Research Challenges in Relational Database Management Systems for LLM Queries

Kerem Akillioglu
University of Waterloo
Waterloo, Canada
k2akilli@uwaterloo.ca

Anurag Chakraborty
University of Waterloo
Waterloo, Canada
a8chakra@uwaterloo.ca

Sairaj Voruganti
University of Waterloo
Waterloo, Canada
sairajv@uwaterloo.ca

M. Tamer Özsu
University of Waterloo
Waterloo, Canada
tamer.ozsu@uwaterloo.ca

ABSTRACT

Large language models (LLMs) have become essential for applications such as text summarization, sentiment analysis, and automated question-answering. Recently, LLMs have also been integrated into relational database management systems to enhance querying and support advanced data processing. Companies such as Amazon, Databricks, Google, and Snowflake offer LLM invocation directly within SQL, denoted as *LLM queries*, to boost data insights. However, open-source solutions currently have limited functionality and poor performance. In this work, we present an early exploration of two open-source systems and one enterprise platform, using five representative queries to expose functional, performance, and scalability limits in today's SQL-invoked LLM integrations. We identify three main issues: enforcing structured outputs, optimizing resource utilization, and improving query planning. We implemented initial solutions and observed improvements in accommodating LLM powered SQL queries. These early gains demonstrate that tighter integration of LLM+DBMS is the key to scalable and efficient processing of LLM queries.

VLDB Workshop Reference Format:
Kerem Akillioglu, Anurag Chakraborty, Sairaj Voruganti, and M. Tamer Özsu. Research Challenges in Relational Database Management Systems for LLM Queries. VLDB 2025 Workshop: Applied AI for Database Systems and Applications (AIDB 2025).

1 INTRODUCTION

Recent advances in artificial intelligence (AI) have enabled large language models (LLMs) to offer enhanced processing capabilities for structured (relational) data [29], as well as workloads that require access to both structured and unstructured data [30, 38, 45]. LLMs excel at row-level inference because they can reason over context and semantics instead of relying on exact string matches. This semantic capability has proven effective for some of the most important data management tasks, such as entity and schema matching [47, 51], and data cleaning [34] by overcoming the limitations of earlier approaches that completely relied on exact string matching.

The promise of row-level LLM inference has led to its integration into database management systems (DBMS). Database vendors such as Amazon [3], Databricks [2], MotherDuck [8], Google [4], and Snowflake [5] have accommodated the invocation of LLM inference within SQL queries. This integration extends relational DBMS capabilities by enabling advanced sentiment analysis and natural language processing, thus enriching query results with nuanced insights. We refer to such SQL queries that invoke LLMs as *relational LLM queries*, or simply *LLM queries*.

Despite their growing adoption by major enterprises, the integration of LLMs with DBMSs remains underexplored, and the incorporation of LLMs introduces additional challenges. For reference, enterprise LLM inference solutions expose data to third-party providers, and it incurs significant privacy risks for sensitive information. Organizations seeking to leverage LLM capabilities without compromising data confidentiality must serve models locally. This paper demonstrates the functional challenges of current LLM+DBMS integrations and shows that, even when they function, the eminent systems are slow because they lack effective optimizations and they scale poorly. As an example, running the simple LLM query in Figure 1 on only 17,000 rows takes 5 hours to complete using local inference on a single A100 GPU combined with an open source DBMS; and it would take around 12 days to process one million rows in our single GPU setup. Hence, closing the functionality and efficiency gaps is critical for DBMSs to scale AI-enhanced workloads.

In this work, we provide an early exploration of processing LLM queries on existing systems through analyzing two open-source systems and one enterprise system from functionality and performance aspects. We use a fixed set of LLM queries in our analysis (see Section 2). We demonstrate that existing systems have difficulty executing many of the queries in this set, and we analyze the underlying challenges and potential solutions. Our contributions are the following:

(1) We present an evaluation of emerging systems that integrate LLM capabilities into SQL querying. Our evaluation uses open source solutions PostgreSQL with pgAI [39] and DuckDB with FlockMTL [17], and an enterprise solution in MotherDuck [8]. For each case, we examine functionality

Proceedings of the VLDB Endowment. ISSN 2150-8097.

and overall performance, and analyze why queries are not able to run or have poor performance.

(2) Our study provides practical insights and recommendations on what opportunities exist for query optimization for queries that involve LLM invocations, and what inference-based optimizations can be implemented to meet the specific requirements of analytical queries.

2 RELATIONAL LLM QUERIES

In our study, we focus on the queries outlined by Liu et al. [29] since they are a good representation for embedding LLM invocations at various points within SQL queries. Each query is identified by a unique name and characterized by its distinct LLM invocation strategy. Below, we summarize the five representative queries we use:

Q1: LLM Projection – This query invokes the LLM in the SELECT clause to project a transformed output from input text fields. The purpose is to derive insights directly from the provided textual data.

```
SELECT LLM("Recommend movies for the user based on
     {movie information} and {user review}",
     m.movie_info, r.review_content)
FROM reviews r
JOIN movies m ON r.rotten_tomatoes_link ==
m.rotten_tomatoes_link
```

Figure 1: LLM Projection Query.

Q2: LLM Filter – Here, the LLM is invoked in the WHERE clause to evaluate and filter rows based on semantic criteria. The LLM function determines if the input text meets a specified condition, thereby controlling which records are included in the result set.

```
SELECT m.movie_title
FROM Movies m
JOIN Reviews r ON r.rotten_tomatoes_link =
m.rotten_tomatoes_link
WHERE LLM("Analyze whether this movie would be
    suitable for kids based on {movie information}
     and {user review}", m.movie_info, r.
    review_content) == "Yes"
AND r.review_type == "Fresh"
```

Figure 2: LLM Filter Query.

Q3: Multi-LLM Invocation – This query combines two LLM invocations: one in SELECT clause to generate primary outputs and the other in the WHERE clause to filter the results based on additional content suitability criteria. The purpose is to refine the final output by sequentially applying multiple LLM functions.

```
SELECT LLM("Recommend movies for the user based on
     {movie information} and {user review}", m.
    movie_info, r.review_content) AS
    recommendations
FROM Movies m
JOIN Reviews r ON r.rotten_tomatoes_link =
m.rotten_tomatoes_link
WHERE LLM("Analyze whether this movie would be
    suitable for kids based on {movie information}
     and {user review}", m.movie_info, r.
    review_content) == "Yes"
AND r.review_type == "Fresh"
```

Figure 3: Multi-LLM Query.

Q4: LLM Aggregation – The LLM is called to assign satisfaction ratings for reviews for each movie title to qualitatively measure average sentiment for overall customer feedback. Then these numerical scores are aggregated using an average function. This approach synthesizes qualitative data into a quantitative summary metric.

```
SELECT AVG(LLM("Rate a satisfaction score between
    0 (bad) and 5 (good) based on {review} and {
    info}: ",r.review_content, m.movie_info)) as
    AverageScore
FROM reviews r
JOIN movies m ON r.rotten_tomatoes_link =
m.rotten_tomatoes_link
GROUP BY m.movie_title
```

Figure 4: LLM Aggregation Query.

Q5: RAG – This query implements a Retrieval-Augmented Generation (RAG) approach by first retrieving relevant context via a similarity search and then using the LLM to generate an answer. The LLM invocation here enhances response generation by leveraging externally retrieved contextual data.

```
SELECT LLM("Given the following {context}, answer
    this question",
     VectorDB.similarity_search(s.question),
     s.question)
FROM squad s
WHERE s.is_impossible == False;
```

Figure 5: RAG Query.

3 SYSTEMS TESTING

Our objectives are two-fold: (1) we want to understand what is required to be able to execute the queries discussed in the previous section, and (2) we want to test the performance of executing them and understand the factors that affect query performance. As we discuss below, most of these queries cannot be executed using these open-source systems. Some of the issues can be addressed by careful engineering designs, but others require new approaches that require research.

3.1 Testing Setup

Our setup uses a local inference model: Meta's LLaMA 3.1 with 8B parameters [19]. This model is served with both Ollama and vLLM to understand the role of the model serving engine's impact on functionality and performance of LLM+DBMS integrations. As noted above, the open source systems we use are PostgreSQL 16 with pgAI 0.8.0 and DuckDB v1.1.4 with FlockMTL v0.2.0 [22]. We run our experiments on a Linux server equipped with a single NVIDIA A100 80GB PCIe GPU, an Intel Xeon® Platinum 8380 CPU with 160 cores. We also test an enterprise system, MotherDuck, through using its API and MotherDuck Prompt() [1] functionality which uses OpenAI GPT4o-mini [6] as the LLM endpoint, and we use their built-in embedding [32] function.

Our queries (Section 2) and the datasets are the same as those in Liu et al. [29]: Rotten Tomatoes [37] and Stanford Question Answering Dataset (SQuAD) [40]. We had to limit the size of our datasets as explained in Section 1, because applying LLMs to large datasets incurs significant computational costs. We used the size of the *movies* table of the Rotten Tomatoes dataset as our limit row number for the *reviews* and SQuAD data, which is 17,712 rows, and we randomly sample from these datasets. Rotten Tomatoes is used for queries 1-4 as it has a relational structure, and SQuAD is used for Q5 since question & answering datasets are natively compatible with RAG because they are context-rich and semantically searchable. For RAG queries on open source systems, we utilize the Stella embedding model [50].

3.2 Functional Testing

Table 1 shows which systems are capable of running which queries, and we discuss the related research challenges in more detail in Section 4. Query 1 runs successfully on all tested systems since it simply summarizes previously retrieved rows and does not incur additional challenges. In contrast, queries 2, 3, and 4 fail in both FlockMTL[1] and pgAI due to structured output problems that are discussed further in Section 4.1. Specifically, queries 2 and 3 require the LLM to strictly output "Yes", and any deviation from this format prevents proper filtering. Similarly, query 4 demands the LLM to provide only a numeric output between 1 and 4 to compute an average. However, the LLM consistently adds extra text, which causes execution failures. MotherDuck, utilizes GPT-4o-mini for inference and successfully executes queries 1–4 due to OpenAI's structured, constrained decoding capabilities [36]. To leverage vLLM's structured decoding capabilities through XGrammar [16], we modified FlockMTL to work with vLLM [9], and ran queries on FlockMTL. This modification enabled the successful execution of queries 2 and 3, but not query 4 (LLM aggregation). The problem with query 4 stems from a structured output and query planner mismatch. Even with constrained decoding, FlockMTL's `llm_reduce`, FlockMTL's LLM-aggregation function, returns an untyped text blob rather than the structured numeric type the planner needs for `AVG()`, so the operator and optimizer cannot align. Query 5 also suffers from query planning issues and fails to run both on MotherDuck as well as FlockMTL. We traced this issue in DuckDB's query planner where the HNSW index lookup fails to trigger and instead performs a

Systems	Q1	Q2	Q3	Q4	Q5
pgai-ollama	✓	✗	✗	✗	✓
flockmtl-ollama	✓	✗	✗	✗	✗
flockmtl-vllm	✓	✓	✓	✗	✗
motherduck-gpt	✓	✓	✓	✓	✗

Table 1: Query success/failure for each system running Queries 1-5 from Section 2. ✓ denotes success, ✗ denotes failure.

cross join between the two vector embedding tables being referenced. This leads to the query running out of memory due to the large intermediate table being materialized. pgAI is the only system that could execute query 5 because we manually enforced the optimal query plan (see Figure 10) to make it run, and the pgvector extension is tightly integrated into the pgAI environment. First, we applied the filter `s.is_impossible == False` to narrow down the data, and then performed the similarity search (filter pushdown & top-k pushdown). This returns only the top 3 relevant contexts for each question before making the LLM call. This fix is specific to this query because we had to alter its plan to make it run. Broader planning issues and requirements are discussed in Section 4.3.

3.3 Performance Testing

In this section, we report query latencies and resource utilization for the studied workload. Table 2 lists the latencies for each query, and Figures 6-9 show GPU utilization graphs for a selected set of queries. As FlockMTL and pgAI follow different execution strategies, we first analyze how these strategies influence overall performance. We then present MotherDuck's results separately since it is a closed enterprise service running on proprietary hardware, which is not directly comparable to the open-source systems.

Systems	Q1	Q2	Q3	Q4	Q5
pgai-ollama	719.5	-	-	-	204.9
flockmtl-ollama	370.0	-	-	-	-
flockmtl-vllm	342.4	447.6	617.2	-	-

Table 2: Query Latency (minutes) for each system. pgai-ollama denotes the relational extension (pgai) and inference engine (ollama) being used. - denotes the query was not able to run.

The general execution strategy of FlockMTL for all its LLM functions is to batch multiple rows in a single prompt request by taking the total context window of the model (e.g., 128,000 tokens for Llama models), subtracting the tokens needed for the system prompt, and then sending enough rows until the window limit is reached. It then instructs the model to output exactly 1 row for each input row in JSON format. In practice, this approach does not work well since the LLM is not robust enough to follow these instructions for hundreds of rows. Another issue is that the LLM may output multiple lines of rows, and sometimes not even a single row for an

[1]After sharing results with the FlockMTL team, they have enabled structured output when using OpenAI-compatible and Ollama providers.

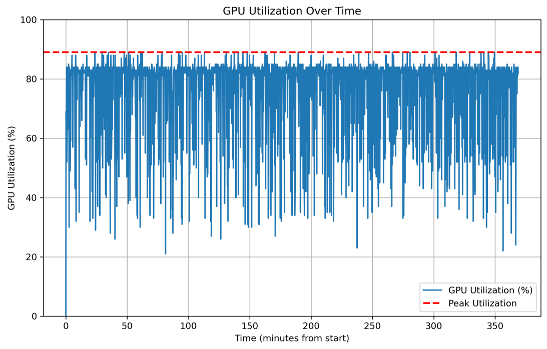

Figure 6: FlockMTL (Ollama) GPU Utilization for Q1

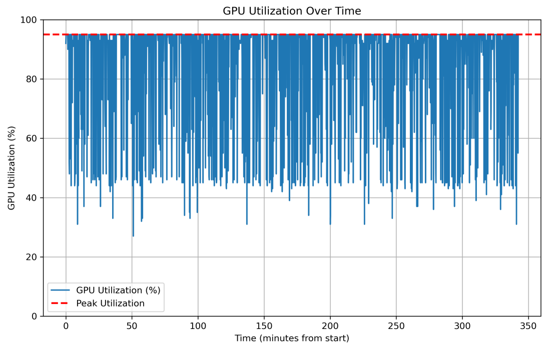

Figure 7: FlockMTL (vLLM) GPU Utilization for Q1

input row. These issues make it infeasible to track which output row from the LLM output maps to an input row. To fix these issues we modified FlockMTL's code to keep the batch size for each prompt to exactly one row, which ensured the queries run to completion. However, in terms of GPU utilization, this approach is inefficient since it only sends 1 input row as a prompt request at-a-time which is why FlockMTL's runtime numbers are an order of magnitude slower (with both Ollama and vLLM) compared to the enterprise system MotherDuck. From Figure 6, we observe that the peak GPU utilization across the query runtime for Q1 with Ollama (85%) is less than the peak GPU utilization with vLLM (Figure 7, 95%), which reflects in the runtime numbers (Table 2).

On the other hand, in PostgreSQL, parallel query execution enhances performance by distributing the work among multiple worker processes. As the data volume increases, PostgreSQL automatically assigns more workers to process the data concurrently, and pgAI is designed similarly. However, when we attempted to run Q1 as a single block, we observed that pgAI's implementation and its corresponding UDF (User-Defined Function) try to initiate subtransactions within these parallel workers. Since PostgreSQL

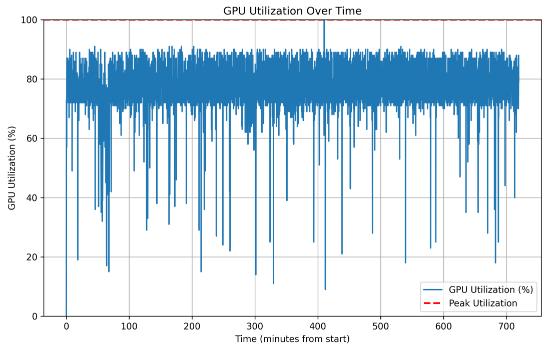

Figure 8: pgAI (Ollama) GPU Utilization for Q1

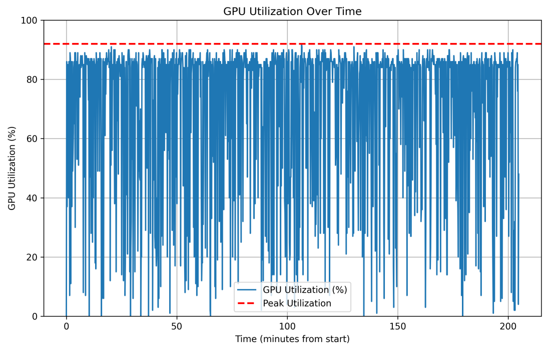

Figure 9: pgAI (Ollama) GPU Utilization for Q5

does not allow subtransactions during parallel operations, an error[2] gets triggered, and the query does not run.

To resolve this issue, we forced the query to run in a single-process context where subtransactions are permitted. While this fix resolves the error, processing one row at-a-time results in poor performance and resource utilization. To improve efficiency, we implemented an optimization to batch requests in pgAI and observed significant performance gains. Without this optimization, processing a single row for query 1 took around 4 seconds on average; and after applying it, per row processing time dropped to 2.5 seconds. Our implementation also showed its effect on GPU utilization. When there's no batch optimization of requests, GPU utilization fluctuates dramatically, as can be seen in Figure 9. With optimization, we recorded a stable and higher GPU utilization graph compared to processing other queries, which shows an average utilization of 76% and a median of 75%.

The enterprise solution (motherduck-gpt) performs best across all queries, as shown in Table 3. The reason is largely because it leverages the OpenAI API for inference rather than relying on local inference. This approach takes advantage of the state-of-the-art performance and efficiency of enterprise systems and hardware

[2]We opened an issue on the pgAI GitHub repository; however, as of this writing, it has remained unanswered for more than a week.

as opposed to having to optimize local inference on a single GPU. In addition, MotherDuck further improves speed by concurrently sending up to 256 requests to the model provider [31]. By parallelizing these requests, the system significantly reduces overall latency and accelerates processing, leading to consistently lower query response times.

System	Q1	Q2	Q3	Q4	Q5
motherduck-gpt	16.2	4.3	13.4	8.3	–

Table 3: Query latency (minutes) for the proprietary MotherDuck platform.

4 RESEARCH CHALLENGES

Existing systems are still in the development stage, and many research challenges remain to be addressed. These challenges span multiple aspects, including structured output handling, resource utilization, and query planning.

4.1 Structured Outputs

Due to their non-deterministic nature, LLMs can generate syntactically inconsistent outputs, and for LLM queries, this can lead to the generation of schema-breaking text that disrupts downstream processing. For example, in an "LLM-Aggregation" query, the model must return a single number between 1 and 5; if LLM instead outputs anything else, the `AVG` SQL function cannot parse the value, and the query fails to execute. For any system that supports relational LLM queries, structured outputs are not a technical enhancement, it's a fundamental requirement to run queries. To address this issue, there are three possible solutions, each with its own trade-offs: fine-tuning the model, prompt engineering and constrained decoding.

Fine-tuning is the process of supplementing a pretrained model with domain-specific training on small datasets to perform better for particular tasks. Once tuned, the fine-tuned model captures nuanced domain semantics with minimal runtime overhead. It has also been effective on tabular data for applications like data synthesis and privacy protection [44, 46]. The trade-offs are as follows: collecting schema-accurate examples can be costly, parameter-efficient tuning consumes significant time and compute resources [15], in the case of a schema change, another tuning pass is required, and closed-source models cannot be tuned at all.

Prompt engineering seeks optimal task performance by tailoring prompts to a specific model–dataset pair [49]. It can be complemented with system prompts that add global instructions [33]. This technique adds virtually no runtime cost and is model-agnostic because behavior changes are just text edits. Its success, however, is probabilistic and context-specific. Despite having semantically equivalent prompts for the same question, performance can vary significantly [12]. Also, each task, schema, or domain requires its own tailored prompt, whose effectiveness can decline in long or multi-turn contexts [25, 28].

Constrained decoding is the technique to modify LLM's token-generation process so that each subsequent token is restricted to choices that preserve the required output structure. Doing so, the generated text can be ensured to adhere to constraints like high-level templates through using regular expressions [10], context-free grammars [16, 48], or a combination of both [23]. It has proven to be effective in text-to-SQL translation tasks [42], and has been used in practical applications like vLLM [16, 18]. Advantages of using structured decoding removes the need for fine-tuning, any additional post-processing, ad-hoc parsing, retrying and prompting on top of LLMs [11]. Although effective in ensuring correctness, it increases the computational overhead and incurs challenges with batch and parallel processing as discussed in Section 4.2.

For our work, our initial solution for this problem was to use prompt engineering. Once the dataset grew, our optimized prompts yielded inconsistent outputs that prevented efficient parsing. We ruled out fine-tuning due to its high cost of curating large volumes of schema-perfect examples and spending hours of GPU compute. Instead, we used constrained decoding: giving the model an explicit grammar that systematically forces every answer to match the table schema and requires no retraining.

4.2 Resource Utilization

During inference, batching strategies play a key role in resource utilization as well as query latency. Frequent data transfers between the CPU and GPU reduce overall GPU utilization, and the absence of an effective batching strategy largely explains the spikes observed in Figures 6, 7, and 9. Our workload requires a combination of GPU-CPU operations, and the inference engine uses the GPU to process the input prompt (prefill) and generate output tokens (decode). A good scheduling design for relational LLM queries should aim for maximum GPU utilization, and involve sending multiple concurrent asynchronous requests to the inference backend instead of blocking API calls. Asynchronous parallel requests to the inference engine ensure that the GPU bound output token generation operation overlaps with the CPU bound operations. This allows the GPU to remain active by generating tokens for a concurrent request while waiting for output validation of a prior request.

Also, when batching requests, the inference engine should separate batches that require constrained generation from those that do not. Due to structured decoding, the logit masks for each decoding step must be generated and validated on the CPU to constrain the output. This results in each decoded token being passed to the CPU one at-a-time. When output token validation occurs on the CPU, the GPU remains idle in the blocking API request case. Therefore, mixing constrained and non-constrained requests in the same batch leads to performance degradation as the overhead of constrained requests slows down the non-constrained ones [7].

On top of batching requests, row level batching is a potential strategy as FlockMTL's design suggests. However, it comes with practical challenges, as mentioned in Section 3.3. When using this approach, the number of input tokens and the output size must be carefully calculated to fit within the model's context window. Ideally, the best performance would come from batching multiple rows into a single request while ensuring each row's output is handled separately and running multiple requests asynchronously. This would depend on new methods being developed beyond constrained decoding that can send multiple rows of input to an LLM, and return separate, per-row output back to the database.

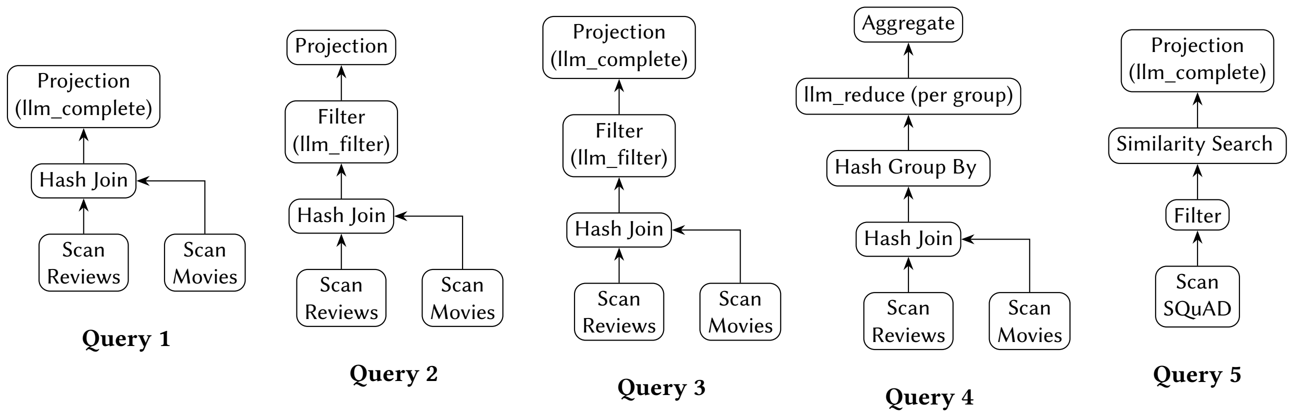

Figure 10: Query plans for relational-LLM queries

4.3 Query Plans

There are two main strategies in integrating LLM functions in the query planning stage. Databricks [2], Amazon Redshift [3], and pgAI use UDFs to call LLMs, while FlockMTL uses custom C++ functions that make LLM calls.

Having UDFs allows LLM queries to run directly through SQL statements without the need to integrate external tools into existing workflows. However, these UDFs are treated as "blackboxes" by the query planner and no optimization rules are applied when making the LLM inference call. Figure 10 shows the query plans for the relational LLM queries, and they are similar across all the systems we tested. While certain optimizations such as filter pushdowns are observed for queries 2 and 4 (Figure 10), optimizations within the LLM function call such as reordering of the column attributes being referenced based on cardinality estimates are not considered.

Current relational DBMS optimizers leave a lot of performance on the table by ignoring the cost of LLM invocations. The planner can recover that performance by factoring in the LLM prompts embedded in UDFs and other custom functions. To minimize latency and cost, the optimizer should consider CPU-GPU resource allocation and execution ordering, batching, KV-cache reuse, and prompt overlap as first-class objectives. Guided by these insights, such as prefix-cache sharing across low cardinality columns [29], the planner can reorder input rows so tuples with similar attribute values appear together. Thus, prompt prefixes can be shared, overlapping text can be batched, attention states can be reused, and pre-fill overhead can be reduced. Incorporating the true cost of LLM calls can close a gap in current relational optimizers and unlock substantial untapped performance.

5 RELATED WORK

As LLMs are increasingly being applied in different areas, a growing research community is addressing challenges to apply LLMs to structured tabular data tasks [21]. This community tackles problems such as generating new table columns or features [35], data cleaning [34], table understanding through representation learning [14], and semantically enriching table content [20]. Even though this community's work is highly relevant, we are not concerned with such tasks, instead, we aim to explore how LLM-enhanced workloads can be accommodated within DBMSs as we conduct an early exploration for the open-source systems that enable the execution of SQL queries that invoke LLMs.

To efficiently process AI-powered analytical queries, LLMs are used for unstructured (text-centric) [13, 27, 43] and structured (relational) [29] workloads, both separately and in combination [30, 38, 45]. Researchers introduce optimizations like prompt prefix-sharing and row-reordering [29], designing declarative querying primitives [38, 45] and physical operators [41] to process data efficiently with LLMs. What we do in our work complements these systems to efficiently support their workloads as it is orthogonal to these solutions, and can be adopted with such frameworks.

RAG enhances retrieval by fetching only the most relevant data from large datasets, keeping LLM prompts within length limits while grounding the model's output in essential information [24, 26]. We are not concerned with using LLMs for RAG operations. Instead, we aim to optimize the DBMS+LLM interaction by enabling LLM-powered SQL queries to be performed efficiently.

6 CONCLUSION

Running LLM queries is attractive because they extend traditional SQL and can greatly enrich analytics in relational DBMSs. Yet executing these queries efficiently and practically is difficult, especially when the model must run locally to avoid exposing sensitive data and maintain privacy. Our early exploration reveals key functionality and performance challenges, as well as the trade-offs of executing LLM queries inside relational DBMSs.

Key challenges are enforcing structured outputs, optimizing batching, and improving query planning. We highlight trade-offs among structured output techniques, show how insufficient batching decreases the GPU utilization, and demonstrate that planner-execution mismatches can harm functionality and performance. We also highlight the need for incorporating LLM costs into query planning to maximize resource utilization. Our initial solutions demonstrate measurable improvements in LLM+DBMS integration. We believe that LLM queries might benefit from adopting different approaches, such as approximate query processing techniques, and may require rethinking relational DBMS query execution strategies to maximize GPU and model efficiency.

REFERENCES

[1] [n.d.]. "https://motherduck.com/docs/sql-reference/motherduck-sql-reference/ai-functions/prompt"

[2] 2025. AI Functions on Databricks. https://docs.databricks.com/aws/en/large-language-models/ai-functions. Databricks Documentation. Last updated Feb 10, 2025.

[3] 2025. Large Language Models for Sentiment Analysis with Amazon Redshift ML Preview. https://aws.amazon.com/blogs/big-data/large-language-models-for-sentiment-analysis-with-amazon-redshift-ml-preview/. AWS Big Data Blog.

[4] 2025. LLM with Vertex AI: Only Using SQL Queries in BigQuery. https://cloud.google.com/blog/products/ai-machine-learning/llm-with-vertex-ai-only-using-sql-queries-in-bigquery. Google Cloud Blog.

[5] 2025. Snowflake Cortex: LLM Functions. https://docs.snowflake.com/en/user-guide/snowflake-cortex/llm-functions. Snowflake Documentation.

[6] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[7] Anyscale. 2025. Constrained generation with JSON mode. https://docs.anyscale.com/llms/serving/guides/json_mode/. Accessed: 17 March 2025.

[8] RJ Atwal, Peter Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, Jacob Lacouture, Yves Le Maout, Boaz Leskes, Yao Liu, Alex Monahan, Dan Perkins, Tino Tereshko, Jordan Tigani, Nick Ursa, Stephanie Wang, and Yannick Welsch. 2024. MotherDuck: DuckDB in the Cloud and in the Client. In *Proceedings of the 14th Conference on Innovative Data Systems Research (CIDR 2024)*. Chaminade, CA, USA.

[9] BentoML and Red Hat. 2025. *Structured Decoding in vLLM: A Gentle Introduction.* https://www.bentoml.com/blog/structured-decoding-in-vllm-a-gentle-introduction Accessed: 15 March 2025.

[10] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1946–1969.

[11] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation. arXiv:2403.06988 [cs.LG] https://arxiv.org/abs/2403.06988

[12] Bowen Cao, Deng Cai, Zhisong Zhang, Yuexian Zou, and Wai Lam. 2024. On the Worst Prompt Performance of Large Language Models. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 69022–69042. https://proceedings.neurips.cc/paper_files/paper/2024/file/7fa5a377b7ffabcce43cd00231bb3f9c-Paper-Conference.pdf

[13] Hanjun Dai, Bethany Yixin Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya Mangpo Phothilimthana, Charles Sutton, and Dale Schuurmans. 2024. UQE: A Query Engine for Unstructured Databases. In *Advances in Neural Information Processing Systems (NeurIPS 2024)*.

[14] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: table understanding through representation learning. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 307–319. https://doi.org/10.14778/3430915.3430921

[15] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLORA: efficient finetuning of quantized LLMs. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 441, 28 pages.

[16] Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2024. XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models. arXiv:2411.15100 [cs.CL] https://arxiv.org/abs/2411.15100

[17] Anas Dorbani, Sunny Yasser, Jimmy Lin, and Amine Mhedhbi. 2025. Beyond Quacking: Deep Integration of Language Models and RAG into DuckDB. arXiv:2504.01157 [cs.DB] https://arxiv.org/abs/2504.01157

[18] dottxt ai. 2025. Outlines: Structured Text Generation. https://github.com/dottxt-ai/outlines. Accessed: 15 March 2025.

[19] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, and et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[20] Yael Einy, Tova Milo, and Slava Novgorodov. 2024. Cost-Effective LLM Utilization for Machine Learning Tasks over Tabular Data. In *Proceedings of the Conference on Governance, Understanding and Integration of Data for Effective and Responsible AI* (Santiago, AA, Chile) *(GUIDE-AI '24)*. Association for Computing Machinery, New York, NY, USA, 45–49. https://doi.org/10.1145/3665601.3669848

[21] Xi Fang, Weijie Xu, Fiona Anting Tan, Jiani Zhang, Ziqing Hu, Yanjun (Jane) Qi, Scott Nickleach, Diego Socolinsky, "SHS" Srinivasan Sengamedu, and Christos Faloutsos. 2024. Large language models (LLMs) on tabular data: Prediction, generation, and understanding — a survey. *Transactions on Machine Learning Research* (2024). https://www.amazon.science/publications/large-language-models-llms-on-tabular-data-prediction-generation-and-understanding-a-survey

[22] FlockMTL. 2024. FlockMTL. http://github.com/dsg-polymtl/flockmtl/releases.

[23] guidance ai. 2025. Guidance: A Guidance Language for Controlling Large Language Models. https://github.com/guidance-ai/guidance. Accessed: 5 June 2025.

[24] Xingyu Ji, Aditya Parameswaran, and Madelon Hulsebos. [n.d.]. TARGET: Benchmarking Table Retrieval for Generative Tasks. In *NeurIPS 2024 Third Table Representation Learning Workshop*.

[25] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15339–15353. https://doi.org/10.18653/v1/2024.acl-long.818

[26] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 793, 16 pages.

[27] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. 2024. A Declarative System for Optimizing AI Workloads. arXiv:2405.14696 [cs.CL]

[28] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. https://doi.org/10.1162/tacl_a_00638

[29] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. 2024. Optimizing llm queries in relational workloads. *arXiv preprint arXiv:2403.05821* (2024).

[30] Shicheng Liu, Jialiang Xu, Wesley Tjangnaka, Sina J Semnani, Chen Jie Yu, and Monica S Lam. 2023. SUQL: Conversational Search over Structured and Unstructured Data with Large Language Models. *arXiv preprint arXiv:2311.09818* (2023).

[31] MotherDuck. [n.d.]. "https://motherduck.com/blog/sql-llm-prompt-function-gpt-models/"

[32] MotherDuck. 2025. Introducing the embedding() function: Semantic search made easy with SQL. https://motherduck.com/blog/sql-embeddings-for-semantic-meaning-in-text-and-rag/.

[33] Norman Mu, Jonathan Lu, Michael Lavery, and David Wagner. 2025. A Closer Look at System Prompt Robustness. *arXiv preprint arXiv:2502.12197* (2025).

[34] Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mohamed Eltabakh, Mourad Ouzzani, and Nan Tang. 2024. RetClean: Retrieval-Based Data Cleaning Using LLMs and Data Lakes. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4421–4424. https://doi.org/10.14778/3685800.3685890

[35] Jaehyun Nam, Kyuyoung Kim, Seunghyuk Oh, Jihoon Tack, Jaehyung Kim, and Jinwoo Shin. 2024. Optimized Feature Generation for Tabular Data via LLMs with Decision Tree Reasoning. arXiv:2406.08527 [cs.LG] https://arxiv.org/abs/2406.08527

[36] OpenAI. 2023. *Introducing Structured Outputs in the API.* https://openai.com/index/introducing-structured-outputs-in-the-api/ Accessed: 15 March 2025.

[37] Bo Pang and Lillian Lee. 2005. Seeing Stars: Exploiting Class Relationships for Sentiment Categorization with Respect to Rating Scales. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, Kevin Knight, Hwee Tou Ng, and Kemal Oflazer (Eds.). Association for Computational Linguistics, Ann Arbor, Michigan, 115–124. https://doi.org/10.3115/1219840.1219855

[38] Liana Patel, Siddharth Jha, Carlos Guestrin, and Matei Zaharia. 2024. LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data. arXiv:2407.11418 [cs.DB]

[39] pgAI. 2024. pgAI. https://github.com/timescale/pgai.

[40] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *arXiv preprint arXiv:1806.03822* (2018).

[41] Dario Satriani, Enzo Veltri, Donatello Santoro, Sara Rosato, Simone Varriale, and Paolo Papotti. 2025. Logical and Physical Optimizations for SQL Query Execution over Large Language Models.

[42] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 9895–9901. https://doi.org/10.18653/v1/2021.emnlp-main.779

[43] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. arXiv:2410.12189 [cs.DB] https://arxiv.org/abs/2410.12189

[44] Mohammed Alkhudhayri Catherine Cao Samuel Guo Nicholas Roberts Frederic Sala Sonia Cromp, Satya Sai Srinath Namburi GNVV. 2025. Tabby: Tabular Data Synthesis with Language Models. *arXiv preprint arXiv:2405.01147* (2025). https://arxiv.org/abs/2405.01147

[45] Matthias Urban and Carsten Binnig. 2024. ELEET: Efficient Learned Query Execution over Text and Tables. *Proc. VLDB Endow.* 17, 13 (Sept. 2024), 4867–4880. https://doi.org/10.14778/3704965.3704989

[46] Yuxin Wang, Duanyu Feng, Yongfu Dai, Zhengyu Chen, Jimin Huang, Sophia Ananiadou, Qianqian Xie, and Hao Wang. 2024. HARMONIC: Harnessing LLMs for Tabular Data Synthesis and Privacy Protection. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 100196–100212. https://proceedings.neurips.cc/paper_files/paper/2024/file/b5aebe9a48398525a9da27a1df827d60-Paper-Datasets_and_Benchmarks_Track.pdf

[47] Yu Wang, Luyao Zhou, Yuan Wang, Zhenwan Peng, and Surya Prakash. 2024. Leveraging Pretrained Language Models for Enhanced Entity Matching: A Comprehensive Study of Fine-Tuning and Prompt Learning Paradigms. *Int. J. Intell. Syst.* 2024 (Jan. 2024), 14. https://doi.org/10.1155/2024/1941221

[48] Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702* (2023).

[49] Qinyuan Ye, Mohamed Ahmed, Reid Pryzant, and Fereshte Khani. 2024. Prompt Engineering a Prompt Engineer. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 355–385. https://doi.org/10.18653/v1/2024.findings-acl.21

[50] Dun Zhang, Jiacheng Li, Ziyang Zeng, and Fulong Wang. 2025. Jasper and Stella: distillation of SOTA embedding models. arXiv:2412.19048 [cs.IR] https://arxiv.org/abs/2412.19048

[51] Yunjia Zhang, Avrilia Floratou, Joyce Cahoon, Subru Krishnan, Andreas C. Müller, Dalitso Banda, Fotis Psallidas, and Jignesh M. Patel. 2023. Schema Matching using Pre-Trained Language Models. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 1558–1571. https://doi.org/10.1109/ICDE55515.2023.00123