SEMA: A High-performance System for LLM-based Semantic Query Processing

Kangkang Qi
Beijing Institute of Technology
China
qikangkang0126@gmail.com

Dongyang Xie
Wenbo Li
Wuhan University
China
xdy1122310013@163.com
liwenbo82@whu.edu.cn

Hao Zhang
The Chinese University of Hong Kong
Hong Kong SAR, China
zhanghaowuda12@gmail.com

Yuanyuan Zhu
Wuhan University
China
yyzhu@whu.edu.cn

Jeffrey Xu Yu
HKUST (Guangzhou)
China
jeffreyxuyu@hkust-gz.edu.cn

Kangfei Zhao*
Beijing Institute of Technology
China
zkf1105@gmail.com

ABSTRACT

The integration of Large Language Models (LLMs) into data analytics has unlocked powerful capabilities for reasoning over bulk structured and unstructured data. However, existing systems typically rely on either DataFrame primitives, which lack the efficient execution infrastructure of modern DBMSs, or SQL User-Defined Functions (UDFs), which isolate semantic logic from the query optimizer. The LLM-powered semantic operators also bring new challenges due to the high cost and non-deterministic nature of the LLM invocation, where conventional optimization rules and cost models are inapplicable for their optimization. To bridge these gaps, we present SEMA, a high-performance semantic query engine built on DuckDB that treats LLM-powered semantic operators as first-class citizens. SEMA introduces SEMASQL, a declarative dialect that allows users to seamlessly inject natural language expressions into standard SQL clauses, enabling end-to-end optimization and execution. At the logical level, the optimizer of SEMA compresses natural language expressions and deduces relational constraints from semantic operators. At runtime, SEMA employs Adaptive Query Execution (AQE) to dynamically reorder operators, fuse semantic operations, and apply prompt batching. This approach seeks a Pareto-optimal execution path that minimizes either token consumption or latency according to user preference, subject to accuracy constraints. We evaluate SEMA on 26 semantic queries across classification, summarization, text extraction, and ranking tasks. Experimental results demonstrate that SEMA achieves $2-10\times$ speedup against three baseline systems while achieving competitive result quality.

PVLDB Reference Format:
Kangkang Qi, Dongyang Xie, Wenbo Li, Hao Zhang, Yuanyuan Zhu, Jeffrey Xu Yu, and Kangfei Zhao. SEMA: A High-performance System for LLM-based Semantic Query Processing. PVLDB, 19(11): 3231-3244, 2026.
doi:10.14778/3836663.3836685

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836685

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BITQiKangK/Sema.git.

1 INTRODUCTION

The powerful semantic reasoning capabilities of modern large language models (LLMs) are unlocking new opportunities for AI-driven analytics over structured and unstructured data. Applications in science, healthcare, and business increasingly require complex reasoning over large datasets, e.g., summarizing recent research, extracting biomedical insights from patient records, or analyzing meeting transcripts at scale. To realize these capabilities, analytics systems need to orchestrate bulk semantic processing with efficiency and usability, promising a revolution in existing database systems. This vision raises two fundamental questions: *How should developers express semantic queries? And how should developers design an analytics system to achieve high efficiency and accuracy?*

Recent systems have begun to explore LLM-powered semantic query processing for both structured [14, 30] and unstructured data [2, 40, 43]. A variety of systems DocETL [40], ZenDB [27], DocWanger [41], Aryn [2] and Unify [43] support semantic queries expressed in extended SQL, declarative operations, or natural language. These systems employ LLMs not only to interpret queries, but also to optimize execution through techniques like query planning [4], rewriting, prompt refinement, and operation decomposition [41]. Palimpzest [29] and its optimizer Abacus [39] focus on reducing LLM query latency and cost through logical and physical optimizations, including Pareto-efficient planning. Lotus [34] formulates LLM-based semantic operations for tabular data and proposes efficient execution algorithms for costly operators. Recent benchmarks [1, 14] have highlighted key challenges in supporting semantic query processing in databases, such as ensuring structured outputs, enabling batch processing, and effective query planning.

Current systems typically express semantic queries in two different ways: DataFrame primitives [29, 34, 39] and SQL user-defined functions (UDFs) [1, 14, 30], both highly intertwined with the underlying system design. Systems built on DataFrame primitives define custom semantic operations and optimize them with specialized algorithms, but often lack the efficient execution infrastructure of a

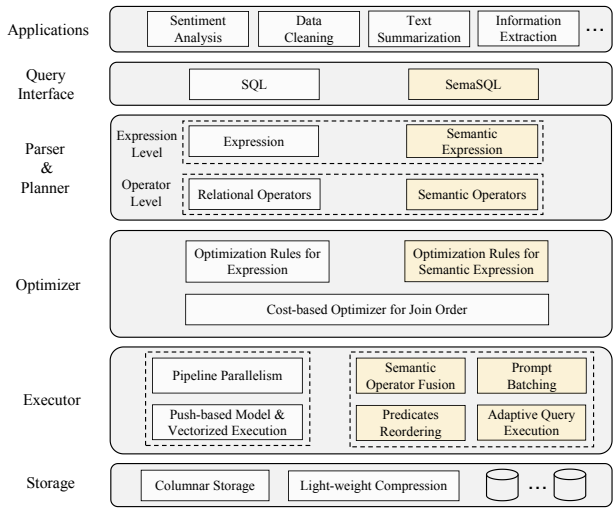

Figure 1: System architecture of Sema

DBMS. This forces developers to redeploy execution optimization such as vectorized and pipeline parallelism. On the other hand, integrating semantic operators as SQL UDFs offers greater flexibility, but places the burden of optimizing complex semantic workloads on users, raising concerns around reliability and efficiency.

With the growing demand for semantic queries, a core set of semantic operators, such as semantic filter, projection, join, and aggregation, has emerged as an AI-powered analogy to classic relational operators [34, 43]. A natural envision arising from this trend is *whether we can efficiently support these semantic operators as native database operators, thereby fully leveraging DBMS optimization?* However, fulfilling this envision is not merely about implementation add-ons. Unlike relational operators, which have deterministic semantics, stable algebraic laws, and cost models approximated from data statistics, semantic operators are context- and model-dependent: their execution depends not only on input tuples, but also on the natural language (NL) expression, prompting strategy, and the specific LLM being invoked. LLM inference optimizations in external LLM serving environments, such as KV cache and continuous batching, also significantly impact execution performance. Modeling the cost of these LLM-powered operators is also beyond the modeling of I/O bounded operators [15]. This shift breaks key assumptions behind traditional query optimization: standard rewrite rules may be unsafe, and selectivity and cost cannot be estimated statically. These challenges call for new system-level optimizations tailored to semantic operators. The system should treat semantic operators as first-class citizens and focus on reducing the cost of LLM invocations, which dominate the performance of query evaluation. Furthermore, semantic operators also bring additional concerns of multiple optimization objectives, including query accuracy, latency, and the monetary cost of LLM invocation.

As a tentative realization, we design and develop a prototype system dubbed Sema back-ended DuckDB [36], injecting native semantic operators that invoke local or remote LLMs. On the basis of vectorized and pipeline execution infrastructure, the design principle of Sema focuses on reducing the cost and latency of LLM invocations. Figure 1 illustrates the architecture of Sema, which extends the query interface, parser, planner, optimizer, and executor of DuckDB accordingly. To express semantic operators, Sema introduces SemaSQL that allows users to insert Natural Language (NL) expressions into SQL clauses. These NL expressions, wrapping table's column names, convey user intent on semantic predicates, ranking, or aggregation criteria. To process these native semantic operators efficiently and flexibly, we propose logical intra-operator optimizations in the optimizer for compressing the NL expressions and deducing relational operators from semantic operators, where the optimizer leverages auxiliary LLMs to fulfill these optimizations. In the executor, Sema integrates semantic operator fusion and prompt batching to reduce the number of LLM invocations in bulk data processing. As the statistical patterns of semantic operator instances are hard to depict before execution, Sema adopts adaptive query execution (AQE) to reorder semantic filters and projections and decides potential operator fusion and prompt batching. The AQE will search for a Pareto-optimal execution path that minimizes either query latency or token cost, subject to a user-specific accuracy deviation. We comprehensively evaluate Sema on 26 semantic queries spanning 4 analytical tasks, including classification, summarization, keyword extraction, and ranking. The results demonstrate that Sema achieves a 2–4× and 2–10× speedup over DataFrame-based systems Lotus and Palimpzest, respectively, while maintaining competitive result quality compared with these two systems and significantly outperforming the UDF-based system FlockMTL. The contributions of this paper are summarized as follows.

- We design and develop Sema, a high-performance semantic query processing system built on DuckDB that treats LLM-powered semantic operators as first-class query plan operators, enabling end-to-end semantic query optimization and execution in a columnar and vectorized database system.
- We propose SemaSQL, a declarative SQL dialect that allows users to inject natural language expressions into standard SQL clauses to define semantic operators, enabling the seamless composition of semantic and relational computation.
- We introduce optimization techniques tailored to semantic operators to reduce the overhead of LLM invocations collaboratively, including logical-level semantic expression optimizations and runtime optimizations via adaptive query execution (AQE) that combines operator fusion and prompt batching.
- We design a semantic query benchmark consisting of 26 queries over 8 datasets and conduct a comprehensive evaluation. Results demonstrate that Sema significantly outperforms baseline systems in query efficiency with competitive result quality, and validates the effectiveness of its optimization strategies.

Roadmap. The following of this paper is organized as follows. We introduce the background and our motivation in §2. §3 introduces the system overview of Sema, followed by the details of its optimizer and executor in §4 and §5, respectively. §6 elaborates on critical engineering details of Sema. We report our system evaluation in §7 and review related works in §8. We conclude the paper in §9.

2 BACKGROUND

We introduce the preliminaries of semantic operators in §2.1 and existing solutions for semantic query processing in §2.2.

2.1 Semantic Operators

A semantic operator is a declarative transformation on tabular data, which is parametrized by a natural language expression as a predicate. In this paper, we consider semantic operators powered by Large Language Models (LLMs). We use $T(A_1, \cdots, A_j)$ or T for short to denote a table, where $A_1, \cdots, A_j$ are the attributes of T. $t \in T$ denotes a tuple in table T and $t(A_j)$ denotes the value of attribute A_j in tuple t. $M : \mathcal{T} \mapsto \mathcal{T}$ is a pre-trained LLM that accepts input and generates output in textual space $\mathcal{T}$. Sema starts from a relational algebra generalization on semantic query processing, a core set of semantic operators in Lotus [34], while other specific systems may provide additional operators beyond these operators.

Table 1 summarizes these five semantic operators we implemented in Sema, i.e., Semantic Filter (*SemFilter*), Semantic Projection (*SemProj*), Semantic Join (*SemJoin*), Semantic OrderBy (*SemOrderBy*), and Semantic Aggregate (*SemAgg*). Each operator is implemented by a default reference algorithm. For *SemFilter* and *SemProj*, their reference algorithms scan the table and invoke an LLM to evaluate each tuple in table T, and the LLM is requested to return True or False for *SemFilter* or a textual attribute for *SemProj*, respectively, which takes $O(|T|)$ LLM invocations. For *SemJoin*, the reference algorithm performs a nested loop join on two tables T_1 and T_2, and uses an LLM to evaluate whether a pair of tuples can be joined on a semantic predicate, which takes $O(|T_1| \cdot |T_2|)$ LLM invocations. The reference algorithm of *SemOrderBy* is a Merge Sort algorithm that compares two pairs of tuples in T regarding a semantic comparison criterion, resulting in $O(|T| \log |T|)$ LLM comparisons. For *SemAgg*, the reference algorithm performs a many-to-one reduce that puts all the values of an attribute in T into an LLM prompt and generates the aggregated result. Managing finite context limits of the underlying LLM is implemented by the system, where hierarchical aggregation over subsets of values can be performed. Similar to relational operators, the evaluation of one semantic operator can be implemented by different algorithms; however, supporting LLM-based semantic operators inside a database engine is non-trivial, not simply an implementation.

2.2 Semantic Operators Handling

Unlike relational operators, whose behavior is grounded in deterministic and decidable semantics, well-defined algebraic laws, and stable cost models, semantic operators are inherently context- and model-dependent. Their outcomes depend not only on input tuples but also on the NL expression, prompting strategy, and the specific LLM invoked. As a result, evaluation is intrinsically uncertain and cannot be characterized by a single canonical truth function. This shift in semantics also weakens the foundations that classical query optimizers rely on. Standard algebraic rewrite rules may not hold: for example, *SemFilter* is commutative only under restrictive conditions (e.g., tuple-independent prompting and zero-temperature inference), while associativity does not generally apply. For *SemProj* and *SemJoin*, commutativity and associativity are typically not semantics-preserving, which makes traditional rule-based rewriting unsafe or inapplicable. Finally, the cost profile of semantic operators is dominated by external inference environments, i.e., latency, throughput, and monetary cost vary with models, prompts, inference optimization, and caching mechanisms; therefore, they are difficult to estimate from database statistics alone. Together, these properties call for a system design that treats semantic operators as first-class citizens rather than black-box add-ons.

Existing solutions for semantic query processing are constrained by their execution environments. Implementations built on lightweight libraries (e.g., Pandas or custom DataFrame APIs) provide accessible programming models but lack mature database capabilities such as vectorized execution, pipeline parallelism, and robust query optimization, resulting in inefficient processing for large-scale, hybrid analytical workloads. In contrast, database-centric approaches frequently encapsulate LLM calls as user-defined functions (UDFs). While convenient, this makes semantic operators opaque to the optimizer and executor, preventing system-level optimizations tailored to semantic workloads.

Motivated by these gaps, we built Sema, a prototype system that integrates semantic operators directly into DuckDB to process semantic operators at scale by exploiting a high-performance OLAP execution engine with an expression-centric design, vectorized processing and pipeline parallelism. By embedding semantic primitives into the query planner, optimizer, and executor, Sema provides a concrete platform for exploring semantic-specific optimization, i.e., expression optimization and adaptive query execution, aiming to reduce the overhead of LLM invocations. In this work, we focus on the system design and end-to-end integration of semantic operators in an mature OLAP engine. We do not aim to fully optimize each individual operator or cover the full space of production concerns. We leave the following to future work: specialized algorithmic improvements for individual operators, storage-level optimizations, LLM routing and model selection, and comprehensive mechanisms for privacy, governance, and fault tolerance in external inference.

3 SYSTEM OVERVIEW

We present an overview of Sema, including query interface (§3.1) and workflow (§3.2). Figure 1 shows the system architecture.

3.1 The Query Syntax

Sema introduces SemaSQL, a declarative syntax with injection of Natural Language (NL) expression, for semantic query processing. The design of SemaSQL follows three main principles: *Transparency*: isolation from underlying implementation and optimization, *Flexibility*: sufficient expressive capability of user intent with high utility, *Compatibility*: extension of SemaSQL compatible with the syntax standard and implementation of SQL.

The first-class citizen of SemaSQL is an NL expression with one or multiple placeholders, denoted as $s`\cdots\{\}\cdots`$ where $\{\}$ is a placeholder that identifies a mention of columns/attributes, instructing the system to process the corresponding tables and values. Semantic operators are implicitly implemented by injecting NL expressions into SQL clauses such as `SELECT`, `WHERE`, `JOIN ON`, etc. Specifically, *SemProj* injects an NL expression into `SELECT` and *SemFilter* injects an NL expression into `WHERE` or `HAVING` clauses, respectively. For *SemJoin*, we inject an NL expression into a `JOIN ON` clause, which is also compatible with other join types, including `OUTER JOIN`, `SEMI JOIN`, etc. For *SemAgg*, we introduce a special aggregation function `sem_agg()` that wraps the NL expression, and `sem_agg()` can be used together with the standard `GROUP BY`

Table 1: Semantic operators in SEMA

Operator	Expression	Description	# LLM calls
Semantic Filter	$\{t \mid t \in T \wedge M(t,e) = \text{True}\}$	Select tuples that can pass a NL predicate e	$O(\lvert T\rvert)$
Semantic Projection	$\{M(t,e) \mid t \in T\}$	Project attribute values specified by a NL predicate e	$O(\lvert T\rvert)$
Semantic Join	$\{(t_i,t_j) \mid t_i \in T_1 \wedge t_j \in T_2 \wedge M(t_i,t_j,e) = \text{True}\}$	Join two tables on a join key that pass an NL predicate e	$O(\lvert T_1\rvert \cdot \lvert T_2\rvert)$
Semantic Order By	$[t_1,\cdots,t_n], s.t. \forall [t_i,t_j] \Rightarrow M(t_i,t_j,e) = t_i \preceq t_j$	Reorder tuples following a NL-specified criterion	$O(\lvert T\rvert \log \lvert T\rvert)$
Semantic Aggregate	$M(\{t_1,\cdots,t_n\},e), t_1,\cdots,t_n \in T$	Aggregate tuples according to a NL-specified aggregate function	$O(\log \lvert T\rvert)$

clause. In addition, users can use AS to rename new textual columns, e.g., the columns generated from *SemProj* and *SemAgg*, and these columns can be further mentioned in other NL expressions. Figure 2 demonstrates an SQL query containing two predicates of *SemFilter* in the WHERE clauses. The query performs a relational join on two tables, with two *SemFilter* operators on the `translated_review` attribute of table `user_reviews`. Here, NL expressions are used as the semantic predicates whose values are inferred by an LLM.

SEMASQL uses interpolation of NL expressions to support semantic operators while preserving the nature of the symbolic logic of SQL. SEMA also supports complex syntax with SEMASQL such as subqueries, Common Table Expressions (CTEs), and window functions, etc. Unlike NL2SQL, SEMASQL makes minimal extensions, i.e., the NL expression on SQL, enabling full compatibility with RDBMS and sufficient semantic reasoning.

3.2 System Workflow

We use the running example in Figure 2 to illustrate the end-to-end workflow of SEMA, which is depicted in Figure 3. The query counts, for each app, the reviews that satisfy two semantic predicates ('is a valid user review' and 'is a positive user review'), together with standard relational predicates on category and type. When the user submits the query, SEMA parses SEMASQL and produces an abstract syntax tree (AST) and an initial logical plan in which NL expressions are identified as explicit semantic operators. Here, two *SemFilter* operators become first-class plan nodes that SEMA can analyze and transform at the logical and physical levels, respectively.

First, logical NL expression optimization in the optimizer operates before conventional SQL rewrites and focuses on reducing and restructuring intra-operator semantic expressions. Specifically, SEMA applies redundancy-aware expression compression to shorten NL predicates and stabilize prompting. More importantly, SEMA performs predicate deduction for *SemFilter*: it opportunistically extracts fully or partially deducible symbolic conditions (e.g., 'translated_review != "nan"') from an NL predicate, rewrites them into relational filters which are pushed down afterwards to prune tuples early, thereby avoiding unnecessary LLM invocations. These expression optimizations are guided by an auxiliary LLM with CoT prompting. §4 will elaborate on these optimizations injected as special rules for semantic expressions in the optimizer.

Then, physical execution optimization and adaptive query execution (AQE) target the dominant runtime cost of semantic operators, under a vectorized, pipeline-parallel execution engine. The executor of SEMA applies semantic operator fusion and prompt batching to reduce the number of LLM calls and improve throughput. Since the effects of these optimizations and the selectivities of *SemFilter* are hard to predict statically, SEMA introduces AQE for *SemFilter* and *SemProj* operators inside an execution pipeline. At runtime,

```
-- Calculate the counts of valid and positive reviews for each app of
↪  'Free' type and 'ART_AND_DESIGN' category.
SELECT ur.app, COUNT(*) AS positive_and_valid_count
FROM user_reviews AS ur INNER JOIN playstore AS p ON ur.app = p.app
WHERE p.category = 'ART_AND_DESIGN' AND p.type = 'Free'
    AND s'{translated_review} is a valid user review'
    AND s'{translated_review} is a positive user review'
GROUP BY ur.app;
```

Figure 2: A running semantic query example

the executor micro-executes a small fraction of the input to collect predicate statistics, explores multiple candidate execution paths, and selects a path that minimizes either token cost or query latency in the Pareto frontier, subject to an accuracy deviation tolerance. The selected path is then used to process the remaining data at scale. We will present the technical details in §5.

4 THE OPTIMIZER OF SEMA

Aligning with the goal of our system design, SEMA's optimizer concentrates on reducing the cost of LLM invocations without compromising accuracy. As users may deliver arbitrary NL expressions in the semantic operators within SEMASQL, NL expressions are inherently verbose or ambiguous, and their evaluation will bring redundant overhead for LLM reasoning. To resolve these issues, SEMA's optimizer introduces specific rules for optimizing these NL expressions, including expression compression and predicate deduction, which are applied to the query plan step by step. Since NL expressions are free text, traditional rule matching cannot comprehensively capture the compression and deduction patterns, therefore, we use LLM-assisted optimizations.

NL Expression Compression. For the NL expression in each semantic operator, the optimizer first employs a light-weight LLM with Chain-of-Thought (CoT) prompting [44] containing demonstrations to compress and refine the NL expression. The CoT prompting includes decomposed instructions on removing stop words, simplifying sentence structure, e.g., passive to active voice, and consolidating repeated pronoun references to the placeholder, which are listed as multiple rules. We also provide demonstrations for each rule to illustrate its utility. This step introduces only one LLM call for each semantic operator, while potentially reducing LLM inference cost and improving semantic clarity.

Predicate Deduction. After expression compression, the optimizer seeks the opportunity to deduce semantic filters into relational filters, i.e., rewriting entire or partial NL expressions of *SemFilter* as SQL predicates. An entire deduction bypasses LLM invocations,

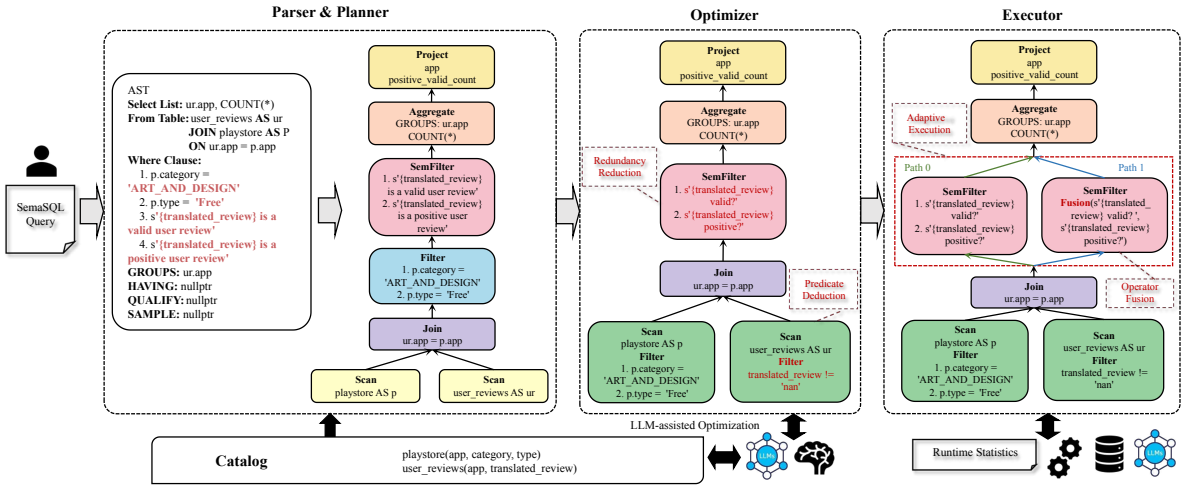

Figure 3: System workflow of Sema

while a partial deduction only surgically replaces partial filter predicates in compound logic with SQL predicates as a necessary condition, preserving the remaining semantics as more concise NL expressions. This partial deduction enables potentially fewer LLM invocations, with the deduced SQL predicates pushdown. Predicate Deduction for *SemFilter* is analogous to predicate simplification for relational query processing, and a similar idea has also been introduced into ML-based predicates [9]. It is worth mentioning that existing NL2SQL approaches [12] cannot directly perform this task. Rule-based approaches fully rely on rule matching for SQL translation, which do not possess flexible reasoning ability for semantic expression. DL-based approaches rely on high-quality annotated data for training. LLM-based approaches have the potential to deduce the NL predicates; however, they cannot support partial deduction, i.e., translating a SQL predicate as a necessary condition, with a corresponding verification mechanism.

Sema supports deducing predicates of categorical matching, numeric comparison, and LIKE operations on strings. First, we use CoT prompting, feeding the DB schema into the context, to guide an LLM to opportunistically identify deductible predicates and then transform them into SQL expressions in string format. The instruction is explicitly decomposed into multiple steps, starting from extracting relevant tables and columns from the expression, determining if the query possesses objective intent, e.g., numeric or string comparison, and whether it can be mapped to a pattern of symbolic comparison, subsequently judging deductibility, and finally generating a SQL statement. In each step, we provide few-shot examples, which also demonstrate special cases of case variations, fuzzy matches, and data type consistency. To alleviate LLM hallucination, we equip a back-forward self-reflection mechanism. Once a syntactically valid SQL statement is deduced, we request the LLM with another CoT prompt to judge whether the SQL statement is a necessary condition for the original NL expression. If the deduction of the self-reflection fails, the system falls back to the original *SemFilter*, preserving correctness.

Example 4.1. (Expression Compression and Predicate Deduction) We use the query in Figure 2, which contains two *SemFilter* operators on the attribute translated_review in table user_reviews. The parser of Sema transforms the query into an initial unoptimized plan shown as Figure 3. In the optimizer, the two NL expressions are first compressed, forming two shorter sentences, respectively. As the placeholders already wrap exact matches of column names, meta-entity resolution directly binds them to the column objects. Subsequently, predicate deduction extracts a SQL predicate 'translated_review != "nan"' from the *SemFilter* with the expression 's{translated_review} is a valid user review'. Here, an auxiliary LLM is prompted with the column metadata and statistics as context, which indicate that for column translated_review, around 41.8% of entries are 'nan'. The 'nan' is obviously an invalid review, but it may not be noticed by the users. Predicate deduction automatically identifies this symbolic logic from a *SemFilter*, and the deduced SQL predicate is further pushed down to enable early filtering, as shown in Figure 3, bypassing a large number of LLM invocations.

In the optimizer of Sema, the LLM-assisted expression optimizations are applied before rule-based optimization for standard SQL. By strategically employing LLM-assisted expression optimizations, the optimizer improves the usability of SemaSQL for general users to access their analytical tasks and enhances the reliability, accuracy, and efficiency of Sema. We provide an algorithm in our technical report [35] to present the overall procedure of expression optimizations. After LLM-assisted expression optimizations and general logical optimizations of SQL, the plan will be fed into the executor for execution optimization.

5 EXECUTION OPTIMIZATION

The executor of Sema integrates new physical optimizations (§5.1) and Adaptive Query Execution (§5.2), which are specifically designed for semantic operators and semantic query workloads.

Table 2: Operator fusion patterns in Sema

# LLM calls	$\sigma_a \oplus \sigma_b$	$\Pi_a \oplus \Pi_b$	$\sigma_a \oplus \Pi_b$	$\Pi_a \oplus \sigma_b$
Original	$2\|T\|$	$2\|T\|$	$(1+s)\cdot\|T\|$	$(1+s)\cdot\|T\|$
Fused	$\|T\|$	$\|T\|$	$\|T\|$	$\|T\|$

5.1 Physical Optimization

Semantic Operator Fusion. In semantic query evaluation, each semantic operator typically triggers an individual LLM per tuple. While this preserves modularity and operator independence, for complex queries with cascaded semantic operators, intermediate data will be repeatedly materialized and transferred between multiple operators, leading to a high token consumption, and API call latency. Inspired by operator fusion for relational operators [32] and linear algebra operators [7], Sema introduces semantic operator fusion, which combines consecutive semantic operators into one operator. Unlike fusing conventional operators, which synthesizes generated code by compiling optimizations, fusing semantic operators is essentially combining their semantic NL expressions into one NL expression, with one LLM invocation per tuple.

Given the logical plan from the optimizer, Sema supports four patterns of operator fusion as listed in Table 2, between two 'unary' semantic operators *SemFilter* and *SemProj* on one table in one query pipeline. We bootstrap semantic operator fusion from these four patterns, since the fusion of more operators that include both unary and binary operators entails parsing and combining complex semantics to guarantee the evaluation accuracy. For *SemFilter*-first operators, the NL expressions of the two operators to be fused should refer to a common attribute in their placeholders. For *SemProj*-first operators, the rename operator AS indicates the intermediate column, which is referred to by the expression of the subsequent operator. Once Sema identifies a match of these patterns, it concatenates the two NL expressions of the two consecutive operators into a single NL expression and adds an instruction for step-by-step processing.

As Table 2 summarizes, a fusion will save $|T|$ LLM invocations for *SemProj*-first patterns and $s \cdot |T|$ LLM invocations for *SemFilter*-first patterns, respectively, where s is the selectivity of the first *SemFilter* in table T.

Example 5.1. (Semantic Operator Fusion) We use the query in Figure 2 to illustrate operator fusion, which contains two *SemFilter* operators referring to the same column translated_review in table user_reviews. Sema combines the two compressed expressions s'{translated_review} is valid' and s'{translated_review} is positive' into a single expression 's{translated_review} is valid and positive', and then generates a new *SemFilter* for the expression to replace the original two operators.

Prompt Batching. As an analytical system, the vectorized execution of DuckDB is the cornerstone of Sema. On this basis, Sema further considers improving token utilization and throughput of semantic operators with prompt batching, i.e., batching multiple tuples into an invocation [26], for *SemFilter*, *SemProj*, and *SemJoin*.

Despite the reduction of LLM calls, operator fusion and prompt batching break the operator-independent and tuple-independent execution. A potential risk is that they usually lead to inconsistent inference results compared to one-by-one LLM invocations. Taking this issue into account, first, for prompt batching, the executor exploits the structured output of LLM API calls that specify the return value of an API call as a Json Array. This enforces the output of one batch prompt to generate a consistent number of results. Second, when fusing two operator instances, Sema adds an instruction in the combined prompt to instruct the LLM to explicitly generate intermediate results of the first operator, in order to maintain the evaluation consistency of the fused operator to the original operator flow. Sema restricts operator fusion to only two operator instances, leaving the fusion of more operators for further work. Third, for AQE, Sema allows users to specify the tolerance regarding the consistency of results. This user preference will be used to choose the real execution plan by adaptive query execution at runtime, which we will elaborate on in §5.2.

5.2 Adaptive Query Execution

The behavior of semantic operators depends on both the NL expression and the particular LLM used, making key planning quantities, such as selectivity, latency, and token cost, difficult to predict statically. Moreover, optimizations that reduce LLM calls (e.g., operator fusion and prompt batching) can introduce hard-to-model accuracy shifts. To address these challenges, we postpone inter-operator optimizations, such as operator reordering and fusion, to runtime through an adaptive query execution (AQE) strategy. Specifically, the executor collects runtime feedback by evaluating a small fraction of data on multiple candidate physical plans, a.k.a. execution paths, at the beginning, and selects the optimal path to process the remaining data. Unlike SQL query processing, where latency is typically the unique optimization goal, semantic query processing entails considering multiple objectives, including query accuracy, query latency, and the monetary cost of LLM invocations.

Accordingly, AQE is designed to find the Pareto optimal path that minimizes either query latency or monetary cost, subject to a user-specified tolerance to accuracy deviation. The optimization is performed over a finite set of candidate paths that are enumerated by applying operator fusion and/or batching on the optimized logical plans. Since ground-truth labels for semantic queries are generally unavailable beforehand, Sema supports three accuracy evaluation modes at runtime: (1) a reference path (e.g., the optimizer-generated plan without fusion or batch prompting) that serves as a proxy ground-truth by default; (2) a set of development tuples with associated ground-truth annotations; and (3) online LLM-as-a-Judge evaluation, which is suitable for large-scale workloads where judging overhead is amortized over subsequent execution.

The AQE framework operates in three phases: expression exploration, path exploration, and path exploitation. In the expression exploration phase, we uniformly sample a small fraction of δ_1 tuples to collect the statistics of each *SemFilter* and *SemProj* using the reference path. In the path exploration phase, candidate paths $\mathcal{P}$ are generated from these statistics and evaluated on another sampled fraction δ_2 of tuples (or the development tuples if available), and the optimal path $p^\star \in \mathcal{P}$ is selected regarding either time-efficiency or token-efficiency preference. In the path exploitation phase, the remaining fraction of $1-(\delta_1+\delta_2)$ tuples is processed by $p^\star$. Users can balance exploration and exploitation by tuning δ_1 and δ_2. The final query result is the concatenation of outputs from

all three phases. The complete algorithmic procedure is shown in the technical report [35].

Phase 1: Expression Exploration. Given a sequence of *SemFilter* operators $\{\sigma_1, \ldots, \sigma_{n1}\}$ and *SemProj* operators $\{\Pi_1, \ldots, \Pi_{n2}\}$, Sema executes each operator independently on a sampled fraction δ_1 of tuples, producing an execution result R_i (represented as a label vector), and a selectivity estimate $\hat{s_i}$ for each *SemFilter*. The results R_i will be used to evaluate the potential of operator fusion in Phase 2. Intuitively, the higher the correlations between two operators, the more likely the fusion can preserve the semantics. Sema adopts Adjusted Mutual Information (AMI) [33] to quantify this correlation:

$$\mathrm{AMI}(R_i, R_j) = \frac{\mathrm{MI}(R_i, R_j) - \mathbb{E}[\mathrm{MI}(R_i, R_j)]}{\mathrm{avg}(H(R_i), H(R_j)) - \mathbb{E}[\mathrm{MI}(R_i, R_j)]}, \quad (1)$$

where $\mathrm{MI}(\cdot)$ denotes mutual information and $H(\cdot)$ denotes entropy. AMI ranges from 0 to 1; values close to 1 indicate strong correlation. As collected from uniformly sampled data, these statistics are unbiased estimates of the whole dataset [28].

Phase 2: Path Exploration. Based on the statistics from Phase 1, Sema generates a set of candidate paths $\mathcal{P}$, including the reference path p_{ref} that preserves the user-specified operator order and a base path p_{base} that reorders *SemFilter* ascendingly by selectivity. For operator fusion, if $\mathrm{AMI}(R_i, R_j)$ exceeds a threshold τ, the two operators are merged into one with selectivity $s_{i\oplus j} = \min(s_i, s_j)$, where selectivity of *SemProj* is regarded as 1. To prevent accuracy degradation and combinatorial explosion, at most one fusion is introduced per candidate path, yielding up to $n(n-1)/2$ fused candidates with $n-1$ operators ordered by selectivity. For all paths, including fused, reference, and base, counterpart paths with batch prompting are also generated.

The executor then evaluates these candidate paths on a fraction δ_2 of tuples, collecting latency, token consumption, and results. By default, the results of p_{ref} without batch prompting serve as proxy ground-truth; any candidate paths whose accuracy (e.g., F1) falls below a threshold τ_{acc} are discarded. Two dynamic pruning strategies reduce exploration overhead. Specifically, first, if the reference path with batch prompting fails the accuracy threshold, all batch prompting variants are skipped. Second, if a fused path fails, its batch prompting counterpart is bypassed. After evaluation, the Pareto frontier over latency and token cost is computed, and the latency-first or cost-first optimal path is selected as $p^\star$ according to user preference.

Phase 3: Path Exploitation. The selected path $p^\star$ processes the remaining data, ensuring predictable performance while amortizing exploration cost.

AQE currently is applied to classification tasks of *SemFilter* and *SemProj* operators within a execution pipeline. Supporting non-classification *SemProj*, which produces free-format text, would require new correlation metrics beyond AMI. Pipeline-breaking operators such as *SemJoin*, *SemAgg*, and *SemOrderBy* introduce substantial challenges, as they define pipeline boundaries and involve blocking execution. Jointly optimizing such operators across pipelines remains an important direction for future work. We present a case study of AQE in §7.5 on a specific query with three *SemFilter* operators, serving as an illustrative example.

6 ENGINEERING DETAILS

We present key engineering details of Sema, focusing on concurrent LLM calls and pipeline-parallel adaptive query execution (AQE).

Concurrent LLM Invocations. To integrate LLM invocation into the system, Sema extends DuckDB's pipeline-parallel engine with a thread-local concurrent I/O design. DuckDB uses a morsel-driven execution framework [24], decomposing work into fine-grained morsels, each processing a small batch of tuples within a pipeline. A central scheduler dynamically assigns morsels to worker threads from a shared pool, ensuring balanced load distribution with minimal synchronization overhead. In Sema, each worker thread maintains a dedicated CURL multi-handle instance as its private HTTP connection pool. During execution, the thread synchronously issues LLM API requests within the morsel loop, while the CURL multi-handle enables non-blocking progress and efficient connection reuse. This design avoids cross-thread synchronization, allowing multiple workers to issue LLM requests concurrently.

Pipeline-Parallel AQE. Sema integrates AQE directly into the execution engine via a customized blocking operator, *PhysicalSemAdaptive*, which distributes the adaptive three-phase execution across DuckDB's standard source and sink interfaces. The source interface implements the first two phases—expression and path exploration. It buffers incoming tuples from upstream operators, randomly shuffles them to enable uniform sampling, and feeds the sampled subset into the exploration logic to select the optimal execution path $p^\star$. Once the best path is determined, the sink interface carries out the third phase, i.e., path exploitation, to process the remaining data in a pipeline-parallel execution across multiple worker threads. This design integrates AQE into the existing system by simply reusing the established operator interface conventions, requiring no modifications to the pipeline scheduling or execution infrastructure. However, the current blocking-operator formulation cannot readily accommodate pipeline-breaker operators such as *SemOrderBy* and *SemAgg*, where adaptive decisions must span across pipeline boundaries. Supporting such cross-pipeline adaptivity would require introducing a globally shared state that coordinates decisions across pipelines. Extending AQE framework to handle cross-pipeline adaptivity remains future work.

7 SYSTEM EVALUATION

We present the experimental setup in §7.1 and report our comprehensive evaluation in the following facets: ① Compare Sema with baseline systems on output quality, efficiency (§7.2); ② Explore the utility of LLM-assisted expression optimizations (§7.3); ③ Conduct a fine-grained ablation study for AQE (§7.4); ④ Analyze candidate paths during path exploration (§7.5); ⑤ Assess sensitivity of the sampling in AQE to data distribution (§7.6).

7.1 Experimental Setup

Datasets. Table **??** briefly summarizes 8 datasets we used, which were collected from BIRD [25], SemBench [23], and *Kaggle*. Each dataset represents distinct analytical applications. Specifically, *Appstore*, *Superstore*, and *Food_Inspection* focus on entity-level semantic extraction, while others like *Movies* and *Authors* emphasize multi-table reasoning and text-intensive query evaluation. This diversity

Dataset	Table (#Rows)	Queries
Social_Media [25]	twitter (99.9K), user (99.3K)	Q1, Q16–Q18
Superstore [25]	product (5.3K)	Q2
Chicago_Crime [25]	IUCR (401)	Q3
Food_Inspection [25]	violations (36.1K)	Q4, Q5
CoinMarketCap [6, 10]	coins (8.9K), historical (4.44M)	Q6, Q16, Q17
Appstore [21]	playstore (10.8K), user_reviews (64.3K)	Q7–Q9, Q11, Q12
Movies [23, 38, 47]	movies (143.3K), reviews (1.44M)	Q10, Q13–Q15 Q21–Q26
Authors [25]	Paper (2.25M), Journal (15K), Conference (4.5K), Author (247K)	Q19, Q20

Table 3: Datasets used in our evaluation.

Table 4: Profile of query set and quality metrics

Query ID	Involved Operators	Task	Quality Metrics
Q1–Q5	*SemFilter*/*SemProj*	classification	Acc., Pre., Rec., F1
Q6–Q10	Consecutive *SemFilter*	classification	Acc., Pre., Rec., F1
Q11–Q13	*SemProj* & *SemFilter*	classification	Acc., Pre., Rec., F1
Q14	Consecutive *SemProj*	classification	Acc., Pre., Rec., F1
Q15	Consecutive *SemProj*	classification	micro-/macro- Jaccard/F1, Acc.
Q16	*SemFilter* & *SemProj*	extraction	macro- Pre.,Rec.,F1, Jaccard
Q17–Q18	*SemFilter* & *SemProj*	classification	macro-/micro-/weighted- F1
Q19–Q20	*SemFilter* & *SemAgg*	summarization	word similarity, overlap
Q21–Q22	*SemJoin*	-	Acc., Pre., Rec., F1,
Q23–Q26	*SemOrderBy*	ranking	NDCG@10, NDCG@20

ensures robust validation of Sema's optimization strategies across both structured and unstructured data. A complete profile of these datasets can be found in our technical report [35].

Benchmark Queries. Based on the 8 datasets, we design 26 semantic queries, forming a comprehensive testbed for semantic query processing as shown in Table 4. The queries fall into these categories: ❶ Single-operator queries (Q1-Q5): contain a single *SemFilter* or *SemProj*. ❷ Consecutive *SemFilter* queries (Q6-Q10): contain 2-3 consecutive *SemFilter* operators, used to evaluate the AQE framework and execution optimizations. ❸ Queries with consecutive *SemFilter* and *SemProj* operators (Q11-Q20): include queries with *SemProj* and *SemFilter* (Q11-Q13), queries with two *SemProj* operators (Q14-Q15) and queries with *SemFilter* and *SemProj* (Q16-Q18). ❹ Queries involving complex semantic operators: include queries with *SemFilter* and *SemAgg* (Q19-Q20), queries with a *SemJoin* operator (Q21-Q22) and queries with a *SemOrderBy* operator (Q23-Q26). The complete 26 queries in SemaSQL are presented in the technical report [35]. Ground-truth results for Q1-Q5 are collected from the original datasets, while those for Q6-Q26 are obtained from the inference results of a reference LLM (Gemma-3 27B by default).

Evaluation Metrics. For quality, we employ a range of evaluation metrics regarding the specific objectives of each query, which are listed in Table 4. For time and cost efficiency, we use query latency, # LLM invocations, # input/output tokens, and cost (¢) of API calls as evaluation metrics.

Baselines. We compare Sema against three representative systems: Lotus [34] built on Pandas, with optimized algorithms for *SemFilter* and *SemJoin*, Palimpzest [29] with a cascaded optimizer and FlockMTL [14], which implements semantic operators as UDFs within DuckDB. The capabilities of these baseline systems and Sema regarding vectorized execution, pipeline parallelism and asynchronous execution are shown as below:

Capability	Lotus	FlockMTL	Palimpzest	Sema
Vectorized Execution	✓	✗	✗	✓
Pipeline Parallelism	✗	✓	✓	✓
Asynchronous Execution	✓	✗	✓	✓

Implementation Details. Sema is built on DuckDB v1.2.2, supporting both local and remote LLM invocations. We use vLLM [22] as the local LLM inference engine and *OpenRouter* as remote LLM APIs, respectively. We deploy Google's Gemma-3 model [42] with 4B, 12B, and 27B, with a temperature of 0 for inference reproducibility. LLMs with a temperature of 0 will generate the same query results regardless of the reordering of *SemFilter*. For vLLM, all models are configured with tensor_parallel_size =4, bfloat16 precision, and a GPU memory utilization limit of 0.9. Local experiments are conducted on a Linux server with four NVIDIA RTX 5090 GPUs and an AMD EPYC 9554 CPU (64 cores, 503 GB RAM). Pricing for remote API calls follows DeepInfra's rate on *OpenRouter*: 0.04$/million input tokens and 0.13$/million output tokens for Gemma-3 12B, 2$/million input tokens and 8$/million output tokens for GPT-4.1.

7.2 Overall Evaluation of Quality & Efficiency

We compare the output quality and query latency of Sema against three baseline systems using 26 semantic queries. To eliminate the influence of network delay and fluctuation, all systems invoke local LLMs through vLLM. For Q1–Q5, which involve only a single operator and are relatively simple, we use the Gemma-3 4B model; for the more complex queries Q6–Q26, which consist of multiple operators, we adopt the Gemma-3 12B model. For Palimpzest, we use the *MinTime* optimizer policy to execute all queries.

Overall Efficiency Comparison. As shown in Figure 4, Sema consistently outperforms all three baseline systems in query latency across all 26 queries. Particularly, the performance advantage of Sema is remarkable for queries with larger data volumes (e.g., Q6, Q7, Q16, Q19, and Q24), where Sema achieves up to a 3–6× speedup. This improvement stems from Sema's combination of vectorized execution, pipeline parallelism, and asynchronous execution mechanisms, which collectively maximize hardware concurrency and enable effective interleaving with vLLM inference. In contrast, the three baseline systems exhibit notable limitations due to incomplete execution optimizations. Lotus lacks pipeline parallelism, preventing effective overlap between consecutive operators. FlockMTL suffers from more fundamental inefficiencies: it neither leverages vectorized execution nor supports asynchronous execution, leading to inefficient resource usage and slower execution. Palimpzest does not exploit vectorized execution, which limits its ability to benefit from high-throughput execution and consequently constrains overall performance. The advantage of Sema over Lotus on Q13, Q14, and Q17 is slight because the data volume to be processed by semantic operators is small, which cannot fully exploit the system-level optimization opportunities. We note that FlockMTL fails to support queries Q1, Q5, Q19, and Q20 due to the presence of columns containing multiple data types. FlockMTL also cannot support *SemOrderBy* (Q23–Q26), and Palimpzest cannot support both *SemAgg* (Q19–Q20) and *SemOrderBy* (Q23–Q26).

Overall Quality Comparison. Table 5 and Figure 5 show that Sema, Lotus, and Palimpzest achieve comparable accuracy across

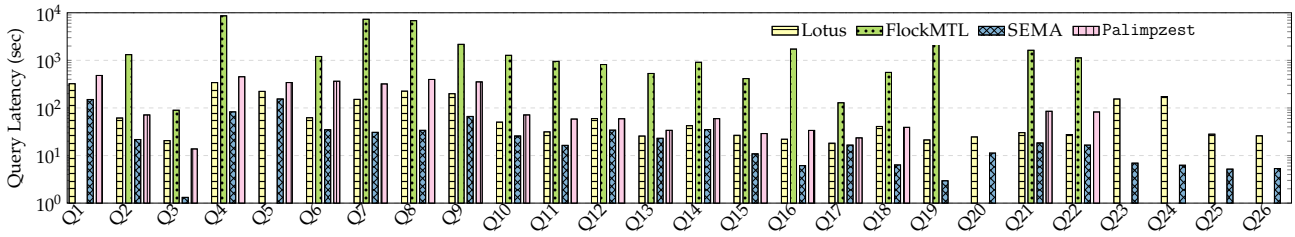

Figure 4: Overall query latency (seconds)

Table 5: Query accuracy for Q1–Q14 and Q21–Q26 (The best scores are highlighted and the second-best are underlined.)

System	Q1		Q2		Q3		Q4		Q5		Q6		Q7		Q8		Q9		Q10	
	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1
Lotus	0.9940	0.9970	0.7367	0.7450	0.7157	0.8343	0.4194	0.4534	0.3040	0.4662	0.6977	0.4078	0.8489	0.7180	0.8542	0.7670	0.9959	0.1875	0.9456	0.5356
FlockMTL	Not Supported		0.7973	0.7994	0.2294	0.3732	0.4897	0.4543	Not Supported		0.7195	0.5017	0.7504	0.3557	0.7445	0.1083	0.9962	0.1429	0.9251	0.2949
Palimpzest	0.9461	0.9723	0.3832	0.4022	0.9426	0.9705	0.2573	0.2326	0.5799	0.7341	0.7011	0.4201	0.8556	0.7480	0.8644	0.7272	0.9956	0.2632	0.9407	0.7166
Sema	0.9934	0.9967	0.7369	0.7407	0.7124	0.8320	0.4428	0.4878	0.1265	0.2245	0.7030	0.4243	0.8148	0.6606	0.8426	0.7160	0.9955	0.2162	0.9543	0.6410

System	Q11		Q12		Q13		Q14		Q21		Q22		Q23 (NDCG)		Q24 (NDCG)		Q25 (NDCG)		Q26 (NDCG)	
	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	@10	@20	@10	@20	@10	@20	@10	@20
Lotus	0.8697	0.5648	0.7358	0.5035	0.7181	0.7666	0.7168	0.7115	0.2683	0.4230	0.1771	0.3009	0.8461	0.8627	0.8315	0.8476	0.6214	0.6473	0.5712	0.6095
FlockMTL	0.1563	0.1360	0.6613	0.3664	0.7440	0.8115	0.7059	0.7249	0.2743	0.4305	0.0229	0.0447	Not Supported		Not Supported		Not Supported		Not Supported	
Palimpzest	0.9381	0.8562	0.6855	0.5532	0.7305	0.7912	0.6517	0.6543	0.2635	0.4312	0.1824	0.3185	Not Supported		Not Supported		Not Supported		Not Supported	
Sema	0.9651	0.9133	0.7273	0.5292	0.7315	0.8883	0.7570	0.7563	0.2864	0.4453	0.2017	0.3357	0.8124	0.8368	0.7987	0.8219	0.6483	0.6926	0.6031	0.6568

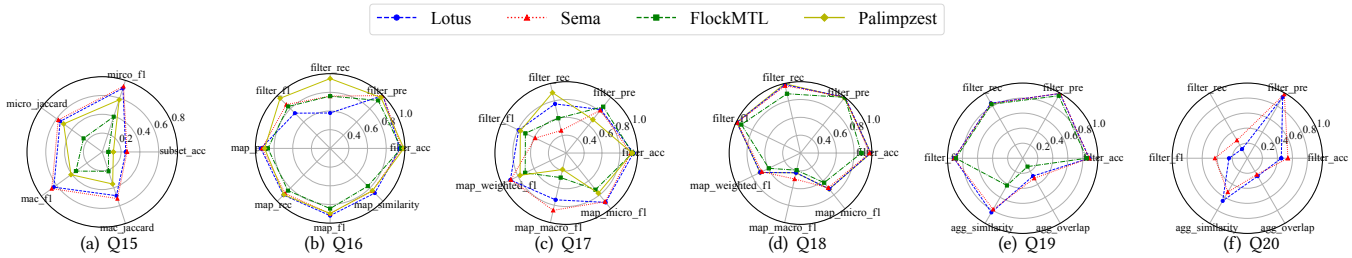

Figure 5: Query accuracy of Lotus, Sema, Palimpzest and FlockMTL for Q15-Q20

most evaluation metrics, with each system exhibiting varying strengths on different queries. This strength inconsistency is largely attributed to differences in prompt formatting across systems; moreover, setting the LLM inference temperature to 0 amplifies sensitivity to even minor prompt variations. Although the system prompt of Sema is carefully aligned with that of Lotus for a fair comparison, slight discrepancies remain due to differences in system implementation. FlockMTL and Palimpzest each employ their own prompt formatting tailored to their architectures. Overall, Sema maintains balanced and competitive accuracy with Lotus and Palimpzest on both simple and complex queries. In contrast, FlockMTL performs considerably worse across nearly all metrics, reflecting the limitations in its prompt construction strategy and model invocation mechanism. For queries with *SemOrderBy* (Q23– Q26), Sema employs a merge-sort-based algorithm, which outperforms and is more robust than the Quicksort-based algorithm in Lotus.

7.3 Expression Optimization Evaluation

We conduct effectiveness studies of two NL expression optimizations, expression compression and predicate deduction, on Q3, Q6, Q13, and Q14; the results are summarized in Table 6. To validate expression compression, we intentionally introduce semantically redundant content into the NL expressions of these queries. To validate predicate deduction, we rewrite certain SQL predicates as NL expressions to assess whether the optimizer can effectively achieve deduction. Query variants can be found in [35].

Expression compression (EC) targets token efficiency by reducing redundancy in NL expressions. EC rewrites verbose expressions into more compact yet semantically equivalent forms, enabling the LLM to operate on shorter prompts. A representative example is Q3, where input tokens decrease from 73.24K (w/o EC) to 72.20K and output tokens decrease from 1.07K to 1.00K after applying EC, while Precision, Recall, and F1 remain nearly unchanged. Similar patterns are observed in Q6 and Q13, confirming that EC effectively improves token efficiency without introducing semantic variation. Predicate deduction (PD) focuses on reducing the number of LLM

Table 6: Impact of Expression Optimization (Compare the performance of the original query with variants of w/ and w/o Predicate Deduction (PD). Here, '+' adds the invocation costs in red of the auxiliary LLM.)

Q	Opt.	Pre.	Rec.	F1	Lat. (s)	IT (K)	OT (K)	LLM Calls
Q3	original	1.0000	0.9950	0.9975	101.82	73.24	1.07	401
	w/o EC	1.0000	0.9880	0.9944	95.43	91.24	1.08	401
	w/ EC	1.0000	0.9890	0.9945	102.05	72.20+1.69	1.00+0.09	401+1
	w/o PD	1.0000	0.9950	0.9975	410.66	191.24	6.08	856
	w/ PD	1.0000	0.9950	0.9975	135.11	73.24+1.07	1.07+0.051	492+2
Q6	original	1.0000	0.9072	0.9513	437.42	7.76	0.16	59
	w/o EC	0.9868	0.9418	0.9637	256.42	9.16	0.15	59
	w/ EC	0.9818	0.9795	0.9803	239.32	8.15+1.46	0.14+0.03	59+1
	w/o PD	1.0000	0.9072	0.9513	443.66	7.76	0.16	59
	w/ PD	1.0000	0.9072	0.9513	447.67	7.76 + 1.13	0.16 + 0.03	59+2
Q13	original	0.8070	0.5567	0.6588	323.02	31.42	3.81	318
	w/o EC	0.8112	0.4610	0.5878	342.25	38.73	3.98	318
	w/ EC	0.7647	0.6569	0.7067	357.97	34.84+3.38	4.28+0.07	318+1
	w/o PD	0.8057	0.5293	0.6388	637.30	57.10	4.62	601
	w/ PD	0.8182	0.5505	0.6582	346.65	31.64+1.25	3.90+0.03	318+2
Q14	original	0.7804	0.6279	0.6636	567.08	57.173	8.71	548
	w/o EC	0.5314	0.2744	0.3031	554.89	74.53	8.04	548
	w/ EC	0.4603	0.2506	0.2814	556.16	62.47+3.42	9.54+0.08	548+1
	w/o PD	0.6134	0.616	0.6147	1055.36	97.15	7.65	1023
	w/ PD	0.3787	0.6338	0.4741	548.93	57.78+0.99	8.80+0.02	552+2

calls during query execution. PD derives executable SQL predicates directly from the NL expressions and injects them into the plan, thereby filtering out non-qualifying tuples before LLM invocation. The benefit is particularly evident in Q13, where PD substantially reduces the number of LLM calls and associated overhead while maintaining comparable result quality.

7.4 Ablation Study of AQE

To evaluate the effectiveness of Sema's AQE framework, we conduct experiments in both local and remote scenarios on queries Q6–Q10, each query involving multiple *SemFilter* predicates. We uniformly sample a fraction of $\delta_1 = \frac{1}{32}$ and $\delta_2 = \frac{3}{32}$ data for expression exploration and path exploration, respectively; the remaining $\frac{28}{32}$ of data is processed by the selected path. We use Gemma-3 12B for LLM inference and set the AMI threshold for operator fusion to 0.4 and the accuracy threshold τ_{acc} to 0.80. Figure 6 provides a fine-grained breakdown of Sema's AQE on Q6–Q10 under both local and remote deployments. Each point represents one execution configuration, with the x-axis denoting latency, the y-axis denoting token cost, and the color (from green to red) indicating relative F1 degradation compared with the reference path. Overall, the results show that batching and fusion are useful but not universally safe optimizations: they may reduce token cost or latency in some settings, but can also introduce non-trivial quality degradation or even increase latency. In contrast, AQE consistently selects paths that remain close to the reference quality while adapting to the cost–latency characteristics of the deployment environment.

Local deployment. In the local setting, applying batching or fusion alone often reduces token cost, but does not consistently improve latency. Batching reduces the number of LLM invocations and hence lowers token overhead, yet it can increase end-to-end latency due to larger prompts, longer decoding sequences, and scheduling inflexibility. Fusion exhibits a similar pattern: although it amortizes system prompt, its latency benefit is dataset- and expression-dependent. Among Q6–Q10, standalone fusion yields a clear latency reduction only in Q6. Critically, these optimizations can substantially degrade quality. The combination of fusion and batching configuration suffers more than 20% relative F1 degradation in four out of five local queries, i.e., Q6–Q9, as indicated by the non-green points. This suggests that aggressively applying batching and fusion as fixed optimizations cannot guarantee semantic reliability. AQE mitigates this issue by evaluating each candidate path before committing to full execution. Across local experiments, AQE-based configurations preserve the F1 relative to the reference path to the largest extent, compared with standalone batching or fusion. The local results highlight a key distinction: the necessity of AQE lies not only in improving latency or cost, but also in preventing quality-unsafe optimizations that violate the accuracy constraint in the multi-objective optimization.

Remote invocations. The remote setting exhibits a different cost-latency tradeoff. Each LLM request incurs non-negligible network round-trip overhead and server-side request-queueing delay. Consequently, batching and fusion become significantly more effective, since they reduce remote API calls, thereby amortizing communication overhead and end-to-end latency while simultaneously lowering token cost. As shown in Figure 6(f)-6(j), batching or fusion configurations reside in the lower-left region compared with the reference path. However, quality remains a concern: in Q6 and Q7, standalone batching/fusion configurations still show noticeable F1 degradation, where the path cannot meet the tolerance for accuracy degradation in the multi-objective optimization. AQE again provides more robust paths, preserving the benefits of latency and cost while keeping quality degradation substantially smaller.

Latency Breakdown. We report the total query latency with a breakdown of planning time and the three-phase latencies in AQE in Table 7. Generally, the planning time for query parsing and logical planning is consistently small across all queries and plan variants. The end-to-end latency is dominated by the final path exploitation phase (t_3), while the two exploration phases (t_1 and t_2) incur only modest overhead. AQE's planning/exploration overhead remains lightweight relative to exploitation, confirming that the framework spends most of its time in exploitation rather than exploration.

7.5 Candidate Path Analysis

Figure 7 illustrates the candidate paths of AQE Phase 2 for Q6–Q8 under local and remote LLM calls. Each point represents a profiled execution path in the latency–cost space. Colors denote different physical optimizations, and markers indicate the AQE decision status. Values near the markers are the calibrated accuracy consistency with p_{ref}, and those below $\tau_{\text{acc}} = 0.8$ are pruned paths marked by crosses. The figure thus makes the AQE decision process explicit: pruning quality-violating paths, then constructing the Pareto frontier over candidates, and finally selecting a path according to the user's objective.

The local setting (Figure 7(a)- 7(c)) shows a clear cost–latency tradeoff. Batching and fusion reduce LLM invocations, but they also enlarge each request prompt. Their benefits are therefore query-dependent: fusion is preferable to batching in Q6, whereas batching is cheaper but slower than fusion in Q7 and Q8. Fusion-plus-batching often yields the lowest cost, but may introduce high latency or be pruned due to consistency loss. Consequently, the local

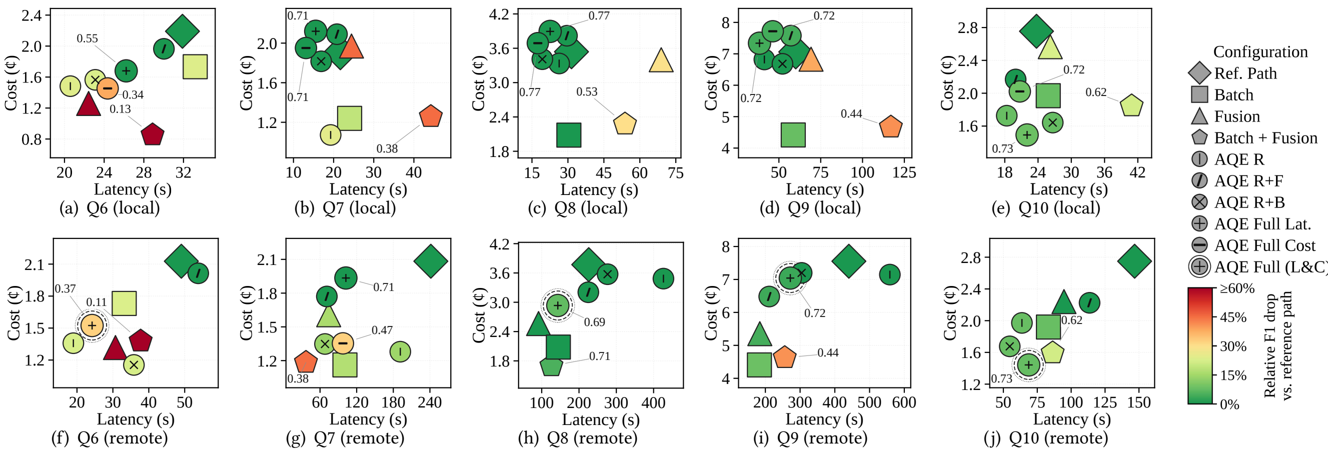

Figure 6: Breakdown evaluation of SEMA regarding query latency, token cost, and F1 under local and remote LLM calls (The color of markers indicates the F1 deviation to the reference path, and the values denote the final F1.)

Table 7: Planning time (in milliseconds), and breakdown time for phases 1–3 of AQE (t1, t2, t3 in seconds)

Metric	**Q6**				**Q7**				**Q8**				**Q9**				**Q10**			
-	p_{ref}	p_{base}	$p^{\star}_{\text{lat}}$	$p^{\star}_{\text{cost}}$	p_{ref}	p_{base}	$p^{\star}_{\text{lat}}$	$p^{\star}_{\text{cost}}$	p_{ref}	p_{base}	$p^{\star}_{\text{lat}}$	$p^{\star}_{\text{cost}}$	p_{ref}	p_{base}	$p^{\star}_{\text{lat}}$	$p^{\star}_{\text{cost}}$	p_{ref}	p_{base}	$p^{\star}_{\text{lat}}$	$p^{\star}_{\text{cost}}$
plan (ms)	0.4	0.4	0.3	0.4	0.5	0.5	0.5	0.5	0.4	0.6	0.6	0.9	0.5	0.5	0.6	0.5	0.5	0.4	0.4	0.4
t_1 **(s)**	0.00	2.48	2.46	3.52	0.00	1.03	1.05	1.03	0.00	1.45	1.44	1.46	0.00	1.58	1.56	1.55	0.00	1.35	1.37	1.42
t_2 **(s)**	0.00	10.00	10.07	10.53	0.00	1.59	3.20	3.19	0.00	2.25	4.55	4.39	0.00	2.69	4.96	5.02	0.00	1.86	2.02	1.89
t_3 **(s)**	94.18	54.42	51.98	50.24	18.28	12.44	15.10	18.55	25.03	19.48	20.37	20.32	49.34	41.27	46.23	45.96	23.72	18.35	17.05	17.51
total (s)	94.18	66.92	64.51	64.29	18.28	15.06	19.36	22.77	25.03	23.19	26.36	26.17	49.34	45.54	52.74	52.52	23.72	21.56	20.44	20.82

latency-first policy typically selects the reference or lightly optimized path, while the cost-first policy selects fusion or fusion-plus-batching only when the quality constraint is satisfied. The remote setting (Figure 7(d) - 7(f)) follows a different pattern. AQE more frequently shifts from the reference path to batching or fusion-plus-batching paths. Overall result quality is comparable between local and remote deployments, while optimized remote paths sometimes show slightly higher consistency, possibly due to provider-side inference infrastructure optimizations. These results confirm that batching and fusion are not uniformly beneficial; their effectiveness depends on the query, dataset characteristics, and deployment environment. AQE is thus necessary to make quality-aware optimal path selection.

Case Study of AQE. We further illustrate AQE dynamics using Q6 with local LLM calls in Figure 7(a). Q6 retrieves cryptocurrencies with large supply (σ_1), active circulation (σ_2), and rising prices (σ_3). In the expression exploration phase, SEMA evaluates the three predicates on a subsample and estimates $s(\sigma_1) = 0.62$, $s(\sigma_2) = 0.70$, and $s(\sigma_3) = 0.18$. AMI analysis gives $\text{AMI}(\sigma_1, \sigma_2) = 0.445$, $\text{AMI}(\sigma_1, \sigma_3) = 0.166$, and $\text{AMI}(\sigma_2, \sigma_3) = 0.056$, indicating that σ_1 and σ_2 are more correlated and promising for fusion, while σ_3 is selective and largely independent. Guided by these statistics, SEMA constructs four candidate paths: $p_1 = [\sigma_3, \sigma_1, \sigma_2]$, its fused variant $p_2 = [\sigma_2 \oplus \sigma_3, \sigma_1]$, and their batched variants $\widehat{p}_1$ and $\widehat{p}_2$. In the path exploration phase, SEMA profiles each candidate path on a subsample to measure the accuracy deviation to the reference path, latency, and token cost. As shown in Figure 7(a), all candidates satisfy the

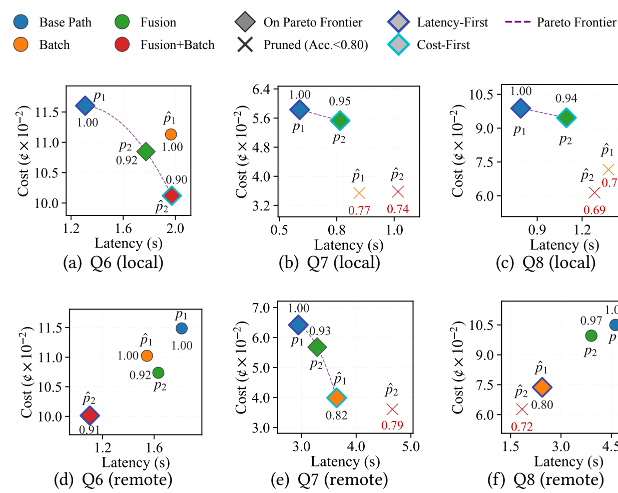

Figure 7: Candidate paths in AQE

accuracy consistency threshold. Among them, p_1, p_2, and $\widehat{p}_2$ form the Pareto frontier: p_1 has the lowest latency, $\widehat{p}_2$ has the lowest token cost, and p_2 provides balanced tradeoff between cost and latency. In contrast, $\widehat{p}_1$ is dominated by p_2 in both latency and cost. Therefore, different user-specified policies choose different paths for the subsequent path exploitation phase: latency-first selects p_1, whereas cost-first selects $\widehat{p}_2$.

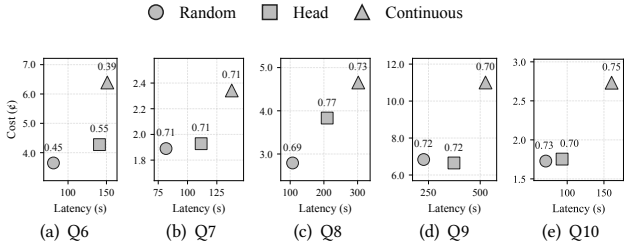

Figure 8: Comparison of different sampling strategies

7.6 Sensitivity of AQE to Data Distribution

We compare three sampling strategies for AQE over clustered data, where the underlying data is clustered by the ground-truth labels of the query's *SemFilter* operators. Here, random sampling adopted by Sema uniformly draws samples from the full input relation before exploitation, and head sampling profiles candidate paths on the prefix of input data. The continuous sampling strategy is adapted from POLAR's *AdaptWindowSize* strategy [16]. Specifically, AQE periodically re-enters the exploration phase, profiles candidate paths on newly observed tuples, and updates the selected path when the profiled tradeoff changes. Figure 8 visualizes the results of token cost, query latency, and accuracy consistency with p_{ref} of the three sampling strategies in the scenario of remote LLM calls. Continuous sampling incurs much higher latency and LLM cost because repeated and sequential path profiling disrupts pipeline-parallel execution. Head sampling is cheap but sensitive to whether the prefix is representative, while random sampling better captures the global distribution and achieves comparable quality in most cases with lower overhead; therefore, Sema adopts randomized initial exploration as the default AQE strategy.

8 RELATED WORK

General ML-based Analytical System. The widespread adoption of ML has driven the development of general ML-based analytical systems. Video database management systems (VDBMS) [4, 18–20, 37, 45] are specifically designed for video data analysis, leveraging DL models as UDFs in DBMS for tasks such as image classification and object detection. To efficiently process video analytical queries, these systems incorporate a variety of optimization techniques, including end-to-end optimization of the preprocessing and query pipeline [20], declarative query hints [37], fine-grained model selection [8], and fast evaluation algorithms for specific operators [4, 18]. Beyond video analysis, ML-powered probabilistic predicates [13, 31, 46] have been applied in a broader range of applications. These studies construct lightweight proxy models and exploit the correlations between different predicates. In addition, some systems also adopt adaptive query processing by dynamically adjusting query plans and resource allocation, or choosing cheaper proxy models, following runtime statistics.

LLM-based Semantic Query Processing. LLMs have introduced a new paradigm in semantic query processing for both structured data [14, 30] and unstructured data [2, 40, 43]. Systems such as DocETL [40], ZenDB [27], DocWanger [41], Aryn [2] and Unify [43] leverage LLMs for analytics over unstructured text by semantic queries, supporting a broad spectrum of query interfaces including extended SQL dialects [27], declarative operations [40], and natural language [2, 41, 43]. In these systems, LLMs also serve as optimization agents during query evaluation, facilitating tasks such as query planning [2], query rewriting, and operation decomposition [41]. UQE [11] extends SQL by allowing natural language join and selection predicates, which intrinsically support semantic join and semantic filter by LLMs. As an LLM-powered analytical system, Palimpzest [29] provides logical and physical optimizations to reduce the query time and monetary cost of LLM invocations. Building on Palimpzest, Abacus [39] introduces a Pareto optimization algorithm and cost estimation for query planning. Lotus [34] formulates LLM-based semantic operations over tabular data and proposes optimized algorithms for costly semantic operations. Akillioglu et al. [1] evaluate semantic query performance in PostgreSQL with pgAI and DuckDB with FlockMTL [14], highlighting the ongoing challenges in enforcing structured outputs and query planning.

Adaptive Query Execution (AQE). Over the past two decades, AQE has progressed from fine-grained tuple routing to robust, domain-specific execution frameworks. The notion of continuous adaptivity was pioneered by Eddies [3], which replaced static operator trees with real-time tuple routing. This idea was later refined by CBR [5], which demonstrated that routing based on data content outperforms simple average statistics. However, to mitigate the high overhead of tuple-level routing, subsequent research has shifted toward more practical, stage-based adaptation. POLAR [16] mitigates cardinality estimation errors by adaptively switching join orders through plans of least resistance, enabling robust execution. Recently, AQE has expanded to support the growing prevalence of ML operators in DBMSs. Aero [17] applies AQE to optimize ML-centric queries, dynamically adjusting the invocation of expensive models based on runtime data characteristics.

9 CONCLUSION

In this work, we presented Sema, a high-performance system that integrates semantic operators for LLM-based semantic query processing in database systems. To optimize such semantic queries, Sema introduces natural language expression optimizations for semantic operators at the optimizer stage and incorporates runtime optimization strategies at the executor stage. Considering the inherent uncertainty of LLMs, Sema adopts an adaptive query execution framework to achieve a balanced trade-off among quality, cost, and latency during execution. Our system demonstrates that endowing database systems with native semantic reasoning capabilities will open new directions for more adaptive, intelligent, and general-purpose query processing as data management enters the era of LLMs and artificial general intelligence. We believe Sema represents an important step toward this vision.

ACKNOWLEDGMENTS

Kangfei Zhao is supported by National Key Research and Development Plan, No. 2023YFF0725101. Yuanyuan Zhu is supported by National Natural Science Foundation of China No. 62272353.

REFERENCES

[1] Kerem Akillioglu, Anurag Chakraborty, Sairaj Voruganti, and M. Tamer Özsu. 2025. Research Challenges in Relational Database Management Systems for LLM Queries. arXiv:2508.20912 [cs.DB] https://arxiv.org/abs/2508.20912

[2] Eric Anderson, Jonathan Fritz, Austin Lee, Bohou Li, Mark Lindblad, Henry Lindeman, Alex Meyer, Parth Parmar, Tanvi Ranade, Mehul A. Shah, Benjamin Sowell, Dan Tecuci, Vinayak Thapliyal, and Matt Welsh. 2025. The Design of an LLM-powered Unstructured Analytics System. In *Proc. CIDR 2025*. www.cidrdb.org.

[3] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proc. SIGMOD 2000*. ACM, 261–272. https://doi.org/10.1145/342009.335420

[4] Jaeho Bang, Gaurav Tarlok Kakkar, Pramod Chunduri, Subrata Mitra, and Joy Arulraj. 2023. SEIDEN: Revisiting Query Processing in Video Database Systems. *Proc. VLDB Endow.* 16, 9 (2023), 2289–2301.

[5] Pedro Bizarro, Shivnath Babu, David J. DeWitt, and Jennifer Widom. 2005. Content-Based Routing: Different Plans for Different Data. In *Proc. VLDB 2005*. ACM, 757–768.

[6] BizzyVinci. 2022. CoinMarketCap Historical Data. https://www.kaggle.com/datasets/bizzyvinci/coinmarketcap-historical-data. Accessed: 2026-05-23.

[7] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proc. VLDB Endow.* 11, 12 (2018), 1755–1768.

[8] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2022. FiGO: Fine-Grained Query Optimization in Video Analytics. In *Proc. SIGMOD 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 559–572.

[9] Surajit Chaudhuri, Vivek R. Narasayya, and Sunita Sarawagi. 2002. Efficient Evaluation of Queries with Mining Predicates. In *Proc. ICDE 2002*. IEEE Computer Society, 529–540.

[10] CoinMarketCap. 2026. CoinMarketCap Historical Snapshots. https://coinmarketcap.com/historical/. Accessed: 2026-05-23.

[11] Hanjun Dai, Bethany 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 *Proc. NeurIPS 2024*.

[12] Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent Advances in Text-to-SQL: A Survey of What We Have and What We Expect. In *Proc. COLING 2022*. International Committee on Computational Linguistics, 2166–2187.

[13] Dujian Ding, Sihem Amer-Yahia, and Laks V. S. Lakshmanan. 2022. On Efficient Approximate Queries over Machine Learning Models. *Proc. VLDB Endow.* 16, 4 (2022), 918–931.

[14] Anas Dorbani, Sunny Yasser, Jimmy Lin, and Amine Mhedhbi. 2025. Beyond Quacking: Deep Integration of Language Models and RAG into DuckDB. *Proc. VLDB Endow.* 18, 12 (2025), 5415–5418.

[15] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proc. SIGMOD 1993*. ACM Press, 267–276.

[16] David Justen, Daniel Ritter, Campbell Fraser, Andrew Lamb, Nga Tran, Allison Lee, Thomas Bodner, Mhd Yamen Haddad, Steffen Zeuch, Volker Markl, and Matthias Boehm. 2024. POLAR: Adaptive and Non-invasive Join Order Selection via Plans of Least Resistance. *Proc. VLDB Endow.* 17, 6 (2024), 1350–1363.

[17] Gaurav Tarlok Kakkar, Jiashen Cao, Aubhro Sengupta, Joy Arulraj, and Hyesoon Kim. 2025. Aero: Adaptive Query Processing of ML Queries. *Proc. ACM Manag. Data* 3, 3 (2025), 174:1–174:27.

[18] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *Proc. VLDB Endow.* 13, 4 (2019), 533–546.

[19] Daniel Kang, Edward Gan, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2020. Approximate Selection with Guarantees using Proxies. *Proc. VLDB Endow.* 13, 11 (2020), 1990–2003.

[20] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. 2020. Jointly Optimizing Preprocessing and Inference for DNN-based Visual Analytics. *Proc. VLDB Endow.* 14, 2 (2020), 87–100.

[21] Lavanya Kannan. 2019. Google Play Store Apps and Reviews. https://www.kaggle.com/datasets/lava18/google-play-store-apps. Accessed: 2026-05-23.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. SOSP*. ACM, 611–626.

[23] Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, H. V. Jagadish, Kris Kissel, Sebastian Schelter, Andreas Kipf, and Immanuel Trummer. 2026. SemBench: A Benchmark for Semantic Query Processing Engines. *Proc. VLDB Endow.* 19, 8 (2026), 1754–1767. https://www.vldb.org/pvldb/vol19/p1754-trummer.pdf

[24] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 743–754.

[25] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *Proc. NeurIPS 2023*.

[26] Jianzhe Lin, Maurice Diesendruck, Liang Du, and Robin Abraham. 2024. BatchPrompt: Accomplish more with less. In *Proc. ICLR 2024*. OpenReview.net.

[27] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeighami, Aditya G. Parameswaran, and Eugene Wu. 2024. Towards Accurate and Efficient Document Analytics with Large Language Models. *CoRR* abs/2405.04674 (2024).

[28] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. 1990. Practical Selectivity Estimation through Adaptive Sampling. In *Proc. SIGMOD*. ACM Press, 1–11. doi:10.1145/93597.93611

[29] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*. 2.

[30] Shu Liu, Asim Biswal, Amog Kamsetty, Audrey Cheng, Luis Gaspar Schroeder, Liana Patel, Shiyi Cao, Xiangxi Mo, Ion Stoica, Joseph E. Gonzalez, and Matei Zaharia. 2025. Optimizing LLM Queries in Relational Data Analytics Workloads. In *Proc. MLSys 2025*. OpenReview.net/mlsys.org.

[31] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *Proc. SIGMOD 2018*. ACM, 1493–1508.

[32] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[33] Xuan Vinh Nguyen, Julien Epps, and James Bailey. 2010. Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance. *J. Mach. Learn. Res.* 11 (2010), 2837–2854.

[34] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4171–4184.

[35] Kangkang Qi, Dongyang Xie, Wenbo Li, Hao Zhang, Yuanyuan Zhu, Jeffrey Xu Yu, and Kangfei Zhao. 2026. Sema: A High-performance System for LLM-based Semantic Query Processing. arXiv:2603.11622 [cs.DB] https://arxiv.org/abs/2603.11622

[36] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proc. SIGMOD 2019*. ACM, 1981–1984.

[37] Francisco Romero, Johann Hauswald, Aditi Partap, Daniel Kang, Matei Zaharia, and Christos Kozyrakis. 2022. Optimizing Video Analytics with Declarative Model Relationships. *Proc. VLDB Endow.* 16, 3 (2022), 447–460.

[38] Rotten Tomatoes. 2026. Rotten Tomatoes. https://www.rottentomatoes.com/. Accessed: 2026-05-23.

[39] Matthew Russo, Chunwei Liu, Sivaprasad Sudhir, Gerardo Vitagliano, Michael J. Cafarella, Tim Kraska, and Samuel Madden. 2026. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. *Proc. VLDB Endow.* 19, 5 (2026), 1060–1073.

[40] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *Proc. VLDB Endow.* 18, 9 (2025), 3035–3048. doi:10.14778/3746405.3746426

[41] Shreya Shankar, Bhavya Chopra, Mawil Hasan, Stephen Lee, Bjoern Hartmann, Joseph M. Hellerstein, Aditya G. Parameswaran, and Eugene Wu. 2025. Steering Semantic Data Processing With DocWrangler. In *Proc. UIST 2025*. ACM, 84:1–84:18.

[42] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, et al. 2025. Gemma 3 Technical Report. arXiv:2503.19786 [cs.CL] https://arxiv.org/abs/2503.19786

[43] Jiayi Wang and Jianhua Feng. 2025. Unify: An Unstructured Data Analytics System. In *Proc. ICDE 2025*. IEEE, 4662–4674.

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Proc. NeurIPS 2022*.

[45] Ziyang Xiao, Dongxiang Zhang, Zepeng Li, Sai Wu, Kian-Lee Tan, and Gang Chen. 2023. DoveDB: A Declarative and Low-Latency Video Database. *Proc. VLDB Endow.* 16, 12 (2023), 3906–3909.

[46] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. 2022. Optimizing Machine Learning Inference Queries with Correlative Proxy Models. *Proc. VLDB Endow.* 15, 10 (2022), 2032–2044.

[47] Andre Zaza. 2024. Clapper: Massive Rotten Tomatoes Movies and Reviews. https://www.kaggle.com/datasets/andrezaza/clapper-massive-rotten-tomatoes-movies-and-reviews. Accessed: 2026-05-23.