so that high-frequency values (responsible for most join output) are over-represented. This yields much more accurate join size estimates than uniform sampling at comparable sample sizes [5]. Babcock and Chaudhuri introduce a technique based on Bayesian inference over precomputed random samples, which models uncertainty and produces estimates together with confidence information. Their method captures multi-dimensional correlations without assuming attribute independence and remains time-efficient. Importantly, it supports user- or application-specified trade-offs between performance and predictability, contributing to the design of more robust query optimisers [1].

Histograms are one of the most widely used methods for cardinality estimation in relational DBMS. The basic idea is to partition the domain of an attribute into buckets and store, for each bucket, the number of tuples (and sometimes additional stats such as distinct values). Query selectivities are then approximated by assuming a uniform distribution of values inside each bucket and summing contributions from the buckets that overlap the predicate. A detailed survey of histogram-based and related synopsis techniques is given by Cormode, who reviews equi-width, equi-depth and more advanced frequency-based histograms, as well as their multi-dimensional extensions and maintenance trade-offs [3]. Joins can be very expensive, and to try to bridge the gap between query efficiency and accuracy, Li et al. introduced the new algorithm Wander Join [6]. Wander Join is a join estimation method that treats a multi-table join like a graph and then performs random walks through the graph instead of computing the full join. Since it only touches a small number of data points and does not need to compute statistics for the full data, the approximation results are produced quickly and more accurately with the increase in performed random

walks.

4 IMPLEMENTATION

4.1 Approach and Planning

4.1.1 Problem Analysis

In order to address the problem of cardinality estimation for path queries, our team first examined the basic properties of graph data. Especially in networks, graph data shows heavy tailed, power-law distribution, variable-length traversals and Kleene closures in comparison to relational tables where data usually follows uniform distribution. The scoring metric for this project emphasizes the accuracy on real workloads with a weight of two and also penalizing the run times. This tuned our strategy to get accurate estimates without expensive computation.

The provided graph contained 10,000 nodes, 22,122 edges, and 36 distinct edge labels. With an average out-degree of approximately 2.92 and a maximum out-degree of 43, the graph is relatively sparse (density approx 0.02%). The degree distribution follows a classic power-law pattern: 37% of nodes have out-degree 1, while only a handful of hub nodes exceed degree 20. This skewed distribution means that a small number of high-degree nodes disproportionately contribute to path counts, making uniform sampling ineffective.

The label distribution is similarly skewed, with label 8 appearing in over 5,000 edges (23% of all edges), while 12 labels appear fewer than 50 times each. This imbalance affects join selectivity and makes certain label combinations far more common than others. The query workload spanned diverse patterns:

- **Single predicates**: e.g., `,1>,`, `,8>,`
- **Concatenations**: e.g., `,1>/8>,`, `,1>/8>/9>/8>,` (up to length 5)
- **Kleene closures**: e.g., `,(1>)+,`, `,(8>)+,`
- **Mixed patterns**: e.g., `,1>/(8>)+/(9>)+,`
- **Label unions**: e.g., `,(0>|1>|2>|3>|4>)+,`
- **Bound source queries**: e.g., `200,1>/8>,`, `227,(1>)+/(8>)+,`

The scoring metric for this project emphasizes accuracy on real workloads with a weight of two while also penalizing preparation and estimation time. This guided our strategy: we needed accurate estimates without expensive runtime computation.

4.1.2 Technique Selection Based on Literature Survey

Based on our literature review, we selected the following approaches for our hybrid estimation model:

- **End-biased Sampling:** This strategy was chosen in order to address the variation brought on by power-law distributions. We reasoned that probabilistic estimate is not appropriate for frequent values (hubs), since they have the greatest influence on join size and variance. We eliminate the main cause of instability in the estimator by explicitly isolating and tracking the top-K highest-degree nodes using exact logic, leaving a lower-variance population (the tail) that is easier to sample.
- **Stratified Reservoir Sampling:** Rare edge labels are not guaranteed to be covered by a global random sample. To make sure that every potential transition in the graph has representative start nodes, we decided to use stratified reservoirs, which allocate a predetermined sample budget per unique label. In order to reduce endless failures on queries with rare predicates, this choice was crucial.
- **Correlated Sampling by Hashing:** Since independent random sampling statistically fails to capture the intersection of joins in sparse samples, we ignored it for the tail. We make sure that sample choices are consistent across various query contexts by using a deterministic hashing approach as suggested by Estan et al. given as follows:

$$h(v) \leq p_v \tag{1}$$

  where p_V is the sampling probability for a given value of v and h is the hashing function which maps values in the range $[0,1]$.
- **XSEED-based Seed Simulation:** In clustered graphs, the assumption of independence between path steps is broken by standard histograms. The idea of employing bound nodes as "seeds" for cardinality estimation is presented in the XSEED paper [11]. We incorporated this idea into our Sampling Model, treating our Stratified Samples as "seeds" and executing the query path from them. This enables us to capture the structural relationships in the graph that are missed by static statistical methods.

We decided to pursue with a Hybrid Cardinality Estimator that adjusts to the structural skew of the graph by combining the above mentioned approaches. By dynamically replicating graph traversals on a subgraph, this synergy enables the system to get around the drawbacks of static statistics and balance the accuracy with the runtime efficiency of sampling.

4.1.3 Expected Challenges

We anticipated several challenges:

- **Balancing accuracy and speed**: More sophisticated statistics and complex simulations improve accuracy but increase both preparation and estimation time. In

addition, reservoir sizes and the estimation time penalty had to be carefully balanced. We anticipated that wander join approaches, which use random walks to estimate join cardinalities, would greatly benefit accuracy but adversely affect estimation time.

- **Handling Kleene closures**: Single predicate queries like 1 and concatenation queries like 1 /2 were straightforward and could be directly computed from our stored label statistics and join synopses. However, transitive closure queries can produce result sets ranging from empty to quadratic in graph size, making estimation inherently difficult.
- **Correlation capture**: Independence assumptions between path segments would lead to systematic over or under estimation, but capturing accurate and extensive correlations would require exhaustive precomputations.
- **Memory constraints**: Storing detailed statistics for all label pairs (pairwise correlations) would scale quadratically with the number of labels in the graph.

4.2 Implementation Progress and Results

We used an iterative life cycle for our implementation. Before finalizing our ultimate End-Biased Stratified architecture, we ran our experiments with a number of approaches, including a computationally costly Random Walk attempt.

4.2.1 Iteration 1: Basic Synopsis

We began by implementing foundational statistics computed in a single pass over the graph. This included edge counts per label and a matrix to store pairwise correlations between labels. By adding up the product of degrees, we were able to determine the precise number of pathways for any length-2 label combination. A join statistics matrix was also introduced to capture the number of two-hop paths between all label pairs, providing exact estimates for length-2 queries.

Result: This baseline achieved reasonable accuracy on simple queries but struggled with complex patterns and showed high variance on longer paths.

4.2.2 Iteration 2: Wander Joins and Random Walks

We implemented a Wander Join estimator to increase accuracy for Kleene closures and long pathways. In order to naturally approximate the path distribution for each query, we conducted random walks beginning from a collection of sampled nodes. To further refine this approach, we explored several advanced adaptive sampling techniques: the Filtering-Sampling Approach (FaSTest), that dynamically adjusted sample sizes based on query characteristics, along with an adaptive stopping criteria that terminated walks once the 95% confidence interval around the Horvitz-Thompson estimate fell below a 5% error threshold, thereby saving time on dense queries.

Result: Since this method physically traversed the graph structure to capture correlations, this method produced remarkable accuracy, however estimation time (T_{est}) grew significantly. Even with our optimizations, a large number of walks were required in order to get stable estimates, which went against the time constraints set up for this project. Instead, we opted for a pre calculated sampling strategy.

4.2.3 Iteration 3: Hybrid Approach

Our estimating model was developed through an ordered breakdown of the problem rather than being chosen as a single technique. In order to address particular data features, we integrated various approaches in three logical steps.

Step 1: Decomposition by Degree The existence of high-degree nodes is the main cause of errors in graph estimation. We developed a split-execution model based on the End-Biased Sampling concepts. We proposed treating the graph G as G_{head} and G_{tail}, into two separate subgraphs. Since any probabilistic variance here would propagate quadratically during joins, we concluded that G_{head} (the top-K nodes) must be effectively retained in memory and queried accurately. In G_{tail}, where the variance is inherently lower, the probabilistic estimation is strongly restricted.

Step 2: Handling Sparsity Standard sampling is ineffective for rare predicates in path queries, despite Estan et al.'s suggestion to weight by frequency. A global sample will be unable to capture a label L if it is present in only 0.1% of the graph. The sampling logic was expanded to be stratified. Each edge label is treated as a separate population strata. Regardless of how uncommon the edge label is, we mathematically guarantee that the probability of providing a "zero-estimate" for a valid query path is reduced by guaranteeing a constant reservoir size for each label.

Step 3: Correlation Preservation In essence, a path query $X \to Y \to Z$ is an intersection problem. The fundamental correlations needed to compute these crossings are eliminated by independent random sampling. We decided to use Correlated Hashing. In case, a node exists in the reservoir for label A (as a target), it will also be present in the reservoir for label B (as a source), as long as it satisfies the probability threshold, by making the inclusion of a node into a reservoir a deterministic function of its ID ($h(v) \leq p_v$). This maintains the linkability.

Step 4: Simulation-Based Estimation Finally, we used the Seed-Based approach from the XSEED paper to solve for complex query types that defy static formulas, such as Kleene closures ($+$). In addition to being statistical representatives, we thought of our stratified samples as "active seeds." Rather than merging histograms, the estimator measures reachability by physically running the query from these seeds and scaling the outcome by the inverse of the inclusion probability ($1/p_v$).

Result: This approach achieved the best trade-off between accuracy and estimation time on simple as well as complex queries. queries. The only drawback of this approach was that it occupied significantly more memory, which was further

refined in the next iteration.

4.2.4 Iteration 4: Optimization

The final iteration focused on performance optimization without sacrificing on the accuracy. This iteration was done in order to reduce Estimation Time (T_{est}) and Preparation Time (T_{prep}).

- **Binary Search for Adjacency:** It was inefficient to search adjacency lists linearly for edges that matched a given start label for the high-degree top-k nodes. Using binary search on the sorted edge lists, we optimized the sampling algorithm. This resulted in a targeted speedup for the most frequently used nodes by lowering the edge lookup difficulty from linear to logarithmic.
- **Memory Reuse:** Our code initially generated fresh temporary lists for each node it handled during the preparation stage. This required more memory and was slow due to large number of nodes. These list creations were relocated outside of the main loop. The same memory block is reused repeatedly since we only need to generate the list once and then clear it after processing each node. As a result, the overhead of continuous memory allocation was significantly reduced.

4.2.5 Results

Our final hybrid estimator shows a good trade-off between runtime and estimation accuracy. The leaderboard's performance metrics are as follows

Table 1: Final Leaderboard Performance

Metric	Value
Discord Username	viper__101
Synthetic Accuracy (Eq_{syn})	1.18
Real-Workload Accuracy (Eq_{real})	8.86
Preparation Time (T_{prep})	153.88 ms
Estimation Time (T_{est})	32.58 ms
Peak Memory Usage (M)	24.07 MB
Final Score	**49.63**

5 Part 2: Efficient Query Evaluation

In Part 2 of the project, we shift focus from cardinality estimation to efficient query evaluation. The goal is to make Quicksilver **SQUIFfy**: Smart, Quick, and Frugal. We focused heavily on balancing the trade-offs between execution speed, preparation time, and strict memory limits. We didn't just want it to be fast; we wanted it to be robust and memory efficient as per Frugal requirement.

5.1 Strategy

Our optimization strategy followed a systematic approach inspired by research into high-performance graph processing systems. We categorized our efforts into three pillars: Smart (choosing optimal algorithms), Quick (reducing computation time), and Frugal (minimizing memory overhead).

5.1.1 Smart

We made informed decisions based on query patterns and empirical benchmarking:

- **Separate Bound Source/Target Handlers:** We used forward BFS traversal for queries with bound sources and backward BFS from the target for queries with bound targets. This method helped us to avoid full-graph traversals and significantly reduced the search space.
- **Node ID Mapping:** We maintained a translation table to map the source and target node IDs from original to reordered IDs at evaluation time so that when the graph is reordered (discussed in Quick) and nodes get new IDs, the evaluator continues work with consistent node identifiers.

5.1.2 Quick: Reducing Computation Time

Here our primary focus was eliminating redundant work and improving cache efficiency:

- **Label-indexed CSR:** We pre-partition edges by label during graph loading instead of filtering them during BFS, which eliminated the label check from the hottest code path. This reduced branch mispredictions and improved instruction level parallelism.
- **Graph Reordering:** We implemented BFS based node ID reassignment, starting from the highest-degree node. This approach assigns consecutive IDs to the nodes visited together during traversal, eventually improving spatial locality when accessing adjacency data in the CSR arrays.
- **Tracking visited nodes using timestamp:** Rather than clearing the visited array between queries (O(V) operation), we used a monotonically increasing timestamp counter to indicate the status of a node. We marked a node with the current timestamp when visiting it, and the node is considered unvisited if its timestamp doesn't match the current query's timestamp (O(1) operation).

5.1.3 Frugal: Memory Efficiency

Here we focused on minimizing dynamic memory operations because memory allocation and deallocation are expensive operations.

- **Memory Pooling:** All vectors used during BFS traversal were pre-allocated once during the prepare phase and reused across all queries via swap() operations rather than repeated allocation and deallocation.
- **CSR Format:** We converted adjacency lists to Compressed Sparse Row format, storing edges in contiguous arrays indexed by node ID, to reduce memory fragmentation and improve cache utilization.

5.2 Literature Survey

Work on fast graph query evaluation repeatedly shows that performance depends less on high-level query language rules and more on how traversal is executed: reducing wasted exploration, using cache-friendly graph layouts, and avoiding overhead inside the traversal loop. Shared-memory graph systems and RDF engines especially emphasize contiguous adjacency storage, locality-aware preprocessing, and access-path choices that prevent repeated filtering during evaluation.

In direction-optimizing BFS, Beamer et al. show that switching between traversal styles can dramatically reduce the number of edges examined on real "small-world" graphs, which is exactly the problem when a naive traversal expands too broadly. Their key message is that how you traverse (direction/strategy decisions) can matter as much as raw low-level optimization, because it directly shrinks the explored search space [2]. Moreover, when graph preprocessing changes vertex IDs (for example after reordering), systems typically need a stable way to preserve semantics across representations. In the graph ordering work by Wei et al., the goal is to rearrange vertex order to improve cache behavior. However, this implicitly creates a "logical graph" (original IDs) and a "physical graph" (reordered IDs). The paper motivates the practical need to keep query evaluation correct even when the physical layout changes — conceptually the same reason many systems maintain an ID mapping layer around reordered graphs [9].

For traversal-heavy workloads, research shows that performance is often memory-bound, so the fastest solutions focus on contiguous layouts and minimizing work in the tight loop. Ligra is a well-known shared-memory framework demonstrating that graph traversal becomes significantly faster when adjacency is stored and processed in cache-friendly ways, enabling efficient neighbor scans and reducing overhead from pointer-heavy structures. Although Ligra is a framework, its results strongly support the idea that CSR-like representations and streamlined traversal primitives are a practical baseline for high throughput BFS-style processing [8]. A second line of work targets locality directly via graph reordering. Wei et al. show that reordering vertices can reduce CPU cache miss ratios across several graph algorithms, producing speedups without changing the underlying algorithm. Their contribution is important for query evaluation because it treats graph layout as an optimization target: by reassigning IDs so that vertices accessed close in time are also close in memory, the cost of adjacency and metadata access drops [9]. Visited-state handling is another recurring bottleneck in fast traversal. Beamer et al. discuss optimizations such as using compact visited representations (for example bitmaps) to reduce costly random accesses and keep visited checks efficient on shared-memory machines. The point is that even "small" bookkeeping operations can dominate runtime when done at scale, so visited tracking must be engineered carefully [2]. A large part of the memory efficiency in graph query engines comes from keeping data structures compact and avoiding repeated allocations during query execution. Many systems therefore rely on two practical ideas such as storing adjacency in a contiguous sparse format (typically CSR), and pre-allocating and reusing working buffers instead of allocating/freeing memory for every query. Several systems make this point even more explicitly by standardizing around CSR as the baseline representation due to its compactness and predictable access pattern. Dhulipala et al. (GBBS) state that graphs in the suite are stored in Compressed Sparse Row (CSR), where neighbor lists live in contiguous arrays with an offset array per node. This design reduces fragmentation and supports cache-friendly scans. The key motivation is that pointer-heavy adjacency lists create overhead and irregular access, while CSR improves both memory footprint and traversal throughput [4]. Many high-performance systems reduce memory overhead by minimizing dynamic allocations during execution. Winter et al. (faimGraph) present a memory-management approach designed for fully-dynamic graphs on the GPU. While faimGraph is GPU-focused, its core principle—using one large pre-allocation to avoid the overhead and fragmentation of repeated memory allocation or vector resizing—maps cleanly to CPU implementations. We adopted this by pre-allocating our working buffers once to keep runtime memory behavior stable. [10].

5.3 Implementation Progress

We followed an iterative approach for optimizing our performance, implementing and benchmarking each technique independently to measure its overall impact. In the sections below, we explain each optimization iteration experiments. We also report the performance metrics achieved in each iteration, along with a improvement percentage from the previous iteration (represented by Δ). A positive value indicates improvement in the performance parameter whereas a negative value represents decrease in the performance.

5.3.1 Iteration 1: Baseline Analysis

The baseline implementation used standard adjacency lists with vectors of (label, target) pairs. Each query allocated fresh vectors for BFS traversal and used a boolean visited array that was cleared between queries. The simple evaluator performed linear scans through the adjacency lists, checking each edge's label against the query predicate.

Results: Profiling revealed several bottlenecks: repeated memory allocation for each query, O(V) cost to clear the visited array, poor cache locality due to scattered memory access, and redundant label checks in the inner traversal loop.

5.3.2 Iteration 2: Bound-Aware Evaluation and Semi-Naive Transitive Closure

After the experiments on our baseline, w e found that queries with bound sources or targets continued to unnecessarily traverse the whole graph. Furthermore, Kleene star searches

Table 2: Iteration 1: Baseline Performance

Metric	Value
Preparation Time (T_{prep})	145.87 ms
Evaluation Time T_{eval}^{syn}	1213.70 ms
Evaluation Time T_{eval}^{real}	17993.72 ms
Load Time (T_{load})	3985.82 ms
Peak Memory Usage (M_{eval})	29.48 MB
Score	**$\sim$19,800**

were creating duplicate work by recalculating every reachable node at each cycle.

- **Bound-Aware Evaluation:** We implemented separate independent evaluation handlers for queries with bound sources and bound targets. Instead of traversing the entire graph, we start a breadth first search from the single known node, significantly reducing the search space for constrained queries.
- **Semi-Naive Transitive Closure:** We replaced the naive transitive closure algorithm with a semi-naive approach for Kleene star queries. Instead of recalculating all reachable nodes at each iteration, we process only newly discovered edges, avoiding redundant computation and significantly improving performance for recursive path patterns.

Results: The score improved from $\tilde{1}$9,800 to 7,248 (63% improvement). This confirmed that algorithmic issues were the major bottleneck. However, this also revealed that significant time was still being spent on constructing and managing intermediate result graphs, hinting us towards data structure optimizations.

Table 3: Iteration 2: Bound-Aware Evaluation

Metric	Value	Δ
Preparation Time (T_{prep})	1505.31 ms	–932%
Eval Time Syn (T_{eval}^{syn})	341.26 ms	+72%
Eval Time Real (T_{eval}^{real})	6403.16 ms	+64%
Load Time (T_{load})	4092.68 ms	–3%
Peak Memory (M_{eval})	24.87 MB	+16%
Score	**7,248**	**+63%**

5.3.3 Iteration 3: Compact Intermediate Results

After the algorithmic changes were implemented, profiling showed that creating and maintaining intermediate result graphs took a considerable amount of time. We observed that these intermediary structures kept duplicate data: reverse adjacency lists were kept even if joins only traveled forward, and labels were kept even though edges had already been filtered. To subside these issues, we used the following approaches:

- **CompactIR Data Structure:** We introduced a lightweight intermediate result structure that stores only source-to-target mappings. We removed labels from intermediate results since once edges are filtered by label during selection, storing the label becomes redundant. We also eliminated the reverse adjacency list because join operations only traverse forward, never requiring backward lookups on intermediate data.
- **Deferred Deduplication:** We also removed per-insert duplicate checking, which required expensive lookups on every edge addition. Instead, we allowed duplicates during construction and performed a batch sort-and-deduplicate operation at the end by using `finalize()`. This approach is more cache-friendly and significantly reduces computation time.
- **Hash-based Existence Checking:** For transitive closure computation, we replaced linear search with hash sets for tracking visited edges, reducing existence checks from O(N) to O(1) and speeding up Kleene star queries.

Result: The score improved from 7,248 to 4,255 (41% improvement). Memory usage and computation time were decreased by eliminating redundant storage and delaying deduplication.

Table 4: Iteration 3: Compact Intermediate Results

Metric	Value	Δ
Preparation Time (T_{prep})	1476.81 ms	+2%
Eval Time Syn (T_{eval}^{syn})	212.29 ms	+38%
Eval Time Real (T_{eval}^{real})	3828.77 ms	+40%
Load Time (T_{load})	3911.26 ms	+4%
Peak Memory (M_{eval})	10.40 MB	+58%
Score	**4,255**	**+41%**

5.3.4 Iteration 4: Direct BFS Traversal

In the previous iteration, our approach continued to materialize intermediate result graphs and carry out explicit join operations in spite of the CompactIR enhancements. Since we only want the final reachable nodes for path queries, not the intermediary edges, we thought if this abstraction was even essential. As a result, we completely stopped using intermediate graph building.

- **Removed Intermediate Graph Construction:** We eliminated the CompactIR data structure and join operations entirely. Instead of building intermediate result graphs and joining them, we now traverse the query path directly while maintaining only the current frontier. This avoids materializing all intermediate edges, significantly reducing memory usage and computation time.
- **Specialized Evaluation Paths:** We implemented three separate evaluation functions: forward BFS for bound-source queries, backward BFS for bound-target queries,

and multi-source traversal for unbound queries. Each path is optimized for its specific access pattern.

- **Bitvector-based Tracking:** We replaced hash sets with bitvectors for tracking frontier membership and visited nodes. For dense node ID spaces, bitvectors provide O(1) lookups with better cache locality than hash-based structures.
- **Label-indexed Edge Lists:** Precomputed lists were added that provide direct access to all edges with a given label, eliminating the need to scan entire adjacency lists and filter by label during unbound evaluation.
- **Memory Optimizations:** vector pre-reservation were added to reduce reallocations during BFS and used move semantics to avoid unnecessary vector copies when swapping frontiers.

Result: The score improved from 4,255 to 2,063 (52% improvement). Eliminating intermediate graph construction was a major win, confirming that the join-based abstraction was unnecessary overhead. However, experiments showed a new bottleneck: the unbound evaluation was allocating thousands of bitvectors per query, one for each active source.

Table 5: Iteration 4: Direct BFS Traversal

Metric	Value	Δ
Preparation Time (T_{prep})	1451.41 ms	+2%
Eval Time Syn (T_{eval}^{syn})	55.70 ms	+74%
Eval Time Real (T_{eval}^{real})	1896.07 ms	+50%
Load Time (T_{load})	4456.80 ms	−14%
Peak Memory (M_{eval})	5.20 MB	+50%
Score	**2,063**	**+52%**

5.3.5 Iteration 5: Timestamp-based Visited Tracking

In the previous iteration, direct BFS implementation showed a new bottleneck: in `evaluateUnbound()`, we were allocating a fresh `vector<bool>(V)` bitvector for each active source to track visited nodes. With thousands of active sources per query, this meant thousands of O(V) allocations which caused a significant overhead on the query time.

- **Removed Per-Source Bitvector Allocation:** In the previous iteration, the unbound evaluation allocated a separate `vector<bool>(V)` for each active source to track visited nodes. With thousands of active sources, this caused thousands of memory allocations per query, creating significant overhead.
- **Shared Timestamp Array:** We replaced per-source bitvectors with a single shared `visitedTime` array. Each source receives a unique timestamp via an incrementing counter, and a node is considered visited if its timestamp matches the current value. This eliminates all per-source allocations and avoids the need to clear the array between sources, with the timestamp automatically invalidating previous visits.
- **Graph Finalization:** We added a `finalizeGraph()` function called once after loading that sorts all adjacency lists by label and sorts label edge lists by source. This ensures edges with the same label are contiguous in memory, improving cache locality during traversal, and improves sequential access patterns during unbound evaluation.

Result: The score improved from 2,063 to 1,395 (32% improvement). The shared timestamp technique eliminated thousands of allocations per query with minimal overhead. With allocation bottlenecks resolved, experiments now showed that cache misses during graph traversal had become the major cost, leading us toward memory layout optimizations.

Table 6: Iteration 5: Timestamp-based Visited Tracking

Metric	Value	Δ
Preparation Time (T_{prep})	1469.60 ms	−1%
Eval Time Syn (T_{eval}^{syn})	50.18 ms	+10%
Eval Time Real (T_{eval}^{real})	1238.07 ms	+35%
Load Time (T_{load})	4428.61 ms	+1%
Peak Memory (M_{eval})	5.00 MB	+4%
Score	**1,395**	**+32%**

5.3.6 Iteration 6: CSR Format, Graph Reordering, and Label-Indexed Structures

Benchmarking revealed that cache misses during graph traversal had emerged as the main bottleneck following the removal of allocation overhead. The adjacency list structure `vector<vector<pair>>` dispersed edges throughout memory, resulting in frequent cache misses while attempting to access neighbors. Furthermore, nodes visited together during BFS frequently had distant IDs due to the arbitrary assignment of node IDs during graph loading, which resulted in poor spatial proximity.

- **Memory Pooling:** We moved all BFS-related vectors (`frontier`, `nextFrontier`, `visitedTime`, etc.) to class members, pre-allocated once during `prepare()`. Instead of allocating new vectors per query, we reuse them via `swap()` and `clear()`, eliminating allocation overhead from the query hot path.
- **Compressed Sparse Row (CSR) Format:** We converted the adjacency list representation (`vector<vector<pair>>`) to CSR format with two contiguous arrays: `offsets[V+1]` storing the starting index of each node's edges, and `edges[E]`

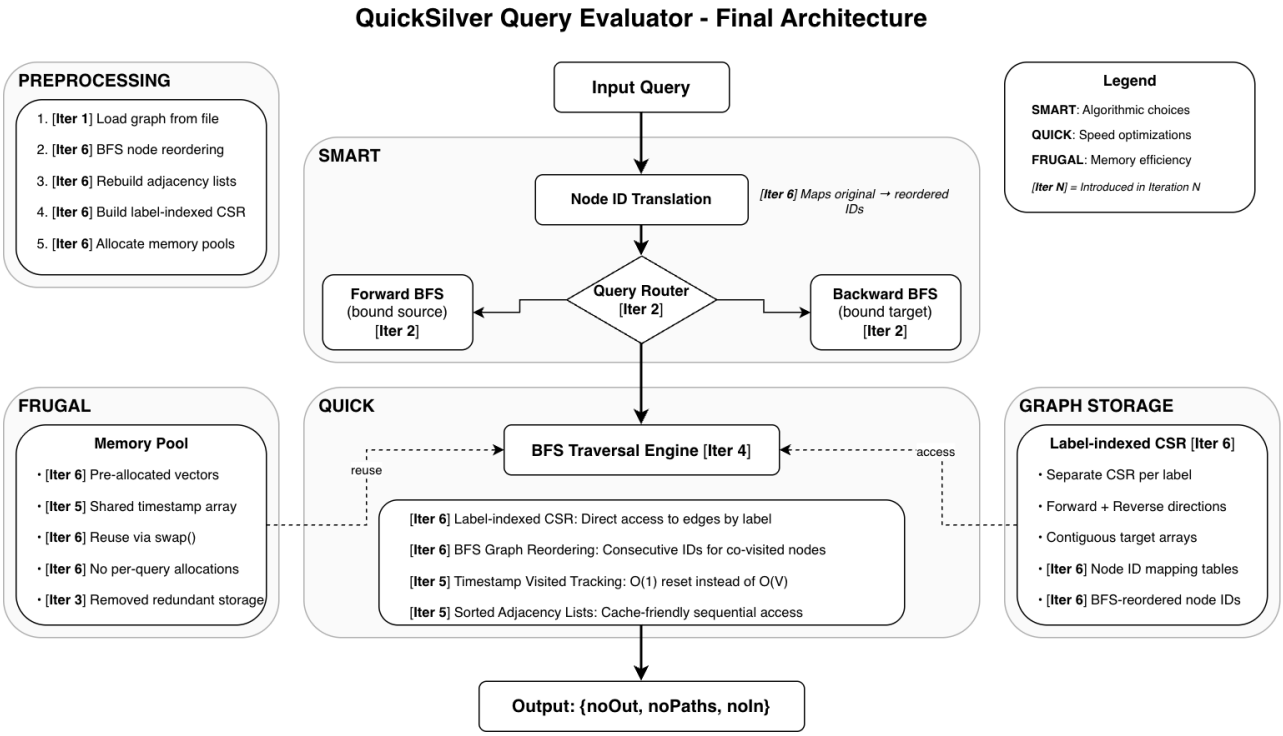

Figure 1: Final architecture of the QuickSilver Query Evaluator. Iteration numbers indicate when each optimization was introduced.

storing the edge data contiguously. This dramatically improves cache utilization during traversal.

- **BFS-based Graph Reordering:** We implemented BFS based node ID reassignment, starting from the highest-degree node. This approach assigns consecutive IDs to the nodes visited together during traversal, eventually improving spatial locality when accessing adjacency data in the CSR arrays.
- **Node ID Translation:** We maintained `oldToNew[]` and `newToOld[]` mapping arrays so that incoming queries using original node IDs are correctly translated to reordered IDs before evaluation.
- **Label-Indexed CSR:** We created separate CSR structures for each label (`csr_label_offsets[label][node]` and `csr_label_targets[label][]`). This eliminated the label comparison from the inner BFS loop entirely, facilitating direct iteration over the edges that are guaranteed to have the correct label instead of checking `if (edge.label == targetLabel)` for every edge.
- **Redundant Storage Removal:** After implementing label-indexed CSR, the original generic CSR arrays were no longer used. We removed them, reducing memory footprint and improving cache efficiency.

Result: Our final result was achieved when the score increased from 1,395 to 716 (49% improvement). The cache locality issue was fully solved by using a combination of label-indexed structures, graph reordering, and CSR format. The final results achieved after this iteration can be found using Table 7.

5.4 Results Summary

Table 7 summarizes our complete optimization journey from baseline to final implementation.

Table 7: Final Leaderboard Performance: After Iteration 6

Metric	Value
Discord Username	**wah_shampy**
Preparation Time (T_{prep})	1834.47 ms
Evaluation Time T_{eval}^{syn}	29.55 ms
Evaluation Time T_{eval}^{real}	564.33 ms
Load Time (T_{load})	5053.52 ms
Peak Memory Usage (M_{eval})	5.67 MB
Score	**716**

Table 8 summarizes our optimization journey from baseline to final implementation.

5.5 Conclusion: Part 2

Through systematic optimization, we achieved a 94.8% reduction in score (approximately 19,800 to 716), corresponding to a significant reduction in evaluation time. The most impactful optimizations were:

Table 8: Overall Optimization Progress

Iteration	Type	Score
1. Baseline	—	19,800
2. Bound-Aware Eval	S	7,248
3. Compact IR	F	4,255
4. Direct BFS	S, Q	2,063
5. Timestamp Tracking	Q, F	1,395
6. CSR + Reordering	Q	716

S = Smart, Q = Quick, F = Frugal

1. **Label-indexed CSR:** Eliminating the label check from the inner loop provided substantial gains by removing branch mispredictions and reducing the iteration count to only relevant edges.

2. **Generic CSR Format:** Converting from vector-of-vectors to contiguous arrays dramatically improved cache locality and eliminated pointer chasing overhead.

3. **Memory Pooling:** Pre-allocating and reusing vectors eliminated allocation overhead that previously dominated query processing time.

4. **Graph Reordering:** BFS-based node ID reassignment improved spatial locality by ensuring frequently co-accessed nodes have nearby memory addresses.

Failed Optimization Attempts: Our optimization journey also included attempts that did not yield positive results, providing valuable lessons about the gap between theoretical improvements and practical performance.

- We implemented a **single-label fast path** that added a conditional branch to bypass the label iteration loop when a query involved only one label. In theory, this should have reduced loop overhead for the majority of our queries which involved single labels. However, this branch was evaluated at every node during BFS traversal, and the CPU's branch predictor struggled with the mixed true/false pattern, causing pipeline stalls that exceeded the time saved by skipping the loop.

- We attempted **frontier sorting**, where we sorted the BFS frontier by node ID before processing each level, expecting that sequential memory access to CSR arrays would improve cache hit rates. While theoretically sound, the O(n log n) sorting cost at every BFS level outweighed the cache benefits, particularly since our graph reordering optimization had already improved spatial locality at preprocessing time, making runtime sorting redundant.

These failures reinforced that micro-optimizations introducing additional branches or per-level computations must be empirically validated, as their overhead can compound across millions of node visits in large-scale graph traversals.

The final system achieves strong performance through a combination of memory efficiency, cache-aware data structures, and algorithmic simplicity. These techniques are broadly applicable to graph processing systems where traversal performance is critical.

6 Conclusion

This project addressed two fundamental challenges in graph database systems: cardinality estimation and efficient query evaluation.

For cardinality estimation (Part 1), we developed a hybrid estimator combining end-biased sampling, stratified reservoir sampling, correlated hashing, and XSEED-based simulation. This approach achieved competitive accuracy on both synthetic and real workloads while maintaining reasonable preparation and estimation times.

For query evaluation (Part 2), we systematically optimized the Quicksilver system through memory pooling, CSR representations, graph reordering, and label-indexed data structures. We learned that careful attention to memory access patterns and data structure design can yield dramatic performance gains.

References

[1] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 119–130, Baltimore, MD, USA, 2005. ACM.

[2] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City, Utah, USA, Nov. 2012. IEEE.

[3] G. Cormode. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2011.

[4] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun. The graph based benchmark suite (gbbs). In *3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'20)*, Portland, OR, USA, June 2020. ACM.

[5] C. Estan and J. F. Naughton. End-biased samples for join cardinality estimation. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 20, Atlanta, GA, USA, 2006. IEEE.

[6] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pages 615–629, San Francisco, CA, USA, 2016. ACM.

[7] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering (ICDE 2011)*, pages 984–994, Hannover, Germany, Apr. 2011. IEEE.

[8] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2013.

[9] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, San Francisco, CA, USA, 2016. ACM.

[10] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. faimgraph: High performance management of fully-dynamic graphs under tight memory constraints on the gpu. In *Proceedings of SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, Texas, USA, Nov. 2018. IEEE.

[11] N. Zhang, M. Ozsu, A. Aboulnaga, and I. Ilyas. Xseed: Accurate and fast cardinality estimation for xpath queries. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 61–61, 2006.