Beyond Quacking: Deep Integration of Language Models and RAG into DuckDB

Anas Dorbani[1] Sunny Yasser[1] Jimmy Lin[2] Amine Mhedhbi[1]
[1]Polytechnique Montréal, [2]University of Waterloo
anas.dorbani@polymtl.ca,sunny.yasser@polymtl.ca,jimmylin@uwaterloo.ca,amine.mhedhbi@polymtl.ca

ABSTRACT

Knowledge-intensive analytical applications retrieve context from both structured tabular data and unstructured free text documents for effective decision-making. Large language models (LLMs) have significantly simplified the prototyping of such retrieval and reasoning data pipelines. However, implementing them efficiently remains challenging and demands significant effort. Developers must often orchestrate heterogeneous systems, manage data movement, and handle low-level concerns such as LLM context management.

To address these challenges, we introduce FlockMTL: an extension for DBMSs that integrates LLM capabilities and enables retrieval-augmented generation (RAG) within SQL. FlockMTL provides LLM-powered scalar and aggregate functions, enabling chained predictions over tuples. It further provides data fusion functions to support hybrid search. Drawing inspiration from the relational model, FlockMTL incorporates: (i) seamless optimizations such as batching and meta-prompting; and (ii) resource independence through novel SQL DDL abstractions: PROMPT and MODEL, introduced as first-class schema objects alongside TABLE.

PVLDB Reference Format:
Anas Dorbani, Sunny Yasser, Jimmy Lin, and Amine Mhedhbi. Beyond Quacking: Deep Integration of LMs and RAG into DuckDB. PVLDB, 18(12): 5415 - 5418, 2025.
doi:10.14778/3750601.3750685

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dais-polymtl/flockmtl.

1 INTRODUCTION

Complexity of Workflows. A variety of workflows take the form of knowledge-intensive analytical applications, *i.e.*, they rely on integrating relevant context from both structured and unstructured datasets to support data-driven decision-making. They further rely on analytics, semantic analysis, or a combination of both for further processing. For example, consider an investigative analyst reporting on an inquiry regarding a new vessel offence. The analyst might: i) consult tabular data to obtain specific vessel details; ii) interpret legal documents to assess the severity of the reported offence; iii) aggregate the vessel's history of similar prior offences; and iv) identify and rank potential interventions.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750685

Novel Data Pipelines. The advent of LLMs has led to a technological step change. Their commoditization, starting with the release of GPT-3 [6], enabled the implementation of pipelines that interleave: (i) analytics; (ii) retrieval; and (iii) semantic analysis using LLM predictions. These pipelines use heterogeneous systems, *e.g.*, DBMSs and search engines, follow the RAG framework [5], and may also include tool calling functionality.

Implementation Challenges. The development of such pipelines is reminiscent of the data management practices prior to the era of the relational model. Data engineers currently make many low-level execution decisions, *e.g.*, choosing specific models for predictions, adapting prompts, managing LLM context, caching predictions for reuse, and deciding when and how to incorporate newly released optimizations. Adding to this burden, any change in application requirements in terms of expected quality, latency, or cost requires substantial re-engineering. Beyond these execution decisions, relying on heterogeneous systems results in significant data shuffling and missed co-optimization opportunities. Often, users rely on DBMSs for initial simple querying and resort to re-implementing complex operations within an orchestration layer.

Our Approach. We propose FlockMTL, an open-source extension for DuckDB [8] that can be adapted to other RDBMSs. FlockMTL enables the use of LLMs within scalar and aggregate SQL functions. Users can construct powerful pipelines by leveraging common table expressions (CTEs) to interleave analytics with LLM-chained predictions. Building on the declarative principles of the relational model, FlockMTL applies seamless optimizations to alleviate the burden of execution details for developers and non-experts.

Core Insights. We summarize the design and implementation of FlockMTL (currently `v0.3.0`) through a set of core ideas:

- *Flexible paradigm*: FlockMTL supports a broad range of semantic operations. These are implemented as LLM-powered scalar and aggregate SQL functions that enable tasks such as classification and summarization within RDBMSs. Additionally, FlockMTL introduces some specialized built-in functions, *e.g.*, reranking (FIRST, LAST) and fusion (RRF, COMBANZ, COMBMED, COMBMNZ, COMBSUM) to enable full hybrid search. Table 1 contains a summary of all functions.
- *Resource-independence*: Functions accept model and prompt specifications as inputs. FlockMTL introduces two new DDL resource types: `MODEL`s and `PROMPT`s, treated as first-class schema objects akin to `TABLE`s. This abstraction allows SQL queries to remain fixed within application code while enabling versioned administrative updates to both models and prompts.
- *Seamless optimizations*: Lower-level execution tasks such as LLM context management and batching of multiple input tuples into a single LLM call are handled seamlessly by FlockMTL. This reduces the complexity of integrating semantic operations for developers and data engineers.

Scalar Functions: Map an input tuple to an output value using chat completion API or embedding API.		
Generic	LLM_COMPLETE	uses an LLM and a user prompt to generate *text* or structured output from an input tuple.
	LLM_EMBEDDING	uses an LLM to generate an embedding vector (fixed-length array) from an input text value.
	DATA_FUSION	fuses N scores from N retrievers — choices: RRF / COMBANZ / COMBMED / COMBMNZ / COMBSUM.
Specialized	LLM_FILTER	uses an LLM and a prompt to return *True/False* given an input tuple.
Aggregate Functions: Reduce multiple input tuples to a single output value using the chat completion API.		
Generic	LLM_REDUCE	uses an LLM and a prompt to generate *text* or structured output from multiple input tuples.
Specialized	LLM_RERANK	uses an LLM and a prompt to rank input tuples based on relevance.
	LLM_FIRST/LAST	uses an LLM and a prompt to return the most/least relevant tuple.

Table 1: Summary of the scalar and aggregate functions supported by FlockMTL.

2 SYSTEM OVERVIEW

LLMs and prompting align well with SQL's goal of making data querying accessible to non-experts. Since the majority of enterprise data is stored in RDBMSs, FlockMTL introduces semantic operations and retrieval augmented generation (RAG) within SQL. FlockMTL is built on top of DuckDB's extension module [8].

DuckDB implements a state-of-the-art analytics engine and makes the DBMS internals easily extensible. Its extension module allows changes to SQL parsing, to the optimizer and execution engine, as well as the addition of new data types. DuckDB already has a rich ecosystem of extensions that can be complementary. For instance, its extensions enable the querying of file formats such as *Parquet* and *CSV* as well as attaching to DBMSs such as `PostgreSQL` and `MySQL`. As such, users can write federated queries over multiple data formats and databases. As a DBMS of choice, the capabilities we add within FlockMTL become instantly available across a variety of file formats and databases. FlockMTL also provides an `ASK` functionality to turn a natural language query into SQL augmented with FlockMTL's functions. It is easy to `INSTALL` and `LOAD` FlockMTL as a community extension using:

```
INSTALL flockmtl FROM community; LOAD flockmtl;
```

In the remainder of this section, we provide an overview of FlockMTL's new schema object resources (`MODEL` and `PROMPT`), functions, and optimizations. To illustrate resources and functions, we consider a simple use case involving the following table:

`research_papers`: (`id`, `title`, `abstract`, `content`)

In this scenario, the user is a researcher aiming to identify relevant papers, extract key insights, and generate summaries using SQL.

2.1 Models and Prompts

Users can define reusable resources: `MODEL`s and `PROMPT`s. These resources can be scoped to the current database using the `Local` setting, which is the default, or configured as `Global`, *i.e.*, accessible across all databases on a given machine. FlockMTL `v0.3.0` supports locally hosted open-source models via Ollama, cloud-based models deployed through Azure, and OpenAI or any OpenAI-compatible providers, *e.g.*, Groq. Inference requests are dispatched to the servers using libcurl.

```
-- (1) Define a model to use
CREATE GLOBAL MODEL('model-relevance-check', 'gpt-4o-mini', 'openai')
-- (2) Define a prompt to check if the paper is a join algorithm
CREATE PROMPT('joins-prompt', 'is related to join algos given abstract')
```

Query 1: Prompt and Model Definitions

Query 1 shows how to define a global model named *model-relevance-check*, configured to point to `GPT-4o-mini` by the provider OpenAI. It also includes a local user-defined prompt for identifying papers relevant to join algorithms. Users have the flexibility to modify or delete these resources. When a resource is modified, FlockMTL automatically creates a new version. Previous versions remain available for inspection and use, while the most recent version is applied by default unless specified otherwise.

2.2 Functions

FlockMTL enables semantic analysis through a combination of generic and specialized functions. The generic functions allow users to define operations that map or reduce tuples to text or structured output using LLMs. These are summarized in Table 1. The specialized functions are for common use cases. For example, `llm_filter` predicts boolean values using structured output, and `llm_first`/`llm_last` are an optimized `llm_rerank` to find the most or least relevant tuple. Finally, data fusion functions such as RRF and COMBANZ enable hybrid search and when used with `llm_rerank` and `llm_complete` enable RAG within SQL.

Scalar Functions. These map each input tuple to a new attribute. We showcase in *Query 2* below an example of semantic filtering, summarization, and extraction. *Query 2* identifies papers relevant to join algorithms using `llm_filter`. Then, for each, it summarizes its abstract in a sentence using `llm_complete`, *i.e.*, text prediction, and also extracts keywords (structured output prediction using `response_format`). In summary, the query uses three semantic operations: `llm_filter`, which produces a boolean value; a first `llm_complete` returning a string; and a second returning a JSON-structured object via the chat completion APIs.

```
-- (1) Select papers related to join algorithms
WITH relevant_papers AS (
   SELECT id, title, abstract, content
     FROM research_papers P
    WHERE llm_filter({'model_name': 'model-relevance-check'},
                     {'prompt_name': 'joins-prompt'},
                     {'title': P.title, 'abstract': P.abstract})
),
-- (2) Summarize the paper's abstract
SELECT RP.id, RP.title, llm_complete({'model': 'gpt-4o'},
                                     {'prompt': 'Summarize the abstract in 1 sentence'},
                                     {'abstract': RP.abstract}) AS summarized_abstract,
   llm_complete({'model_name': 'gpt-4o',
     'model_parameters': '{"response_format": {...}}'},
     {'prompt':'Extract the keywords and paper type.'},
     {'title': P.title, 'abstract': P.abstract})
  FROM relevant_papers RP
```

Query 2: Finding relevant join algo papers.

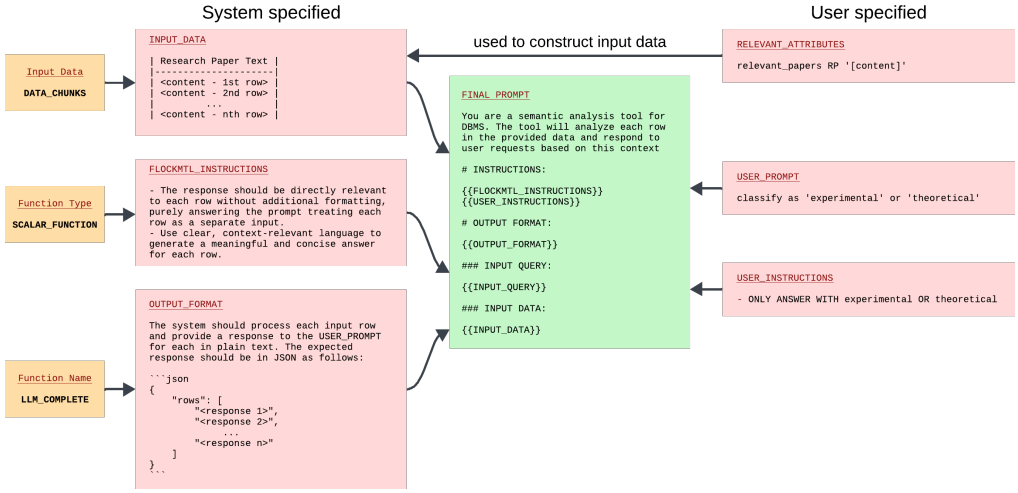


Figure 1: Prompt construction example for *llm_complete*.

Notably, only `llm_filter` uses the predefined model and prompt: `model-relevance-check` and `joins-prompt`; the remaining functions specify these parameters directly within the query. While the use of CTEs in Query 2 is not required, it illustrates how to chain LLM predictions in SQL.

Aggregate Functions and Full Hybrid Search. We showcase in the next query an example of a full hybrid search pipeline within FlockMTL. To our knowledge, this is the first such implementation within SQL. *Query 3* aims to find passages from research papers relevant to *join algorithms in databases*. Among those, it further reranks the results to prioritize passages related specifically to *cyclic join queries*. We consider a table containing previously chunked passages from publications: `research_passages: (id, text)`.

```
-- (1) BM25 retriever over chunked text contents of papers
WITH BM25_Chunks AS (
    SELECT id, text, score_normalized ... LIMIT 100
)
-- (2) Vector similarity search over chunks of paper content
VS_Chunks AS (
    SELECT id, text, score_normalized ... LIMIT 100
)
-- (3) Vector similarity search over chunks of paper content
SELECT COALESCE(bm.content, vs.content) AS passage
  FROM BM25_Chunks BM FULL OUTER JOIN VS_Chunks VS ON BM.idx = VS.idx
 ORDER BY flockmtl_fusion_func(BM.score_normalized, VS.score_normalized)
 LIMIT 10
-- (4) Rerank top 10 elements if they are relevant to cyclic join queries
SELECT llm_rerank({'model_name':'gpt-4o'},
       {'prompt':'studies cyclic join algorithms'}, {'passage': passage});
```

Query 3: Hybrid search to find top 10 passages on cyclic joins

Query 3 is split into multiple steps: (i) retrieve the top-100 relevant passages using BM25, implemented via DuckDB's Full-Text Search extension; (ii) retrieve the top-100 relevant passages using vector similarity search, either through DuckDB's VSS Extension or via a full scan using FlockMTL's `llm_embedding`; (iii) merge the two result sets using a `FULL OUTER JOIN` to align normalized scores, followed by fusion using one of FlockMTL's data fusion methods (*e.g.*, RRF); and finally (iv) rerank the fused results using an LLM-based list-wise aggregation method [7]. Note that `llm_rerank` returns a single value containing a ranked list of tuples within a *JSON*, which must be unpacked for further SQL processing. We omit these details for simplicity of presentation.

2.3 Optimizations

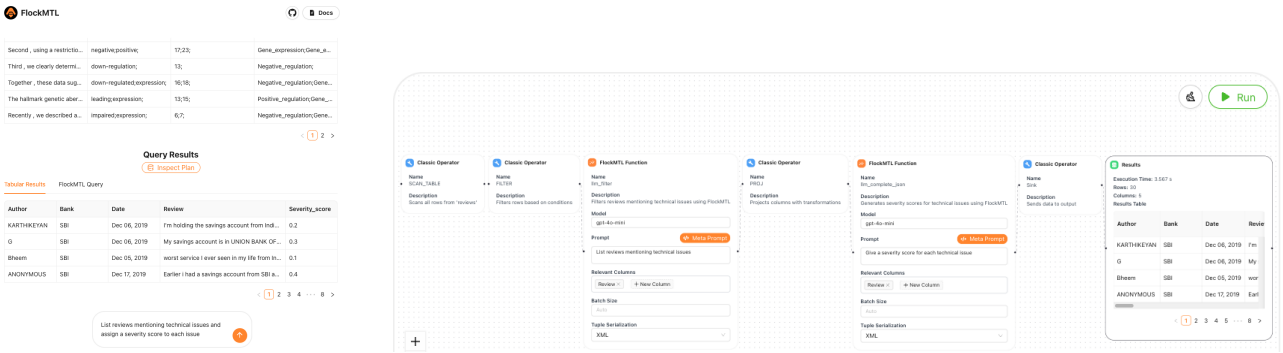
FlockMTL introduces several key optimizations aimed at improving both efficiency and usability, including: (i) meta-prompting for more robust predictions and simplified user queries; (ii) both asynchronous and synchronous function variants; and (iii) batching of tuples into a single prediction to reduce latency.

Meta-prompt. In FlockMTL, users provide prompts intended for a single tuple (for map functions) or for multiple tuple (for reduce functions). The system then constructs a full prompt by embedding the user-defined prompt within a structured meta-prompt template, illustrated in Fig. 1. This template enriches the user input with formatting guidelines, output constraints, and serialized representations of tabular input. The meta-prompt is designed to be KV-cache friendly for efficient LLM inference.

Batching. Although FlockMTL's scalar functions are written as if operating on individual tuples, executing one API call per tuple is inefficient. To address this, FlockMTL automatically batches multiple tuples into a single prompt for inference. The system determines the batch size dynamically, based on the size of the input tuples and the available context window of the LLM. It fills the prompt with as many tuples as possible, then sends a single batched request. If the LLM returns an error due to exceeding the context window, FlockMTL automatically reduces the batch size by 10% and retries. This process continues iteratively until the call succeeds. If even a single tuple causes overflow, its result is set to `NULL`. Users can manually control batching by setting the `batch_size` parameter; setting it to 1 disables batching, which may help mitigate model inaccuracies.

Async/sync variants. If the number of working threads is small, batching can lead to bottlenecks since autoregressive LLMs suffer from high decoding latency due to their sequential token generation. To mitigate this, FlockMTL functions are asynchronous by default, issuing small batches of non-blocking calls within a single thread and blocking per vector of $\sim$ 2048. For cases where synchronous execution is required, each function provides a sync variant, accessible by appending `_s` to the function name (*i.e.*, `llm_function_s`).

Next, we give a general sense of performance. On Kaggle's banking reviews dataset and an M4 Pro Mac machine (model MX2H3LL/A),



(a). Data application with NL interface. **(b). The plan inspection interface.**

Figure 2: Screens of Prepared Demonstration for Users to Get Started.

asynchronous execution using a single thread and OpenAI services, FlockMTLprocesses ~ 17,000 tuples in 121.27 seconds. In this setup, the LLM simply reproduces attributes in a concatenated format. In contrast, using four threads with dynamic batching takes 2047.26 seconds. Finally, no batching leads to over 100× slowdown in performance before a timeout occurs.

3 DEMONSTRATION SCENARIOS

Goal. Our demo has two main goals. First, we aim to showcase how easily users can build data applications using the `ASK` functionality, which combines analytics and semantic operations without requiring the orchestration of multiple external systems. Second, we highlight the importance of FlockMTL's low-level optimizations through an interactive challenge involving the audience.

Interaction. The landing page of the demonstration presents a data application where users can explore and interact with multiple tabular datasets sourced from Kaggle and spanning domains such as biomedical, academic, and product reviews. In this demonstration, we use three kaggle datasets: (i) *Online Banking Review Dataset*; (ii) *arXiv Dataset*; and (iii) *GENIA Biomedical Event Dataset*.

Users begin by viewing a preview of the dataset made of one or more tables. To explore the dataset, attendees can issue a natural language query, *e.g.*, "`list reviews mentioning technical issues and assign a severity score to each issue`" as done in Fig. 2a on for instance the banking review dataset, and FlockMTL's `ASK` functionality automatically generates a SQL query augmented with FlockMTL's functions. Users can inspect the generated SQL query and its output. This part of the demonstration illustrates the power of integrating semantic operations directly into SQL.

Following this, we introduce a challenge to the audience. By clicking on `Inspect Plan` for the generated query shown in Fig. 2a, users are taken to a separate interface for plan debugging and analysis. The separate interface shows the plan of our example query in Fig. 2b. This query plan is produced through a multi-stage process in which our `query plan generator` analyzes both the original query and the plan generated by DuckDB, synthesizing this information to construct a new and editable plan for making detailed changes and optimizations. This query plan includes: (i) standard SQL operations such as scans and filters, and (ii) FlockMTL specific functions, such as `llm_filter` as well as `llm_complete_json`. The FlockMTL function box on the UI contains additional system-level details, such as access to the full meta-prompt used, the serialization format, and the batch size chosen automatically by the system. At this stage, any issues in the generated query, *e.g.*, , in the prompt, can be manually fixed.

In this challenge, users are first presented with the default setting where batch size is set to `Auto`, hiding the one FlockMTL used. They can change it manually and select a different batch size that might match the system's performance and accuracy. The default serialization format shown by default is `XML`, but users may modify it to `JSON` or `Markdown`. For instance, if a user sets the batch size to 30 and reruns the query, they might observe a latency increase from ~3 to ~12 secs, resulting in a 4× slower execution. Each user is given attempts to match the performance and quality achieved by FlockMTL's seamless optimizations. This exercise highlights the trade-offs in latency and prediction accuracy with different parameters. Finally, users are asked to replace the full prompt using a Jinja template, and then compare their version in both structure and output with the meta-prompt used by FlockMTL.

Related work. There is a recent shift from monomodal to multimodal query processing. Prior work includes LOTUS [1], focused on dataframe-APIs and Palimpzest [2], introducing cost-based optimizations. Our work is closest to TAG [4] and Galois [3] and has similar approaches to optimization. FlockMTL introduces new DDL resource types, asynchronous variants of function execution, and its own specific implementation for these optimizations such as batching within the constraints of an RDBMS. We hope it serves as a foundation for further research and system development.

REFERENCES

[1] Asim Biswal et al. 2025. Text2SQL is Not Enough: Unifying AI and Databases with TAG. *CIDR* (2025).
[2] Chunwei Liu et al. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. *CIDR* (2025).
[3] Dario Satriani et al. 2025. Logical and Physical Optimizations for SQL Query Execution over Large Language Models. *SIGMOD* (2025).
[4] Liana Patel et al. 2024. LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data. *CoRR* abs/2407.11418 (2024).
[5] Patrick Lewis et al. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS* (2020).
[6] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. *NeurIPS* (2020).
[7] Xueguang Ma et al. 2023. Zero-Shot Listwise Document Reranking with a Large Language Model. *CoRR* abs/2305.02156 (2023).
[8] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. *SIGMOD* (2019).