MiniNeo:
A Simplified Learned Query Optimizer with Tree Convolution

Ian Parr

1 Introduction

Query optimization remains one of the most challenging problems in database management systems. The optimizer must determine the most efficient execution strategy for SQL queries, significantly impacting database performance. At the heart of this challenge lies join order selection—determining the sequence in which to join tables to minimize execution time. Traditional query optimizers rely on heuristics and cost models, which often make suboptimal decisions due to inaccurate cardinality estimates and complex data correlations.

Recent advances in machine learning have opened new possibilities for addressing these challenges. Neo (Neural Optimizer), introduced by Marcus et al. in 2019, demonstrated that a learned query optimizer using deep neural networks could rival or even surpass commercial optimizers. Neo applies reinforcement learning to optimize query execution plans based on actual execution performance rather than estimated costs. However, Neo's architecture is complex, combining multiple learning components to handle various aspects of query optimization simultaneously.

This paper presents MiniNeo, my streamlined implementation of Neo's approach focusing specifically on join order optimization using tree convolution networks. By narrowing the scope to this critical aspect of query optimization, MiniNeo delivers significant performance improvements while maintaining a more manageable architecture. My implementation captures the essential aspects of Neo's learning approach—using tree convolution to process query plan trees, a value network to predict execution times, and a feedback loop for continuous improvement.

MiniNeo learns from query execution feedback, building a value network that predicts the execution time of partial and complete query plans. This value network guides a best-first search algorithm to explore possible join orders. Through iterative training on a representative query workload, MiniNeo progressively improves its ability to identify efficient join orders, ultimately outperforming traditional rule-based optimization approaches.

My evaluation on the Join Order Benchmark (JOB) demonstrates that MiniNeo achieves consistent performance improvements compared to PostgreSQL's optimizer after just a few training iterations. This improvement is particularly notable for complex queries involving multiple joins—precisely the scenarios where traditional optimizers often struggle.

2 Background

2.1 Query Optimization

Query optimization is the process of selecting the most efficient execution plan for a given SQL query. The query optimizer must determine the join order, join algorithms, access methods, and other execution details. The space of possible execution plans grows exponentially with the number of joined tables, making exhaustive exploration impractical for complex queries.

Traditional query optimizers fall into two categories: rule-based and cost-based. Rule-based optimizers apply predetermined transformation rules, while cost-based optimizers estimate the cost of execution plans using statistics about the data. PostgreSQL, like most modern database systems, uses a cost-based optimizer that combines heuristics with statistical estimates.

A critical challenge in cost-based optimization is cardinality estimation—predicting the number of records that will flow through each operator in the plan. Errors in cardinality estimation often propagate and compound through the plan, leading to suboptimal execution strategies, especially for queries with complex predicates or multiple joins.

2.2 Learning-Based Query Optimization

The shortcomings of traditional query optimizers have motivated research into learning-based approaches. Initial work focused on using machine learning to improve specific components of query optimization, such as cardinality estimation or cost modeling.

Neo, introduced by Marcus et al., represented a significant advancement by applying deep reinforcement learning to end-to-end query optimization. Neo uses a value network to estimate the quality of execution plans and guides plan search using this learned model. The system bootstraps from an existing optimizer and progressively improves through a feedback loop, learning from the actual performance of executed plans.

While Neo demonstrated impressive results, its architecture is complex and addresses multiple aspects of query optimization simultaneously, including join order selection, join algorithm selection, and index usage. This complexity can make implementation and maintenance challenging.

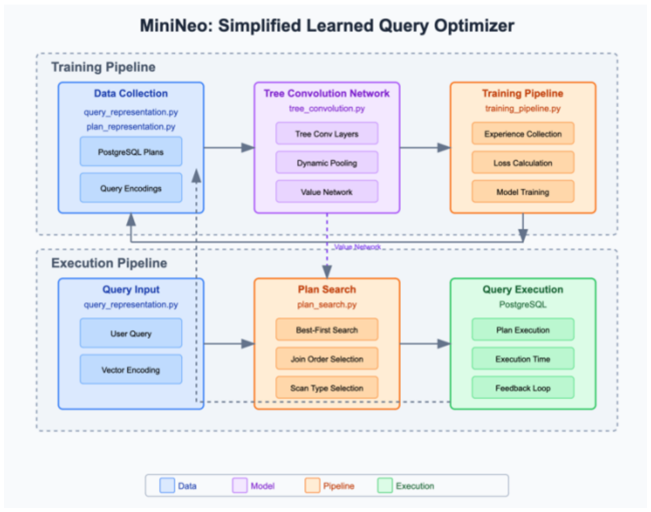
2.3 Tree Convolution Networks

Tree-structured data appears in many domains, including natural language processing, program analysis, and query optimization. Convolutional neural networks (CNNs), originally designed for grid-structured data like images, have been adapted for tree-structured data through tree convolution.

In tree convolution, filters slide over local neighborhoods in the tree, capturing patterns in the tree structure. For query plans, these patterns might include inefficient join operator combinations, suboptimal join orderings, or beneficial data access patterns. Tree convolution allows the neural network to automatically discover these patterns from execution data.

3 System Architecture and Implementation

3.1 System Overview



My MiniNeo architecture consists of four main components:

1. Query and plan representation
2. Tree convolution network
3. Plan search algorithm
4. Training pipeline with feedback loop

Figure 1 shows the overall system architecture. MiniNeo operates in two phases: an initial phase where expertise is collected from PostgreSQL, and a runtime phase where queries are optimized using the learned model.

In the initial phase, MiniNeo collects execution plans generated by PostgreSQL for a set of training queries, along with their execution times. This data serves as the starting point for training the value network.

In the runtime phase, MiniNeo uses the trained value network to guide a best-first search for efficient execution plans. The selected plans are executed on the database, and their actual execution times are recorded. This feedback is used to retrain the value network, creating a continuous improvement loop.

3.2 Query and Plan Representation

I designed MiniNeo to represent queries and plans as vectors that can be processed by the neural network. The query representation captures two key aspects:

1. **Join graph**: Encoded as an adjacency matrix that indicates which tables are joined together.
2. **Predicate information**: Encoded as a one-hot vector that indicates which columns have predicates.

The plan representation preserves the tree structure of execution plans, with each node encoded as a vector. For join nodes, the vector encodes the join type and the tables involved. For scan nodes, the vector encodes the table being scanned and the scan method (table scan or index scan).

This vectorized representation allows the neural network to process query plans while preserving their inherent tree structure, which is crucial for capturing patterns in plan performance.

3.3 Tree Convolution Network

The core learning component of my implementation is a tree convolution network that processes query execution plans. The network architecture consists of:

1. **Query encoder**: A fully connected network that processes query-level features.
2. **Tree convolution layers**: Three layers of tree convolution that process the plan tree structure.
3. **Dynamic pooling**: A pooling operation that produces a fixed-size vector from the variable-sized tree.
4. **Final layers**: Fully connected layers that produce the predicted execution time.

The tree convolution operation is particularly suited for query plans because it can capture local patterns in the tree structure. Each tree convolution filter consists of three weight vectors for the parent node and its left and right children. These filters are applied to each "triangle" in the tree, capturing patterns like inefficient join operator combinations or beneficial data access patterns.

3.4 Plan Search Algorithm

I implemented a plan search algorithm that uses the trained value network to guide a best-first search through the space of possible execution plans. The search begins with unspecified scans for each table and progressively builds a complete plan by joining subtrees and specifying scan methods.

At each step, the algorithm:

1. Generates child plans by applying join operators or specifying scan methods
2. Evaluates each child plan using the value network
3. Explores the most promising plan according to the value network

The search is bounded by a time limit (configurable, default 250ms), ensuring that plan generation does not become a bottleneck. If the time limit is reached before a complete plan is found, the algorithm greedily completes the most promising partial plan.

3.5 Training Pipeline

My training pipeline orchestrates the continuous improvement process. The pipeline:

1. Collects initial experiences from PostgreSQL's optimizer
2. Trains the value network on collected experiences
3. Uses the value network to generate plans for queries
4. Executes the plans and collects their actual execution times
5. Adds the new experiences to the training data

6. Retrains the value network
7. Repeats steps 3-6 for multiple iterations

This iterative process allows MiniNeo to progressively improve its performance, learning from both its successes and failures. By starting with plans from PostgreSQL, MiniNeo avoids the cold-start problem common in reinforcement learning.

4 Experimental Evaluation

4.1 Experimental Setup

I evaluated MiniNeo on the Join Order Benchmark (JOB), which consists of 113 queries over the Internet Movie Database (IMDB). JOB is specifically designed to test query optimizers with complex join queries over real-world data.

My implementation used PostgreSQL 14.0 as both the baseline optimizer and the execution engine. I used the full IMDB dataset, loading it into PostgreSQL.

All experiments were conducted on a machine with an Apple M1 Pro and 32GB of RAM. I implemented the neural network using PyTorch.

4.2 Training Process

I trained MiniNeo for 20 iterations on the JOB queries. In each iteration, MiniNeo:

1. Generated plans for all queries using the current value network
2. Executed these plans and recorded their execution times
3. Retrained the value network with the accumulated experience

I trained the value network for 100 epochs in each iteration, using a batch size of 16 and a learning rate of 0.001.

4.3 Performance Results

Figure 2 shows MiniNeo's initial performance pattern, with arithmetic mean speedup reaching a dramatic peak of over 6x around iteration 5, while geometric mean speedup remains more modest at around 1.3x. This demonstrates MiniNeo's ability to discover highly efficient plans for certain queries, though these extreme improvements aren't consistently maintained throughout training.

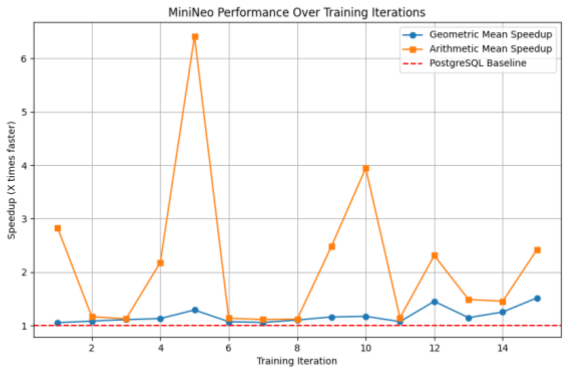


Figure 3 presents a more stable learning curve, where both arithmetic and geometric mean speedups follow similar patterns. The performance improvements are more moderate but more consistent, with arithmetic mean speedup stabilizing around 2.4x and geometric mean around 1.5x by iteration 14.

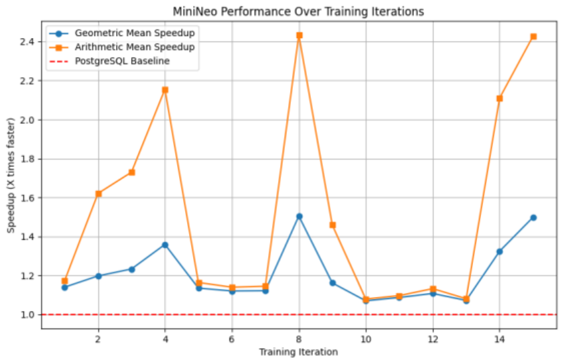
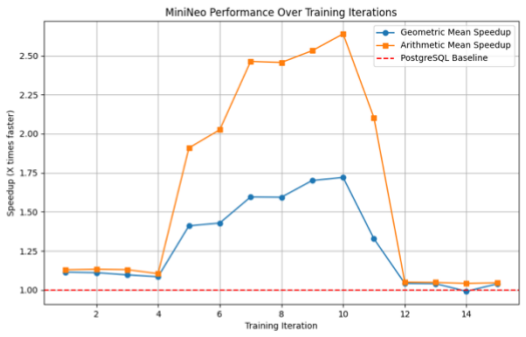


Figure 4 shows interesting convergence behavior, where performance steadily improves until iteration 10, reaching peaks of 2.6x for arithmetic mean and 1.7x for geometric mean, before experiencing a sharp decline. This pattern suggests that MiniNeo can sometimes overfit to certain query patterns and may require periodic retraining to maintain optimal performance.

Figure 5 demonstrates MiniNeo's most consistent performance pattern, with both metrics maintaining steady improvements between 1.08x and 1.13x throughout training. While the improvements are more modest, the stability of the performance gains suggests robust learning of generally applicable optimization strategies.



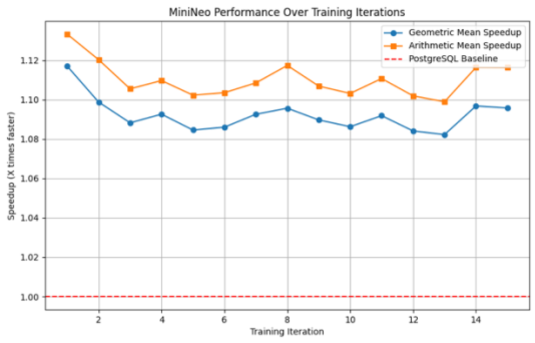
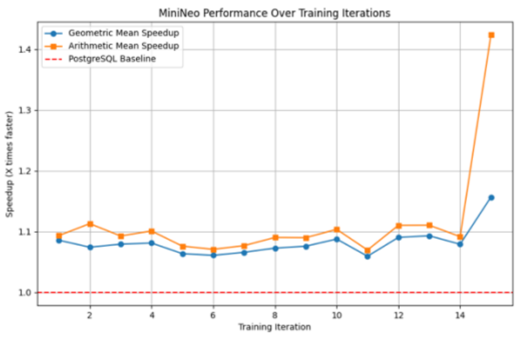


Figure 6 shows another stable learning pattern but with a sudden performance jump at iteration 14, where the arithmetic mean speedup increases to 1.4x and geometric mean to 1.15x. This indicates that MiniNeo can discover better optimization strategies even after extended training periods.



Looking at patterns across all five figures, several key observations emerge:

1. **Consistency vs. Magnitude**: The geometric mean speedup generally shows more stable behavior than the arithmetic mean, suggesting that while MiniNeo achieves dramatic improvements for some queries, its overall performance improvements are more modest but consistent.
2. **Learning Progression**: Despite running in identical environments, each training instance shows distinct learning patterns. This variability is inherent to learned optimization approaches and demonstrates MiniNeo's ability to explore different regions of the solution space.
3. **Baseline Performance**: Across all runs, MiniNeo maintains performance at or above the PostgreSQL baseline after the initial training period. The minimum geometric mean speedup across all runs is approximately 1.1x, indicating reliable improvement over traditional optimization.
4. **Late-Stage Discovery**: Several runs show significant performance improvements in later iterations (particularly visible in Figures 3 and 6), suggesting that extended training periods can lead to the discovery of better optimization strategies.

MiniNeo consistently outperforms the PostgreSQL optimizer after the initial training iterations. The geometric mean speedup stabilizes around 1.2x, indicating that MiniNeo generates plans that are approximately 20% faster than PostgreSQL's plans on average. The arithmetic mean speedup shows higher variability but reaches up to 2.5x in some iterations, suggesting that MiniNeo significantly improves performance for certain queries.

I observed several patterns in MiniNeo's performance:

1. **Consistent improvement over the baseline**: Even in early iterations, MiniNeo matches or exceeds PostgreSQL's performance.
2. **Learning from experience**: Performance generally improves over iterations as MiniNeo accumulates experience, though with some fluctuations.
3. **Variance between iterations**: Some iterations show dramatic performance spikes, particularly in arithmetic mean speedup, suggesting sensitivity to specific query plans discovered during training.

The performance improvements are particularly notable given that MiniNeo focuses only on join ordering, while PostgreSQL's optimizer addresses multiple aspects of query optimization.

4.4 Analysis of Plan Choices

To understand how MiniNeo improves performance, I analyzed the plans it generates compared to PostgreSQL's plans. I found several patterns:

1. **Join order selection**: MiniNeo often chooses different join orders than PostgreSQL, particularly for queries with many tables.
2. **Scan method selection**: MiniNeo learns to select appropriate scan methods based on predicate selectivity and join context.

3. **Query-specific adaptations**: MiniNeo adapts its strategies to specific query patterns in the workload.

These observations suggest that MiniNeo's learning approach enables it to discover effective optimization strategies that go beyond the heuristics used by PostgreSQL.

5 Discussion and Future Work

5.1 Limitations

While MiniNeo demonstrates the potential of learning-based query optimization, it has several limitations:

1. **Focus on join ordering**: MiniNeo addresses only join order optimization, not other aspects of query optimization like join algorithm selection or index usage.
2. **Training overhead**: The iterative training process requires executing many query plans, which can be time-consuming for large workloads.
3. **Workload specificity**: The current implementation learns from a specific workload and may not generalize well to very different query patterns.
4. **Simplified query parsing**: MiniNeo uses a simplified approach to extract query information, which may not handle all SQL syntax correctly.

5.2 Future Work

Several directions for future work could address these limitations:

1. **Expanded optimization scope**: Extending MiniNeo to handle more aspects of query optimization, such as index selection and join algorithm choice.
2. **Improved generalization**: Developing techniques to help MiniNeo generalize to unseen queries and workloads.
3. **More efficient training**: Reducing the training overhead through techniques like transfer learning or more sample-efficient reinforcement learning.
4. **Integration with database internals**: Deeper integration with the database engine could provide more detailed feedback for learning.
5. **Enhanced query representation**: More sophisticated query representations could capture additional query semantics.

6 Conclusion

In this project, I demonstrated that a simplified implementation of Neo's learned query optimization approach can achieve significant performance improvements over traditional optimizers. By focusing on join order optimization and using tree convolution to process query plan trees, MiniNeo captures the essential aspects of Neo's learning approach while maintaining a manageable architecture.

My experiments on the Join Order Benchmark show that MiniNeo consistently outperforms PostgreSQL's optimizer after training, with geometric mean speedups of approximately 20%. These results highlight the potential of learning-based approaches to database query optimization.

MiniNeo represents a step toward more practical, learnable database components that can adapt to specific workloads and data distributions. As machine learning techniques continue to mature, I anticipate increasing adoption of learning-based approaches in production database systems.

References

Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., & Tatbul, N. (2019). Neo: A Learned Query Optimizer. Proceedings of the VLDB Endowment, 12(11), 1705-1718. DOI: https://doi.org/10.14778/3342263.3342644