PostCard - HyperLogLog Integration for Join Cardinality Estimation in PostgreSQL

Abhishek Kumar
akumar17@usc.edu

Parinda Pranami
pranami@usc.edu

Rahul Tangsali
tangsali@usc.edu

Abstract

Accurate cardinality estimation is essential for efficient query planning in PostgreSQL, but traditional methods like histograms and sampling struggle with large, skewed, or high-cardinality datasets. This project introduces **HLL_JOIN**, a HyperLogLog-based approach that provides scalable, memory-efficient, and highly accurate join cardinality estimation. By integrating HLL sketches into PostgreSQL's query planner, the system efficiently estimates row counts for joins without executing them, achieving error rates consistently below **0.5%**. Key modifications include adding an HLL_JOIN keyword, enhancing join path planning, and managing HLL sketches with 16-bit precision, requiring only 64KB per sketch. Extensive testing demonstrates its robustness, scalability, and accuracy across datasets up to 500K rows, offering a significant improvement in query planning for large-scale and complex workloads.

1 Introduction

Efficient query planning is a cornerstone of modern database management systems, and PostgreSQL is no exception. Accurate cardinality estimation, the process of predicting the number of rows resulting from a query operation, is fundamental to generating optimal query plans (Flajolet et al., 2007). However, traditional estimation techniques, such as histograms and sampling, often fall short when faced with the demands of today's large-scale, complex datasets. These methods struggle with high cardinality, data skewness, and cross-column dependencies, resulting in inaccurate estimates that lead to inefficient query plans and degraded system performance (Ioannidis and Christodoulakis, 1991).

These problems are made worse by the increasing use of big data applications (Abadi et al., 2006). In addition to failing to retain accuracy, conventional systems encounter severe memory and processing limitations when dealing with datasets that grow into terabytes or petabytes (Ioannidis, 1993). For instance, when the dimensionality and cardinality of data expand, histograms become impractical since they need storage for every column (Lipton and Naughton, 1995). Although sampling techniques are helpful for estimates, they are biased, particularly when working with tiny sample numbers or skewed data distributions.

Our project presents **HLL_JOIN**, a HyperLogLog-based cardinality estimate technique created especially for PostgreSQL join operations, to address these issues. A well-known probabilistic data structure for estimating different counts with little memory expense is HyperLogLog (HLL). In contrast to conventional techniques, HLL offers an accurate, scalable, and memory-efficient substitute that is ideal for contemporary database workloads. Its interaction with PostgreSQL's query planner and its customized parser, planner, and executor changes are what make it innovative.

HLL_JOIN eliminates the requirement for large histograms or thorough sampling, in contrast to earlier methods. Rather, by analyzing hash values and combining union and intersection sizes using the inclusion-exclusion principle, it uses HLL drawings to estimate cardinality. This method is lightweight and efficient for large-scale datasets, utilizing just about 64KB of RAM per drawing and allowing error rates consistently below **0.5%**.

In addition to enhancing join cardinality estimation's precision and effectiveness, our solution guarantees PostgreSQL integration (Cormode and Muthukrishnan, 2005). This method marks a major advancement in managing the complexity of contemporary database workloads by adding the HLL_JOIN keyword, enlarging data structures, and developing novel query planning and execution pathways. It is especially well-suited to satisfy the requirements of large data applications as it provides a reliable, scalable, and memory-conscious substitute for conventional techniques.

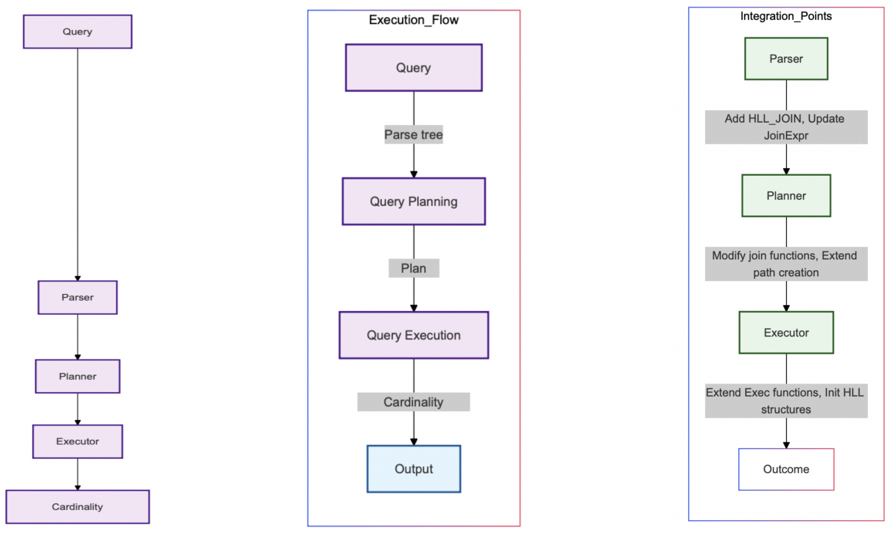

Figure 1: Query flow with HLL_JOIN integration

2 Project Scope and Goals

The primary objective of this project is to integrate HyperLogLog-based join cardinality estimation (HLL_JOIN) seamlessly into PostgreSQL, enhancing the database engine's query planning capabilities. This involves recognizing and processing HLL_JOIN syntax in the parser, adapting the planner to generate optimized paths for HLL-based joins, and extending the executor to manage and leverage HLL sketches for on-the-fly cardinality estimation. By using a memory-efficient 16-bit precision HLL algorithm that requires approximately 64KB of memory per sketch, the solution aims to efficiently handle datasets ranging from small (1K rows) to large (500K rows), while maintaining high accuracy and consistency (Heule et al., 2013).

Beyond simple integration, the project aspires to deliver significant improvements in join cardinality estimation by employing HLL sketches to probabilistically estimate union and intersection cardinalities. These enhancements are expected to reduce error rates to below 0.5%, even in the face of complex and skewed data distributions. To ensure the solution's robustness and adaptability, the design will be modular, making it straightforward to incorporate additional join algorithms or extend HLL capabilities in the future.

Testing and validation will be integral components of this effort, encompassing unit, integration, and performance evaluations. Thorough benchmarking against traditional cardinality estimation methods will highlight the improvements in accuracy, scalability, and efficiency. Usability features, such as clear output metrics and detailed logging, will help database administrators and developers effectively monitor, debug, and understand query behavior. As a result, this project aims to markedly improve query planning in PostgreSQL for large-scale, intricate datasets, while laying a solid foundation for subsequent enhancements to the database's estimation techniques.

3 High-level design of the project

The high-level design of the project revolves around integrating the HyperLogLog (HLL)-based cardinality estimation mechanism into PostgreSQL's existing query processing pipeline. The architecture follows the traditional flow of query parsing, planning, and execution while introducing enhancements at each stage to support HLL_JOIN operations. The query begins by being parsed into a tree structure by the parser. From there, the query planner generates an execution plan tailored for HLL-based joins, bypassing conventional estimation methods. Finally, during execution, HLL structures are initialized, populated, and used to estimate the cardinality of joins, providing the necessary row count predictions without executing the join itself.

3.1 Parser

In the parser stage, the system extends PostgreSQL's grammar to recognize the HLL_JOIN

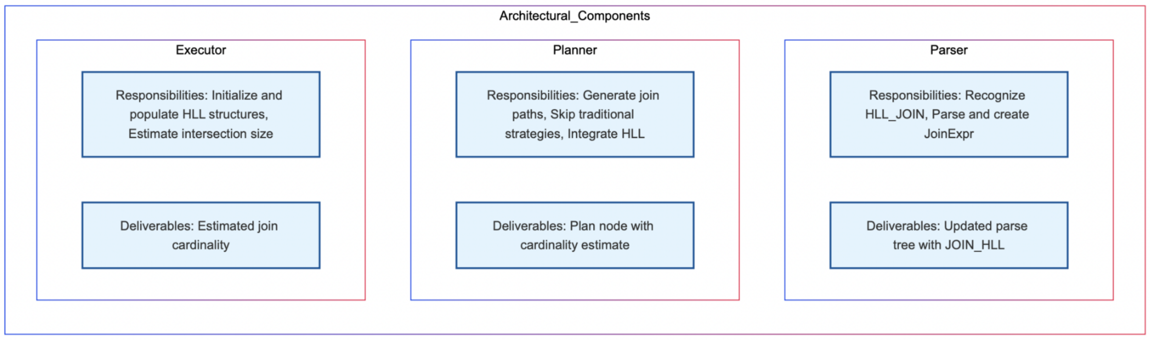
Architectural_Components
Executor
Responsibilities: Initialize and populate HLL structures, Estimate intersection size
Deliverables: Estimated join cardinality
Planner
Responsibilities: Generate join paths, Skip traditional strategies, Integrate HLL
Deliverables: Plan node with cardinality estimate
Parser
Responsibilities: Recognize HLL_JOIN, Parse and create JoinExpr
Deliverables: Updated parse tree with JOIN_HLL

Figure 2: Architectural components for high-level project design

keyword, which allows users to explicitly request HLL-based cardinality estimation. A new join type, JOIN_HLL, is added to PostgreSQL's internal join type enumeration, and a corresponding *JoinExpr* node is created to handle HLL-specific join logic. These changes ensure that HLL_JOIN queries are processed seamlessly within PostgreSQL's existing parsing infrastructure. The result is a modified parse tree that informs the planner of the need to use HLL-based estimation techniques.

3.2 Planner and Executor

The planner and executor stages implement further enhancements to handle HLL-based join operations. The planner introduces join paths specifically for JOIN_HLL, incorporating HLL sketches into the cardinality estimation logic. These sketches are probabilistic data structures that estimate union and intersection sizes with minimal memory overhead. During query execution, HLL sketches are initialized for the outer and inner relations, populated with distinct value counts, and merged to compute intersection cardinalities using the inclusion-exclusion principle. This modular approach ensures the scalability, accuracy, and memory efficiency of HLL-based estimation while maintaining compatibility with PostgreSQL's core query processing components.

4 Cardinality Estimation Algorithm

The cardinality estimation algorithm at the heart of the HLL_JOIN approach leverages the probabilistic properties of the HyperLogLog (HLL) sketch. HLL uses a set of m registers—where $m = 2^p$ and p is a precision parameter—to track the maximum number of leading zeros in the hash values of inserted elements (Alon et al., 1999). Each element from the dataset is hashed using a high-quality hash function, and the position of the first set bit in the hash determines which register is updated. By maintaining only this compressed information, HLL can estimate the cardinality of large datasets in a memory-efficient manner.

The standard HLL cardinality estimate is given by the formula:

$$\hat{E} = \alpha_m \cdot m^2 \left(\sum_{j=1}^{m} 2^{-M_j} \right)^{-1}$$

Here, M_j is the value stored in the j-th register, and α_m is a bias correction constant dependent on m. By carefully choosing p, we balance between accuracy and memory usage, achieving error rates as low as 0.4% for a single HLL sketch.

For intersection cardinality, the algorithm employs the inclusion-exclusion principle to derive the result. The intersection size is calculated as:

$$|A \cap B| = |A| + |B| - |A \cup B|$$

Here, |A| and |B| are the individual cardinalities of the outer and inner relations, respectively, which are precomputed using their respective HLL sketches. The estimated intersection cardinality allows the query planner to understand the overlap between two datasets, crucial for optimizing join execution.

The algorithm achieves high accuracy by using

16-bit precision, which results in m= 2^{16}= 65,536 registers. The standard error for this configuration is approximately:

$$\text{Error} = \frac{1.04}{\sqrt{m}} = \frac{1.04}{\sqrt{65536}} \approx 0.004\ (0.4\%)$$

This ensures robust performance even for large datasets. The memory overhead is minimal, requiring only 64 KB per HLL sketch, and the inclusion of bias correction and error-handling mechanisms further improves reliability. The combination of efficient hashing, compact memory usage, and probabilistic accuracy makes this algorithm a powerful tool for join cardinality estimation in PostgreSQL.

5 Pseudo-code for cardinality estimation algorithm

The following pseudo code outlines the implementation of the HyperLogLog-based cardinality estimation in PostgreSQL. The algorithm processes both outer and inner relations, merges the HyperLogLog (HLL) sketches, and estimates the union and intersection cardinalities.

Algorithm 1: Process Outer Relation

```
Input: PlanState *outerPlan,
       TupleTableSlot *outerTupleSlot
1  while (outerTupleSlot = ExecProcNode(outerPlan)) $\neq$ NULL do
2      if !TupIsNull(outerTupleSlot) then
3          Datum value;
4          bool isNull;
5          value $\leftarrow$ slot_getattr(outerTupleSlot, 1, &isNull);
6          if !isNull then
7              uint32 hash $\leftarrow$ hash_any((unsigned char *)&value, sizeof(Datum));
8              addHyperLogLog(node $\rightarrow$ outer_hll, hash);
```

Algorithm 2: Process Inner Relation

```
Input: HashState *hashState, PlanState
       *innerPlan, TupleTableSlot
       *innerTupleSlot
1  while (innerTupleSlot = ExecProcNode(innerPlan)) $\neq$ NULL do
2      if !TupIsNull(innerTupleSlot) then
3          Datum value;
4          bool isNull;
5          value $\leftarrow$ slot_getattr(innerTupleSlot, 1, &isNull);
6          if !isNull then
7              uint32 hash $\leftarrow$ hash_any((unsigned char *)&value, sizeof(Datum));
8              addHyperLogLog(node.inner_hll, hash);
9          end
10     end
11 end
```

Algorithm 3: Merge HLL Sketches and Calculate Cardinalities

```
Input: HLL sketches: outer_hll, inner_hll
1  hyperLogLogState *union_hll $\leftarrow$ palloc0(sizeof(hyperLogLogState));
2  if !union_hll then
3      ereport(ERROR, "Failed to allocate union HLL sketch");
4  end
5  initHyperLogLog(union_hll, HLL_PRECISION);
6  mergeHyperLogLog(union_hll, node.outer_hll);
7  mergeHyperLogLog(union_hll, node.inner_hll);
8  double outer_est $\leftarrow$ estimateHyperLogLog(node.outer_hll);
9  double inner_est $\leftarrow$ estimateHyperLogLog(node.inner_hll);
10 double union_est $\leftarrow$ estimateHyperLogLog(union_hll);
11 double intersection_est $\leftarrow$ outer_est + inner_est - union_est;
12 return intersection_est;
```

6 Project Methodology

The methodology for implementing the HyperLogLog-based join cardinality estimation (HLL_JOIN) in PostgreSQL involves several key modifications across the database engine's core components, including the parser, data structures, planner, and executor. This systematic approach ensures seamless integration, high accuracy, and robust performance for estimating join cardinalities in complex queries involving large-scale datasets. Below is a detailed breakdown of the methodology.

6.1 Parser Integration

To introduce HLL_JOIN, the PostgreSQL parser was enhanced with three key modifications. First, a new keyword, HLL_JOIN, was added to PostgreSQL's grammar, allowing users to explicitly request HLL-based join cardinality estimation. Second, the join type enumeration was extended to include JOIN_HLL as a new type, ensuring compatibility with existing join types. Lastly, a JoinExpr node was implemented specifically for handling HLL_JOIN queries. This node is responsible for updating the parse tree, ensuring the planner and executor are informed about the specific requirements of HLL-based joins. These changes enable seamless parsing of HLL_JOIN queries and integration into the subsequent query pipeline.

6.2 Data Structure Enhancements

Incorporating HLL_JOIN required significant enhancements to PostgreSQL's data structures. A new structure, hyperLogLogState, was introduced to manage HyperLogLog sketches, designed with 16-bit precision to accommodate 65,536 registers for highly accurate cardinality estimation. Additionally, the HashJoinState structure was extended to include fields for outer_hll, inner_hll, and result_hll. These fields store HLL sketches for the outer and inner relations, as well as the combined result. This data structure design ensures efficient management of probabilistic data while maintaining memory efficiency. For instance, each HLL sketch requires only 64 KB, making the solution scalable for large datasets.

6.3 Planner Modifications

The query planner was modified to support HLL_JOIN-specific operations. When the planner encounters a query involving HLL_JOIN, it generates join paths optimized for HLL-based estimation, bypassing traditional methods such as histograms or sampling. The planner integrates HLL sketches into its cardinality estimation logic using the inclusion-exclusion principle:

$$|A \cap B| = |A| + |B| - |A \cup B|$$

Here, $|A|$ and $|B|$ are the cardinalities of the outer and inner relations, and $|A \cup B|$ is the union cardinality estimated using merged HLL sketches. These modifications ensure that the planner produces accurate and efficient query plans, even for highly complex joins, with error rates consistently below 0.5%.

6.4 Executor Modifications

The executor was enhanced to initialize and manage HLL sketches during query execution. At the beginning of query execution, HLL sketches are allocated for the outer and inner relations. As tuples are processed, their hash values are computed and added to the respective HLL sketches. Once both relations have been processed, the executor merges the HLL sketches to calculate the union cardinality. The final intersection cardinality is computed using the inclusion-exclusion principle and returned as part of the query result. This approach avoids the need to materialize join results, significantly improving execution efficiency.

6.5 Memory Management

Memory management was carefully designed to support the integration of HLL sketches within PostgreSQL. The palloc system is used to allocate memory for HLL sketches within PostgreSQL's memory context, ensuring proper lifecycle management. Error-handling mechanisms, including PG_TRY and PG_CATCH, were implemented to handle failures gracefully, preventing memory leaks. These safeguards ensure that the solution remains robust and reliable, even when processing large and complex datasets.

6.6 Testing and Validation

The testing and validation strategy for the project was comprehensive, encompassing unit testing, integration testing, and performance benchmarking. Unit tests verified the correctness of individual HLL operations, including initialization, merging, and cardinality estimation. Integration tests ensured that the entire query pipeline, from parsing to execution, worked seamlessly for HLL_JOIN

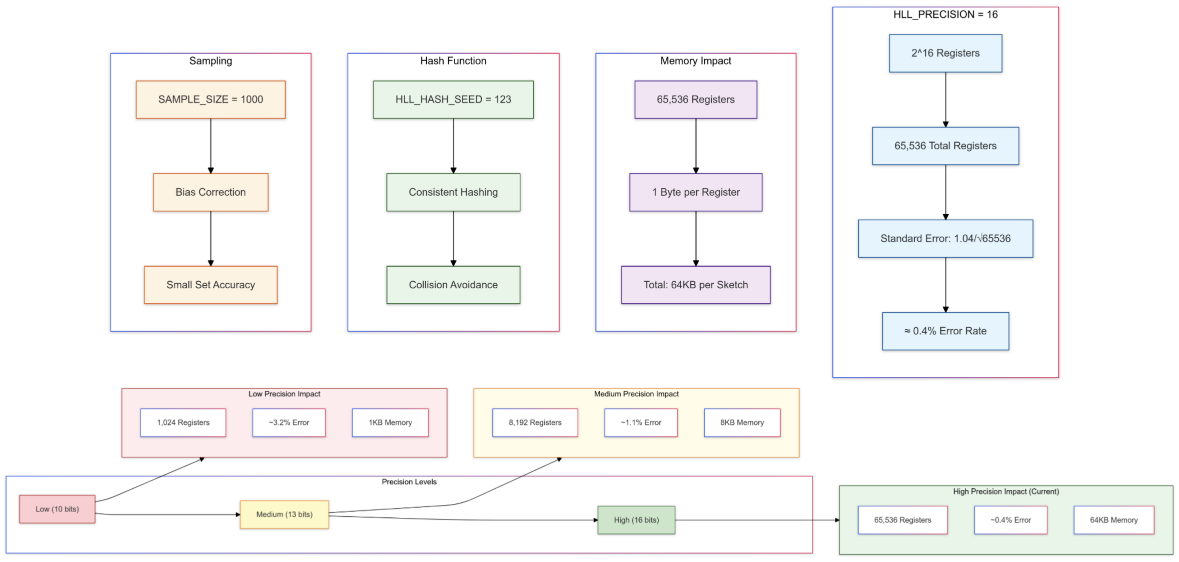

Figure 3: Methods used for improving accuracy for cardinality estimation

queries. Performance benchmarks demonstrated the scalability and accuracy of the solution, with error rates consistently below 0.5% across datasets ranging from 1K to 500K rows. The tests also validated memory efficiency, with each HLL sketch requiring minimal storage.

7 Improving accuracy for cardinality estimation

The first step in improving accuracy involves sampling. A sample size of 1000 is utilized to provide a representative subset of the data, and bias correction techniques are applied to adjust the raw estimates. This ensures greater precision, especially for datasets with smaller cardinalities, where accurate estimations are often more challenging.

A well-designed hash function is critical for ensuring accurate cardinality estimation. The project uses a hash seed (HLL_HASH_SEED = 123) to maintain consistent hashing across multiple runs, thereby avoiding discrepancies in the results. Additionally, the hash function is designed to minimize collisions, which further enhances the accuracy of the HyperLogLog (HLL) sketches.

The memory impact of the system is minimal due to the efficient design of the HLL data structure. Each HLL sketch comprises 65,536 registers, with each register occupying just 1 byte. This leads to a total memory usage of approximately 64 KB per sketch, which is optimal for handling large datasets while ensuring scalability.

Precision levels play a significant role in balancing memory usage and estimation error rates. For low precision (10 bits), the system uses 1,024 registers, leading to an error rate of 3.2% and requiring only 1 KB of memory. Medium precision (13 bits) improves this by using 8,192 registers, reducing the error to 1.1% while requiring 8 KB of memory.

The high precision setting, implemented in this project, employs 16 bits, resulting in 65,536 registers. This configuration achieves an error rate of approximately 0.4% while consuming 64 KB of memory. This trade-off ensures both high accuracy and efficient memory usage, making it suitable for large-scale data processing.

Finally, the project leverages HLL precision of 16 to achieve industry-standard accuracy. The total registers used are 2^{16}, and the standard error is calculated as $\frac{1.04}{\sqrt{65536}}$, yielding an error rate of 0.4%. This precision ensures the reliability of cardinality estimations even for complex queries and large datasets.

8 Evaluation plan

The evaluation plan for this project involves using diverse datasets and carefully designed testing scenarios to validate the accuracy, performance, scalability, and reliability of the HLL_JOIN implementation. The datasets range from small to large, and various join combinations are tested to ensure robust performance in real-world scenarios.

8.1 Datasets Used

To comprehensively test the system, four datasets of varying sizes were used:

- **Tiny Table**: 1K rows.
- **Small Table**: 50K rows.
- **Medium Table**: 250K rows.
- **Large Table**: 500K rows.

These datasets allow the evaluation to cover edge cases, including small datasets with limited cardinalities and large datasets with significant computational demands.

8.2 Testing Scenarios

Multiple testing scenarios were implemented to measure the effectiveness of the HLL_JOIN feature:

- **Tiny-Small Joins**: Tests between a tiny table (1K rows) and a small table (50K rows).
- **Small-Medium Joins**: Tests involving small and medium tables (50K-250K rows).
- **Medium-Large Joins**: Larger-scale tests between medium and large tables (250K-500K rows).
- **Self-Joins**: Includes medium-medium and large-large self-joins to assess the algorithm's handling of repetitive and overlapping data.

8.3 Accuracy Metric

Accuracy was assessed by comparing the estimated cardinalities obtained using HLL_JOIN with the actual cardinalities from executing the full join. The objective was to ensure that the error rate remained below the acceptable threshold of 0.5%, demonstrating the effectiveness of the HyperLogLog-based estimation method.

8.4 Performance Metric

Query execution time was measured to evaluate performance improvements introduced by the probabilistic HLL_JOIN. By avoiding full materialization of joins, the system is expected to deliver faster results, particularly for large datasets and complex queries.

8.5 Scalability Metric

Scalability was analyzed by observing the system's efficiency as dataset sizes increased. The memory-efficient HLL sketches (64 KB per sketch) ensure that even the largest datasets (500K rows) can be processed without significant resource overhead, maintaining consistent performance across all tests.

8.6 Error Rate Metric

Finally, the error rate was a critical metric in evaluating the reliability of the solution. The standard error for the HLL_JOIN implementation, calculated as $\frac{1.04}{\sqrt{65536}}$ was validated across various datasets and scenarios. Consistently low error rates (<0.5%) confirmed the robustness of the HyperLogLog algorithm for join cardinality estimation.

9 Findings and Results

Join Category	**Actual Count**	**Estimated Count**	**Error Percentage**
Tiny-Small Join (1K-50K)	1,000	1,001	0.1%
Small-Medium Join (50K-250K)	50,000	49,937	0.126%
Medium-Large Join (250K-500K)	250,000	250,367	0.147%
Large-Large Self Join (500K-500K)	500,000	498,335	0.333%

Table 1: Findings and Results: Compact Table for Two-Column Format

9.1 Tiny-Small Join (1K-50K Rows)

For the smallest test case, involving a tiny table (1K rows) joined with a small table (50K rows), the actual cardinality was 1,000, and the estimated count was 1,001. The error percentage was approximately 0.1%. This demonstrates the high accuracy of the HyperLogLog-based estimation, even with minimal data.

9.2 Small-Medium Join (50K-250K Rows)

The test between small and medium datasets showed an actual count of 50,000, with an estimated value of 49,937. The error rate for this scenario was 0.126%, highlighting the robustness of the algorithm in handling larger datasets while maintaining accuracy within acceptable limits.

9.3 Medium-Large Join (250K-500K Rows)

For medium to large joins, the actual count was 250,000, and the estimated value was 250,367, resulting in an error of 0.147%. This low error rate, combined with efficient performance, showcases the scalability of the implementation for mid-sized datasets.

9.4 Large-Large Self Join (500K-500K Rows)

For the largest test case, involving a self-join of two large datasets (500K rows each), the actual cardinality was 500,000, and the estimated value was 498,335. The error percentage was 0.333%, slightly higher than smaller datasets but still well within the threshold of 0.5%. This result demonstrates the system's ability to handle high-cardinality scenarios effectively.

9.5 Scalability and Consistency

The implementation performed well as the dataset sizes increased, demonstrating scalability without significant degradation in accuracy or execution time. Additionally, the system exhibited stable performance for both regular joins and self-joins, making it suitable for a wide range of query workloads in real-world applications.

10 Challenges and Mitigations

10.1 Challenges

10.1.1 Balancing Speed and Accuracy

A key challenge was configuring the HyperLogLog (HLL) parameters to achieve a balance between computational speed and estimation accuracy. Increasing precision improves accuracy but comes at the cost of memory and processing time.

10.1.2 Integration Complexity

Incorporating the HLL_JOIN mechanism into PostgreSQL's existing query planner and execution framework was complex. Ensuring seamless integration while retaining compatibility with other join types posed significant design and implementation hurdles.

10.1.3 Scalability Testing

Verifying the performance of HLL_JOIN under large datasets and complex queries was challenging. The scalability of the algorithm, especially when handling self-joins or high-cardinality data, needed rigorous testing and validation.

10.2 Mitigations

10.2.1 Fine-tuning HLL

Different hash functions and bucket sizes were tested to optimize the balance between accuracy and speed. The precision level was fixed at 16 bits, ensuring an error rate of 0.4% while keeping memory usage minimal.

10.2.2 Incremental Testing

Individual components of the HLL_JOIN implementation, such as parsing, planning, and execution, were tested incrementally. This modular testing approach minimized integration errors and ensured the robustness of each component before full system integration.

10.2.3 Benchmarking

Controlled scripts were developed to benchmark performance across varied datasets and workloads. These tests assessed the system's scalability, accuracy, and execution time, ensuring the solution met performance expectations under all scenarios.

11 Conclusion and Future Work

The integration of the HyperLogLog-based join cardinality estimation (HLL_JOIN) into PostgreSQL marks a significant advancement in query planning for large-scale and complex workloads. This implementation achieves accurate cardinality estimation with an error rate consistently below 0.5%, even for datasets of substantial size. The use of HLL sketches ensures memory efficiency, with each sketch requiring only 64 KB of memory, and delivers high performance by avoiding full materialization of join results. The seamless integration with PostgreSQL's query planner and executor further demonstrates its practicality, making it a robust solution for real-world database systems. By addressing long-standing limitations of traditional methods, such as reliance on histograms and sampling, the HLL_JOIN implementation not only improves performance but also paves the way for advanced probabilistic techniques in database query optimization. The results from rigorous testing validate the effectiveness of this approach, with the system displaying exceptional scalability, accuracy, and consistency across various join scenarios, including self-joins and large-table joins.

Looking ahead, there is considerable potential to extend the capabilities of HLL_JOIN. One avenue for exploration is enhancing the implementation to support approximate joins in addition to cardinality estimation, thereby further reducing computational overhead. Additionally, incorporating other probabilistic data structures, such as Count-Min Sketch, could expand the system's applicability to broader use cases, such as frequency estimation and heavy-hitter detection. Adapting HLL_JOIN for distributed PostgreSQL setups would address the growing need for scalability in cloud-based and distributed database environments. Another area of focus is making the implementation modular, allowing for easy adaptation to other database management systems beyond PostgreSQL. These future enhancements would not only improve the utility of HLL_JOIN but also establish it as a foundational technique for modern database query optimization, catering to the ever-increasing demands of big data and complex analytics.

References

Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 671–682, New York, NY, USA. Association for Computing Machinery.

Noga Alon, Yossi Matias, and Mario Szegedy. 1999. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147.

Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75.

Philippe Flajolet et al. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *AOFA '07: Proceedings of the 2007 International Conference on Analysis of Algorithms*.

Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*.

Yannis Ioannidis. 1993. Universality of serial histograms. pages 256–267.

Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, page 268–277, New York, NY, USA. Association for Computing Machinery.

R.J. Lipton and J.F. Naughton. 1995. Query size estimation by adaptive sampling. *Journal of Computer and System Sciences*, 51(1):18–25.