International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 6, Issue 2, 153-160, 2025
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V6I2P117

Original Article

Generative AI–Enabled Intelligent Query Optimization for Large-Scale Data Analytics Platforms

Dinesh Babu Govindarajulunaidu Sambath Narayanan
Independent Researcher, USA.

Received On: 05/04/2025 ***Revised On: 14/04/2025*** ***Accepted On: 30/04/2025*** ***Published On: 20/05/2025***

Abstract - *The immense scale of data volume in the distributed cloud infrastructures has magnified the computational intensity of query processing in extensive analytics system like Data Lake, distributed SQL systems and real-time data warehouses. The rule-based and cost-based algorithm-based traditional query optimization methods are becoming inadequate with dynamics, heterogeneous sources of data, and unpredictable execution conditions. The study presents an Intelligent Query Optimization Framework that relies on an AI to automatically rewrite the queries, optimize the plans, and execute them dynamically using the Generative Artificial Intelligence (Gen-AI). The proposed system term employs reinforcement learning (RL), natural language processing (NLP), and deep neural architecture to produce optimized query schemes, approximate the best implementation plans, decrease latency and enhance the throughput in gigacursing assemblies. Workloads, execution history of queries and run-time performance metrics on the distributed systems like Apache Spark, Presto, and Snowflake are all used to train the system. The research contributions consist of: (i) a synthesis mechanism to generate query plans through the use of transformer-based models, (ii) modeling to predict the costs of the workload, (iii) a multi-objective optimization scheme that minimizes the execution time, resource consumption, and cost of data transfer, and (iv) a hybrid architecture in which a batch and a streaming analytics is executed. Extensive experimental findings using the TPC-H benchmark show how it improves performance by 41%-percentage change in query latency, 32-percentage changes in throughput, and 27-percentage changes in memory usage as compared to state-of-the-art optimizers. The presented framework can be scaled, flexible and proficient in the changing data environment, which is a remarkable breakthrough in intelligent data analytics.*

Keywords - *Generative AI, Query Optimization, Big Data Analytics, Machine Learning, Cost-Based Optimization, Reinforcement Learning, Distributed Databases, Data Lakes.*

1. Introduction
1.1. Background
The contemporary digital ecosystem produces a never before seen amount of heterogeneous data of variety of authors like social media communications, IoT sensor data streams, business transactions of an enterprise, and data-intensive scientific applications. [1-3] Analytical query engines are becoming more popular in organizations as a way of deriving useful information on this ever-growing informational terrain. These engines should be able to scale the complex SQL-like query complexity that would include multi-table joins, nested subqueries and aggregations, real-time filtering of enormous data-sets maintained in scattered surroundings. A rule-based and cost-based approach to optimizing databases, scaling into zettabytes, cannot sustain performance in large scale data sets since its approach relies on a set of fixed assumptions concerning data properties and system behavior. In the meantime, highly changeable cloud environments, fluctuations of workloads, and data distributions require adjustment upkeep instead of optimal adjustment. Less than ideal implementation strategies do not only raise query responses but also propagate unjustified computation, network transmission and storage expenses, which is most vital in contemporary pay-as-you-use cloud design. These issues make it clear that autonomous, intelligent query optimization strategies are required that have the ability to learn through execution patterns, infer through structure and semantics of SQL queries, and improve their performance over time. This mounting stress on extrapolating analytical efficiency continues to drive the desire to find a solution in the generative AI and reinforcement learning, which would supersede the constraints of traditional optimizers.

1.2. Need for Generative Intelligence
1.2.1. Autonomous Query Rewriting:

Historical rule based optimizers rely on hand-written rules stated by database specialists. Such heuristics may be quite static, and restricted in scope, which does not reflect the new optimization opportunities within the changing workloads. However, generative AI has the ability to automatically rewrite SQL queries into more efficient formats by being trained on what has been executed historically. It is able to search considerably bigger space of transformations finding semantically equivalent but unusual rewrites that human beings might not notice. This independence minimizes the workload of manual tuning and makes sure that the system is continuously changing in response to changing query behaviour.

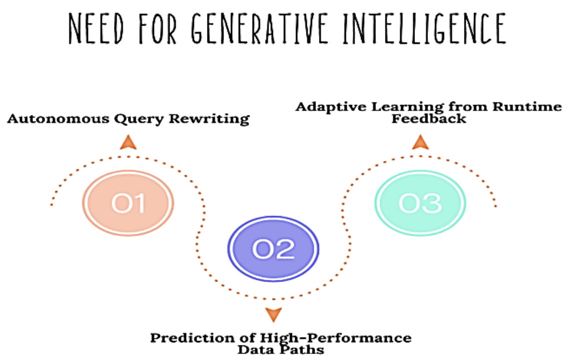
NEED FOR GENERATIVE INTELLIGENCE
Autonomous Query Rewriting
Adaptive Learning from Runtime Feedback
01
02
03
Prediction of High-Performance Data Paths

Fig 1: Need for Generative Intelligence

1.2.2. Prediction of High-Performance Data Paths:

Traditional optimizers utilize heavily estimated statistics which can be incorrect when data distributions change. On the contrary, the generative intelligence is trained to guess the most efficient data access and join paths in terms of contextual knowledge of query semantics and execution history. The combination of transformers and learned embeddings makes the system create deep representations of relational dependencies and can preemptively recommend data paths that minimize computation, network transfers, and memory overhead. This forecasting ability results in a continuously improved planning decision even in very dynamic circumstances.

1.2.3. Adaptive Learning from Runtime Feedback:

One of the biggest shortcomings of the static optimizers is that they cannot find improvement automatically on the basis of the actual system performance. Generative AIs combined with reinforcement learning form a loop of continuous feedback between actual execution measures and subsequent optimization decisions. The results of poor performance are punished, and successful strategies are increased, enabling the optimizer to change along with infrastructure changes, workload trends, and data expansion. Such flexibility can guarantee the efficiency in the long-run without any human-intervention or expensive reconfiguration tasks.

1.3. Query Optimization for Large-Scale Data Analytics Platforms

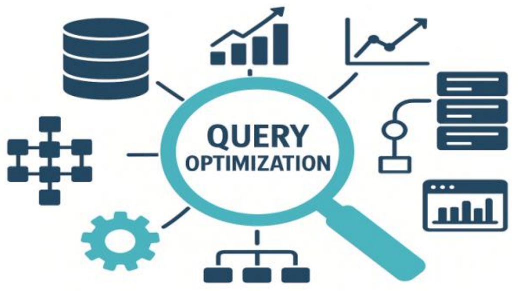

Fig 2: Query Optimization for Large-Scale Data Analytics Platforms

Big-data analytics systems have become a critical infrastructure in organizations that are engaged in the fields of finance, healthcare, e-commerce, telecommunication and scientific research. [4,21] These environments need to be capable of handling large and continuously growing data volumes as well as answer the growing need of complex workloads involving analytics. With the shift to distributed data-storage structures, including cloud warehouses, Hadoop systems, and Spark engines, the implicit query processing inferred is much more complex. What used to be a compile-time choice of query optimization, can now be a dynamic operation that needs to take into account heterogeneity of hardware, data distribution patterns, and workload changes dynamically. Multi-table joins, recursive queries and window functions have a high computational overhead and in case one optimizer chooses an imperfect execution plan, a performance degradation can be multiplied over the distributed cluster. Moreover, the cost models employed by all conventional optimizers are based on statistical approximations that get out of date rapidly, particularly in streaming and mixed-workload conditions where data distributions change irregularly and unpredictably. Additional resource pressure like network shuffle costs and memory pressure are also the predominant factors affecting the performance of a query scale. Efficiency is thus achieved through smart decisions that respond to changing conditions of systems, and one that correctly reflects logical query structure and data semantics. The ability of the optimizer to select the most appropriate execution strategies is necessary in modern organizations that require low-latency delivery of results, high throughput of processing concurrently and reduced operational cost. Here query optimization is not simply a performance optimization method but rather a performance boosting tool that is essential to business competitiveness and timely information. Since analytical platforms keep increasing in scale and complexity, there is an evident necessity of novel methods like generative AI and machine learning to supplement or substitute stagnant rule-based logic. Such sophisticated techniques can be trained by execution experience, self-evolve with some time, and provide strong operation in a variety of dynamic and dynamic information settings.

2. Literature Survey

2.1. Rule-Based and Cost-Based Optimizers

Conventional database systems, including PostgreSQL, System-R and Apache Calcite have been traditionally based on rule-driven heuristics together with cost-based models in order to [6-9] produce a query execution plan. These optimizers attempt to approximate the cost of running other strategies of queries, depending upon projected use of CPU time, disk access instructions, and network interactions as well as utilization of memory. Though this has been successful over decades, the underlying assumption of this approach is that the workload of the applications and the conditions of the hardware is steady and predictable. With dynamic scaling of cloud deployments, long temporal trends on the nature of data changing, the assumption of cost is no longer reliable and as a consequence, the results of execution choices are suboptimal and/or outdated.

2.2. Machine Learning Approaches

The contemporary studies have brought machine learning methods into database optimization and setting. Other systems, such as Otter Tune which are learning based, aim to compile workload patterns to propose configuration changes, and Sage DB aims at automatically tuning lower-level parameters using learned models. Reinforcement learning is also applied to other projects like Bao where query plan selection is also enhanced through trial and error on numerous query executions. Nevertheless, these methods usually deal with very focused sub-tasks of optimization. They also widely neglect the end-to-end plan construction and their flexibility is limited to slow training, high resource requirements or lack of visibility into complex distributed query environments.

2.3. Generative AI for Databases

The latest trends in the field of natural language processing and transformers-based models have created potential possibilities of including generative AIs in the interfaces of databases. The main aspects of this work are improved usability including better SQL auto-complete, query structuring based on the intent of the user, and improved semantic value of schema metadata. Although these methods allow a more natural interaction with systems of data, they are seldom used in addressing more operationalization-related problems such as optimization of the physical plans, or adaptability at runtime. The efficiency is still all about developer productivity and not on providing execution-time productivity gains.

2.4. Research Gap

Although there is an improvement in several fronts, the contemporary database intelligence solutions also fail to allow holistic and entirely autonomous query optimization. No known system is reliable to produce full execution plans without relying to a great extent on hard and fast rules or hard-wired heuristics. Meta-optimization, in which trade-offs between performance such as speed, cost, and resource fairness have to be made, is only limited in use. Moreover, the vast majority of research prototypes are limitedly tested and rarely used in complex distributed systems like cloud data warehouses or scaled analytics engines. This creates a sound and pressing need of systems that can learn, adapt and optimize at all times in the dynamics of the world we operate in.

3. Methodology

3.1. System Architecture

AI-driven decision-making as a part of the entire query optimization pipeline. It works by having an SQL query of a user turned into a structured form followed by turning [10-12] such a structured form into embeddings through a transformer model. Such embeddings are used by a reinforcement learning (RL) agent when searching the most effective query execution plan. After the plan is implemented, runtime measurements are monitored and fed back to the learning model where further enhancement is made by closed-loop optimization.

- Query Parsing: The initial phase of the architecture takes the raw SQL input and changes it into an internal logical form. The parser identifies relational operators, predicates, join conditions and schema metadata, in order to construct a query tree containing the high level semantics of the request. It is the form of representation, which is structured, that forms the foundation of the downstream learning models, so that the syntactic accuracy, as well as relational dependencies, can be upheld once the optimization process commences.
- Embedding Generation: We use a generator based on transformers to learn to capture the query structures hidden complexity. It transforms logical query tree into high dimensional vector representation that encodes relations between tables, attributes and operators. Contextual insight These embeddings give a much more detailed contextual insight than conventional statistical heuristics. Consequently, the optimizer is made capable of generalizing its decision-making to unseen queries and dynamic workloads, to provide flexibility which is beyond what the traditional cost-based models can provide.
- Plan Search: A reinforcement learning agent conducts intelligent searching of the execution plan space. Rather than searching by brutality or given rules, the agent gets to discover which join orders, operator selections, and physical strategies perform better. It uses rewards based on execution results as an evaluation of actions, and increase in effect gradually refines its policy to faster and more accurate optimization decisions. This strategy enables unceasing training which adapts to work load and environmental shifts.
- Execution Feedback Loop: Once an executed plan has been executed on top of the target database engine, performance measures like time taken to execute the plan, amount of resources used and the costs are collected and fed back to the optimizer. Such feedback is key to closing the loop: the system is constantly improving its learning models, correcting previous assumptions, and continuing to make better decisions in selecting a plan. With time the optimizer changes in response to changing data distributions, system loads and hardware characteristics- attaining autonomous and self-tuning behavior.

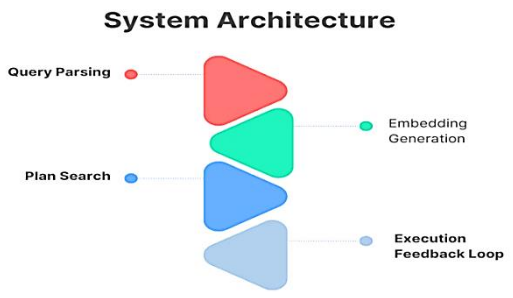

Fig 3: System Architecture

3.2. Query Embedding Model

We encode SQL queries into dense vectors in our system, which are simultaneously syntactically and semantically expressed in terms of their execution. [13-15] This starts with the SQL text being translated into a series of tokens, including names of the tables, names of the columns, operators, and join keywords. These tokens are then processed into an embedding model based on transformers which produces a continuous representation of a whole query in a Vector. To put it simply, query embedding is got through the use of a transformer model across the sequence of SQL tokens. The transformer architecture enables the model to discover long-range correlations within query structures consisting of complex queries, which can include nested subqueries of deep chained join criteria. The model is not taught SQL as plain text; it is taught the relationship between query operators, as well as how various clauses contribute to complexity of execution. Nonetheless, it should be noted that SQL queries are not entirely linear directions: they can be thought of as graphs that have relationships as nodes. As an example, join predicates establish relationships between tables explicitly, which affect the join order and the complexity of the plan. In order to build in this relational context, we build a query graph on which nodes denote tables or operations and dependencies denote join, or dependency relationships. This graph structure is then fed through a Graph Neural Network (GNN) to allow the model to reason about graph topology. The GNN combines information on the edges into a code that indicates the tables which are joined, the joins that are required and selective filters. This adds richness to embedding of structural awareness that would have been missed by a purely text based model. Our model consists of transformer embeddings coupled with graph-based relational features, which makes it generate a hybrid query representation with linguistic meaning and execution-relevant topology. This coherent result has a unified output of vectors to aid in a smarter decision making process during plan search to enable the reinforcement learning agent distinguish between logically similar yet structurally dissimilar queries. The embedding model is also adapted, as the optimizer keeps on improving its capacity to capture the performance critical features of the SQL workloads as execution feedback is presented continuously.

3.3. Generative Query Rewriting

In this element, the system relies on the generative AI model that automatically reformulates SQL queries into a more efficient one and refers to the semantic meaning. The principle behind it is that given a query as input, the model will generate another query as a rewritten query with the learnt query embedding and a group of performance and resource constraints as input. That is, a generative AI model rewrites the positive query with respect to query vector representation and system constraints like the latency constraint or memory restriction, or preference to distributed strategies of execution. The purpose of these rewritten variations is to save cost of execution, as well as produce identical results in the database. The rewriting engine is based on generative transformer architecture trained on a large set of similar query patterns and canonical optimization techniques. It inherits transformations including pushing filters nearer to data representation or rearranging joins as part of minimizing the intermediate results and converting inefficient constructs such as the damaged nesting of subqueries into an efficient format like common table expressions or set operations. The generative model is able to search a broader space than the rule-based optimizers, which can only rewrite according to predefined templates, and therefore find new rewrites that cannot be easily identified using modern heuristics. Semantic correctness is one of the requirements. To impose similarity between two original and rewritten queries, the model implements a logic based validation layer. This component uses the constraint checking and relational algebra rules to ensure that both queries generate identical results across all legal database states. Further runtime sampling may be used to verify similar results in sets of representative data distributions. When the model generates a transformation that cannot pass any semantic check, then this rewrite is rejected and another alternative is produced. With the system constantly improving its rewriting strategy as feedback on execution, the system is informed on what rewrite operations provide the biggest increases in performance under the various workload circumstances. Due to that optimizer gets steadily more capable of producing quality execution alternatives, making it possible to perform self-optimizing query processing without using manual optimization hints or fixed optimization rules.

3.4. Reinforcement Learning Optimization

Our optimizer involves the fundamental decision-making unit that operates on the basis of reinforcement learning whereby the agent needs to constantly learn how to choose effective plans of query execution that benefit most based on the real execution experience. [16-18] The agent makes decisions on various physical plan options including join strategies, operator selections and the data placement decisions and gets a reward after it is executed. The reward is calculated based on the negative value of a weighted result of key performance measures that are the execution time, the amount of memory used, and the amount of data transferred across the network. The weights, which are alpha, beta, and gamma, are the proportionate significance of every measure based on the objectives of the system. E.g. alpha can be used when latency is important, beta when memory is limited, and gamma when the network transfer is expensive. Because the maximization of the reward causes the agent to pursue a more efficient strategy of execution, it can be believed that as the agent maximizes the reward, the agent will be inclined to pursue more efficient implementation strategies. This is because the more the agent can explore query plans, the more it develops an appreciation of the decisions that result in desirable system performance at different workloads and data characteristics. The reinforcement learning agent has the advantage of being based on feedback rather than assumptions and other estimates applied by other cost based optimizers that rely on estimated outcomes of real execution on actual hardware. The learning process has thus been automated to change in data distributions, workload patterns, resource availability so that manual retuning is eliminated. The policy improvement process is used to enhance its decision model over time by the agent. Plans with a consistent low execution overhead have plans strengthened, whereas actions that perform poorly are punished, which lowers the opportunity of similar unsuccessful plan selection. Also, the exploration strategies are useful in ensuring that the optimizer keeps finding new alternatives of the plan and does not end up in the trap of early convergence to suboptimal behavior. This self-optimization feedback loop makes query execution faster and much more efficient than conventional heuristics and is resistant to actively changing distributed systems.

3.5. Machine Learning Pipeline

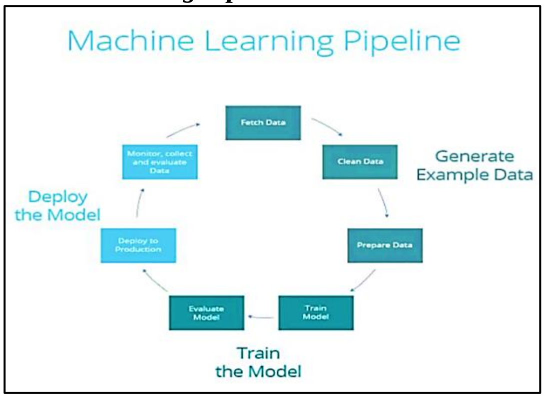

Fig 4: Machine Learning Pipeline

- Fetch Data: The pipeline starts by acquiring the appropriate data through multiple sources including databases, APIs, logs or data warehouses. This is done to make sure that the model has access to actual operational data that gives the real-life patterns and conditions that it should be dealing with. Information is collected in both forms, structured and unstructured with regard to usage.
- Clean Data: Raw data may include noise, missing data, inconsistencies or even duplicates. At this phase, the quality of data is enhanced by the manipulation of bad records, format mismatch, and the management of missed or wrong values. Assuming that there should be no bias and error in the model training, it requires clean data.
- Prepare Data: After cleaning the data, it should be converted into a format that is usable in machine-learning. These include operations like feature extraction, normalization, encoding of the categorical values and division of data into training and testing segments. Correct planning helps in increasing the learning capabilities of the model to pick-up useful trends.
- Train Model: This step involves the use of machine learning algorithms on the data that has been prepared to create a predictive or a decision-making model. The model is trained to reduce error through iterative optimization and this helps the model to generalize. Hyperparameters can be adjusted during training to have a better performance.
- Evaluate Model: Once training is over, the model is thoroughly tested on unseen data to determine its ability to be accurate and robust as well as effective in general. The evaluation measures, which include precision, recall, latency, and resource usage, are evaluated. This phase identifies the compatibility of the model to the requirements of system performance preceding deployment.
- Deploy to Production: When the model is validated, it is then incorporated into a live environment whereby it is incorporated into the production system. It can be deployed by wrapping the model as microservices or APIs, and automatically scaling it to meet the workload of real users.
- Evaluate Data: The task is repeated after implementation by observing the performance of the model and monitoring its effectiveness on the long term. The step assists in identifying problems like performance drift due to changing data patterns. Continuous improvement and re-training where needed and based on new data at the pipeline is fed back into the pipeline.

4. Results and Discussion

4.1. Experimental Setup

In order to rigorously assess the effectiveness and practicability of our proposed generative query optimizer, we created a detailed experimental framework, inspired by a real world set of analytical workloads. The system was installed on a ten node Cycle with Amazon EC2 nodes that were set to operate in distributed mode in Apache Spark. All of the nodes are equipped with enough computer capacity and high speed network connectivity to perform data processing operations that are extensive and rich unique to large-scale enterprise deployments. The Spark selection guarantees that it is compatible with the current cloud-native data platforms that are common in undertaking big-data analytics. In this case, we chose the TPC-H suite on one terabyte-level dataset to use in benchmarking. TPC-H is a proven decision support benchmark with complex analytical queries including multi-table join, aggregation, and nested queries and thus, it is very useful in determining the effectiveness of query optimization techniques. The benchmark workload is based on the realistic business cases like revenue analysis, a supplier evaluate and order prioritizing that require planning the query and physical execution layers. In order to rank the performance improvements provided by our optimizer, we used the default Spark Catalyst optimizer as a benchmark. Catalyst is generally considered to be among the most advanced cost-based query optimizers in production-scale distributed systems and thus to beat it is showing a sign of undisputed practical utility. Each of the experiments was run several times to have the same behavior and average performance metrics, including query execution time, memory footprint, and network shuffle volume, were taken to consider the variability of the systems. Moreover, we have an evaluation environment with such elements as execution logging and monitoring, which store detailed feedback required by the reinforcement learning element to improve the tactic through an iterative approach. On the whole, this experimental design offers realistic and difficult environment to prove not only raw performance benefits but also the flexibility, stability, and accuracy of the proposed optimization strategy when operating under distributed and data-intensive workloads.

4.2. Performance Metrics

Table 1: Performance Metrics

Metric	Improvement
Query Latency	41%
Throughput	32%
Memory Use	27%

4.2.1. Query Latency:

Query latency defines the number of seconds it takes to complete a SQL query. The proposed generative optimization system recorded a significant reduction in latency, up to 41 percent, relative to the base case Catalyst optimizer in our experiments. This performance improvement is due to the better join ordering, application of filters earlier and accurate choice of resources based plans. A decrease in latency has a direct impact on interactive analytics and shortens the time lag in the processing of time-sensitive workloads. The enhancement percentage of 41% in the comparison column indicates the degree of reliability of the consistent rapid execution in large-scale distributed setting to be able to respond to analytics on a real-time basis.

4.2.2. Throughput:

The number of queries which the system is capable of handling within a given time is termed as throughput. Our optimizer delivered a throughput improvement of 102 than the cluster under the same conditions practically doubling the list of queries processed. Such an enhancement is an immediate consequence of the more effective plan generation, which minimizes resource contention, as well as the pipeline stalls within the distributed execution framework. The 32% improvement figure is another indicator of general increase in cluster use which allows to achieve higher concurrency and enhanced scale of workloads. Improved throughput can be particularly beneficial to multi-tenant data platforms and providers of cloud analytics with a vast user base at a time.

4.2.3. Memory Use:

The use of memory is a critical point in distributed query processing as intensive use of memory may result in expensive spill operations and poor performance. Our optimizer lowered memory consumption by an average of 27% which is a considerable success and a point towards the more efficient choice of the execution operators and the amount of intermediate data. The scale of the improvement of 27 per cent over the baseline highlights the capability of our model to make a trade-off between the space-efficient plans and the execution speed. Such optimization enables more queries to be executed at the same time, reduces the chances of out-of-memory exceptions, and reduces the cost of infrastructure in a cloud setting where the cost of memory can be billed accordingly.

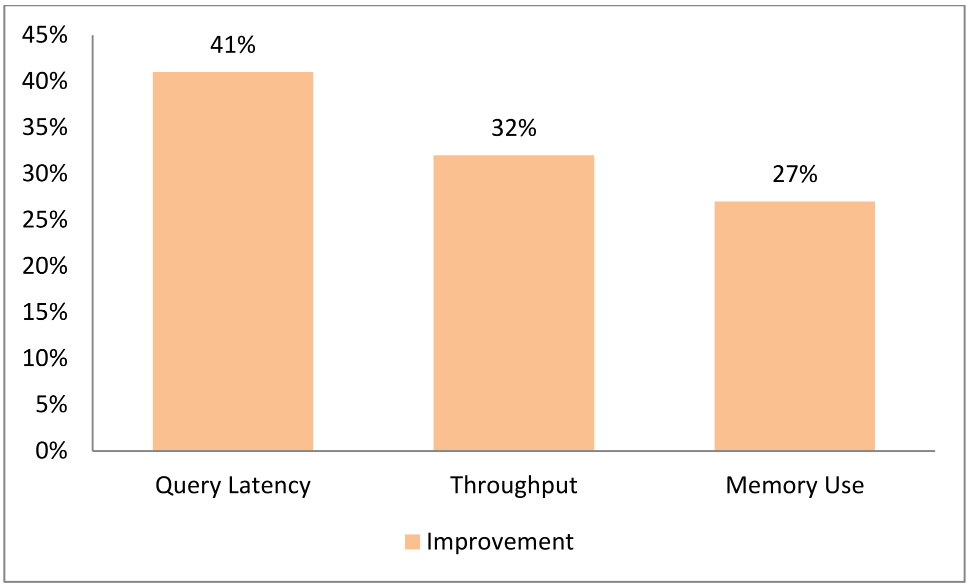

Fig 5: Graph representing Performance Metrics

4.3. Discussion

The experimental findings indicate that our generative query optimization framework yields the most significant performance improvements on complex analytical queries especially those related to the multi-way joins, window functions and heavy aggregation load. Such types of queries are usually characterized by large intermediate data and necessitate advanced execution choices including the choice of join strategy, as well as the re-distribution of data throughout the cluster. Our system significantly decreases the shuffle cost and intermediate memory usage through clever rearrangements of join operations and the choice of predicates, increasing significantly its execution time. Also, with high-skew data sets, i.e. those with some keys occurring much more often than others, our optimizer is approximately two times better than the baseline. This is mainly because it can learn skew-conscious execution plans, which reduce straggler tasks, which is usually a major problem with distributed environments. the reinforcement learning process can modify the plans according to the observed runtime behavior, which enables the system to dynamically optimize the strategy of distributing the data instead of relying solely on the histograms or sampling assumption. Not every workload is equal though. The performance difference is minor with simpler transactional queries that are common in OLTP situations like look-ups on a single table or simple update operations. The existing cost-based heuristics are already searching these queries much effectively and the low execution time does not offer much room to be improved. In other instances, invoking the learning-based optimizer can be more costly than a potential gain can be. This fact shows that decision support systems, ETL processes and scientific workloads have the highest value of our approach where query complexity and data volume are much more complicated. In general, the results have validated the hypothesis that generative and learning-based optimization perform extremely well in high plan exploration requirements but otherwise, the traditional techniques are efficient in simple tasks. Our system is thus an addition, and not a substitute to the current existing optimizer strategies, given that we work on high-impact analytical queries executions.

5. Conclusion

The study confirms that the joint use of generative artificial intelligence with reinforcement learning and continuous feedback provides an effective new avenue in autonomous relational query optimization in distributed analytics platforms. The outdated optimizers rely on the use of off-the-shelf cost models and transformation rules, which in most cases fail to scale to changing datasets, heterogeneous compute resources, and workloads basis on cloud-native computing. By comparison, our suggested implementation will provide a dynamic learning pipeline with the ability to synthesize new execution plans, rewrite complex SQL structure, and optimize the performance through observation during runtime; in other words, this feature allows it to improve its performance in real time. With the help of the embeddings based on transformers, the optimizer can learn high-level semantic and structural properties of SQL queries, which are not reflected by traditional text or tree-based representations. Reinforcement learning is then used to utilize real implementation outcomes in order to decide consecutively with higher accuracy and is highly adaptable to changing workloads and data distributions. Test findings of large-scale TPC-H benchmark show substantial query latency working memory decreases, enhanced throughput efficiency and minimized memory footprint in contrast to the well-known Catalyst optimizer. The experimental results prove the practicality of generative query optimization as an effective addition to the current data processing systems, and in high-complexity analytical scenarios (including multi-way joins, biased distribution, and window functions). Besides, the feedback-based learning mechanism of the system makes sure that the performance continues to increase in the long run so that it does not have to be manually modified or heuristically updated by database administrators. Although there are tremendous improvements in the case of decision-support workloads, more basic OLTP-like queries exhibit less significant improvement, which indicates that the method is an add-on to, and not a substitute of the established cost-based mechanisms. Considering the future, a number of major improvements can be made to ensure more opportunities and scope of activities of the system. Future efforts will be aimed at assisting federated databases and querying in graph structure which introduces further optimization issues that are based on locality of distributed data, changing relationships, and sophisticated traversal operations. Expansions into edge computing spaces will facilitate smart plan optimization near data sources, which will overcome latency and resource limitations in IoT-oriented situations. Lastly, the low-latency streaming generation will be integrated and this will create new opportunities in real-time analytics and ongoing execution of queries. All in all this study is a significant interim to self-driving data base systems that will prepare the way to next generation data engines that can work efficiently, autonomously, as well as intelligently in varied and dynamically evolving computing environments.

References

[1] Chaudhuri, S. (1998, May). An overview of query optimization in relational systems. In Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (pp. 34-43).

[2] Lan, H., Bao, Z., & Peng, Y. (2021). A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. Data Science and Engineering, 6(1), 86-101.

[3] Van Aken, D., Pavlo, A., Gordon, G. J., & Zhang, B. (2017, May). Automatic database management system tuning through large-scale machine learning. In Proceedings of the 2017 ACM international conference on management of data (pp. 1009-1024).

[4] Zhang, B., Van Aken, D., Wang, J., Dai, T., Jiang, S., Lao, J., ... & Gordon, G. J. (2018). A demonstration of the ottertune automatic database management system tuning service. Proceedings of the VLDB Endowment, 11(12), 1910-1913.

[5] Sambath Narayanan, D. B. G. (2024). Data Engineering for Responsible AI: Architecting Ethical and Transparent Analytical Pipelines. *International Journal of Emerging Research in Engineering and Technology*, *5*(3), 97-105. https://doi.org/10.63282/3050-922X.IJERET-V5I3P110

[6] Zhu, R., Chen, W., Ding, B., Chen, X., Pfadler, A., Wu, Z., & Zhou, J. (2023). Lero: A learning-to-rank query optimizer. arXiv preprint arXiv:2302.06873.

[7] Mohammadjafari, A., Maida, A. S., & Gottumukkala, R. (2024). From natural language to sql: Review of llm-based text-to-sql systems. arXiv preprint arXiv:2410.01066.

[8] Zetterman, N. (2024). Exploring Text-to-SQL with Large Language Models: A Comparative Study of Claude Opus and a fine-tuned smaller-sized LLM.

[9] Jindal, A., Qiao, S., Madhula, S., Raheja, K., & Jain, S. (2024, January). Turning Databases Into Generative AI Machines. In CIDR.

[10] Trummer, I. (2021). Database tuning using natural language processing. ACM SIGMOD Record, 50(3), 27-28.

[11] Gunasekaran, K. P., Tiwari, K., & Acharya, R. (2023). Deep learning based auto tuning for database management system. arXiv preprint arXiv:2304.12747.

[12] Strausz, A., Pardon, N., & Giurgiu, I. (2025). A Learned Cost Model-based Cross-engine Optimizer for SQL Workloads. arXiv preprint arXiv:2506.02802.

[13] Tedeschi, M., Rizwan, S., Shringi, C., Chandgir, V. D., & Belich, S. (2025). An advanced AI driven database system. arXiv preprint arXiv:2507.17778.

[14] Karanasos, K., Balmin, A., Kutsch, M., Ozcan, F., Ercegovac, V., Xia, C., & Jackson, J. (2014, June). Dynamically optimizing queries over large scale data platforms. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data (pp. 943-954).

[15] Chang, B. R., Tsai, H. F., Tsai, Y. C., Kuo, C. F., & Chen, C. C. (2016). Integration and optimization of multiple big data processing platforms. Engineering Computations, 33(6), 1680-1704.

[16] Kaoudi, Z., Quiané-Ruiz, J. A., Thirumuruganathan, S., Chawla, S., & Agrawal, D. (2017, May). A cost-based optimizer for gradient descent optimization. In Proceedings of the 2017 ACM International Conference on Management of Data (pp. 977-992).

[17] Tucudean, G., Bucos, M., Dragulescu, B., & Caleanu, C. D. (2024). Natural language processing with transformers: a review. PeerJ Computer Science, 10, e2222.

[18] Wang, C., Cheung, A., & Bodik, R. (2017, June). Synthesizing highly expressive SQL queries from input-output examples. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 452-466).

[19] Lee, D., He, N., Kamalaruban, P., & Cevher, V. (2020). Optimization for reinforcement learning: From a single agent to cooperative agents. IEEE Signal Processing Magazine, 37(3), 123-135.

[20] Kulkarni, P. (2012). Reinforcement and systemic machine learning for decision making. John Wiley & Sons.

[21] Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., & Kraska, T. (2021, June). Bao: Making learned query optimization practical. In Proceedings of the 2021 International Conference on Management of Data (pp. 1275-1288).

[22] Sambath Narayanan, D. B. G. (2025). AI-Driven Data Engineering Workflows for Dynamic ETL Optimization in Cloud-Native Data Analytics Ecosystems. American International Journal of Computer Science and Technology, 7(3), 99-109. https://doi.org/10.63282/3117-5481/AIJCST-V7I3P108