Beyond Relational: Semantic-Aware Multi-Modal Analytics with LLM-Native Query Optimization

JUNHAO ZHU, Zhejiang University, China
LU CHEN, Zhejiang University, China
XIANGYU KE, Zhejiang University, China
ZIQUAN FANG, Zhejiang University, China
TIANYI LI, Aalborg University, Denmark
YUNJUN GAO, Zhejiang University, China
CHRISTIAN S. JENSEN, Aalborg University, Denmark

Multi-modal analytical processing has the potential to transform applications in e-commerce, healthcare, entertainment, and beyond. However, real-world adoption remains elusive due to the limited ability of traditional relational query operators to capture query semantics. The emergence of foundation models, particularly the large language models (LLMs), opens up new opportunities to develop flexible, semantic-aware data analytics systems that transcend the relational paradigm.

We present Nirvana, a multi-modal data analytics framework that incorporates programmable semantic operators while leveraging both logical and physical query optimization strategies, tailored for LLM-driven semantic query processing. Nirvana addresses two key challenges. First, it features an agentic logical optimizer that uses natural language-specified transformation rules and random-walk-based search to explore vast spaces of semantically equivalent query plans — far beyond the capabilities of conventional optimizers. Second, it introduces a cost-aware physical optimizer that selects the most effective LLM backend for each operator using a novel improvement-score metric. To further enhance efficiency, Nirvana incorporates computation reuse and evaluation pushdown techniques guided by model capability hypotheses. Experimental evaluations on three real-world benchmarks demonstrate that Nirvana is able to reduce end-to-end runtime by 10%–85% and reduces system processing costs by 76% on average, outperforming state-of-the-art systems at both efficiency and scalability.

CCS Concepts: • **Information systems** → *Online analytical processing engines.*

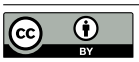
Additional Key Words and Phrases: Semantic data processing, Query optimization, LLMs

ACM Reference Format:
Junhao Zhu, Lu Chen, Xiangyu Ke, Ziquan Fang, Tianyi Li, Yunjun Gao, and Christian S. Jensen. 2026. Beyond Relational: Semantic-Aware Multi-Modal Analytics with LLM-Native Query Optimization. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 14 (February 2026), 25 pages. https://doi.org/10.1145/3786628

1 Introduction

Semantics-aware multi-modal data analytics has long been an attractive vision, with the potential to power important applications across domains such as e-commerce, healthcare, and entertainment. However, despite its promise, reliable deployment in production environments is still lacking.

Authors' Contact Information: Junhao Zhu, Zhejiang University, Hangzhou, China, zhujunhao@zju.edu.cn; Lu Chen, Zhejiang University, Hangzhou, China, luchen@zju.edu.cn; Xiangyu Ke, Zhejiang University, Ningbo, China, xiangyu.ke@zju.edu.cn; Ziquan Fang, Zhejiang University, Ningbo, China, zqfang@zju.edu.cn; Tianyi Li, Aalborg University, Aalborg, Denmark, tianyi@cs.aau.dk; Yunjun Gao, Zhejiang University, Hangzhou, China, gaoyj@zju.edu.cn; Christian S. Jensen, Aalborg University, Aalborg, Denmark, csj@cs.aau.dk.



ACM 2836-6573/2026/2-ART14
https://doi.org/10.1145/3786628

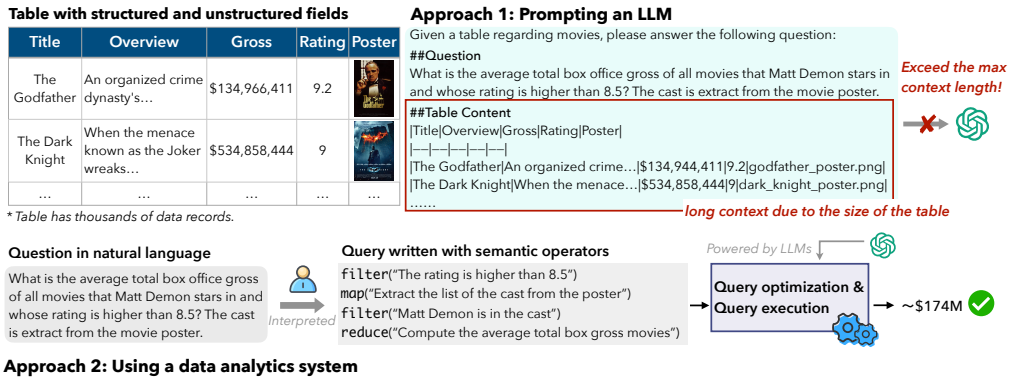


Fig. 1. Example of performing data analytics on a multi-modal movie table by (1) directly prompting an LLM or (2) using a data analytical processing system.

One major impediment is the rigidity of existing analytical processing systems, whose semantics-unaware operators often struggle to support the querying of unstructured and heterogeneous data. Fortunately, the emergence of foundation models offers new opportunities, with Large Language Models (LLMs) demonstrating remarkable general-purpose reasoning capabilities, such as document understanding and question answering. This opens up new possibilities: the realization of flexible, semantic-aware multi-modal data analytics systems that hold the potential to fundamentally transform analytics by leveraging the complementary strengths of foundation models and large-scale data systems. On the one hand, LLMs bring semantic understanding across modalities — including text, images, and audio — into the core of data analytics. On the other hand, system engineering enables these models to be deployed with greater efficiency and transparency.

As a type of compound AI system, multi-modal semantic analytical processing systems can transcend monolithic models by being envisioned to offer the following appealing features:
(1) **High-Quality & Versatile Data Analytics.** These systems are designed to support a wide range of analytics tasks, including descriptive, predictive, and prescriptive analytics, while delivering high-quality results that are robust and production-ready.
(2) **Efficiency & Scalability.** Like relational database systems, semantics-aware multi-modal analytics systems must be capable of processing large-scale, data-intensive workloads efficiently and cost-effectively, scaling to tens of millions of records and beyond.
(3) **Automatic Optimization.** Inspired by traditional relational query optimization, these systems should provide transparent, end-to-end optimization of semantic analytical workflows. Users should be able to specify *what* they want to achieve, without worrying about *how* to achieve it.

Example 1 (Prompt vs. Compound System). *Figure 1 illustrates two approaches to answering the query, "Compute the average total box office gross of all movies that Matt Damon stars in and whose IMDb rating is higher than 8.5," over a multi-modal table, where the cast must be extracted from movie posters. The first approach (upper part) constructs a single prompt by concatenating the query with the full table contents and submits it directly to an LLM. In realistic scenarios — with tables containing hundreds or thousands of rows — this method fails because the prompt's token length exceeds the model's maximum context window, causing truncation or outright rejection. The alternative approach leverages a semantic analytical processing system that exposes a set of programmable operators powered by LLMs. The system decomposes the original query into a sequence of atomic semantic tasks — for example, extracting the cast from posters, filtering out movies with IMDb ratings below 8.5 or without Matt Damon in the cast, etc. Each task is executed on appropriately sized batches of records,*

ensuring that no single LLM invocation exceeds the context limit. Additionally, the system employs various system-level optimizations, such as parallel execution, to make the analytics more efficient and cost-effective.

The recent surge of interest in semantic analytical processing has given rise to two complementary research strands. First, several efforts integrate AI-powered operators into existing SQL engines. For example, ThalamusDB [14] introduces semantic filtering backed by pretrained models into DuckDB. CAESURA [28] augments traditional SQL operators (*e.g.*, joins and aggregations) with semantic ones (*e.g.*, TextQA and VisualQA) for question answering over text and images. While these extensions unlock new functionality, they exhibit low compatibility with the query optimizers in existing systems. Second, a growing body of studies target ground-up, purpose-built semantic analytical systems [2, 17, 21, 24, 29]. These systems expose declarative, natural-language-parameterized transformations — such as the 'filter' and 'map' operators used in Figure 1 — alongside traditional relational operators. Notably, they incorporate system-level optimizations: for example, Palimpzest [17] adapts the Cascades optimization framework to generate semantic query execution plans that strike a Pareto-optimal balance among quality, latency, and cost. Despite this progress, fundamental questions remain unanswered. Classical cost estimation falls short when the selectivity and performance of operators depend on LLM inference behavior. Moreover, *analysis quality* emerges as a first-class optimization objective due to the probabilistic nature and occasional hallucinations of foundation models. *Beyond simply extending SQL or reusing relational optimizers, what new optimization paradigms can unlock the full promise of multi-modal semantic processing?*

We present Nirvana, a novel multi-modal analytical processing framework that offers programmable semantic operator interfaces and introduces new strategies for system-level optimization. Rather than sidelining existing efforts on semantic processing, the goal of Nirvana is to present complementary optimization opportunities for semantic processing that existing systems overlook. In addition to implementing semantic operators that enable versatile data analytics, Nirvana contributes two new optimization designs, driven by the following key questions:
Question 1: What makes semantic query optimization different from traditional query optimization? Unlike traditional relational operators, semantic operators invoke LLM inference to process multi-modal data, which is expensive and dominates the processing cost due to the high computational cost associated with LLM inference. Therefore, the prime object of query optimization is to reduce the number of LLM calls. Borrowing from relational DBMSs, Nirvana adopts transformation-based logical plan optimization (semantic query rewriting), employing traditional rules like predicate pushdown and new rules tailored to semantic analytics, *e.g.*, operator fusion and non-LLM replacement. These transformations require the optimizer to understand query semantics that existing query optimizers cannot handle. LLMs are a silver bullet for semantic query optimization, as they have been shown to be capable of solving complex tasks such as workflow orchestration [12, 25, 32, 37]. Nirvana employs an agentic logical plan optimizer that leverages LLMs to explore semantically equivalent but more efficient plans using a random walk-based search framework. However, we note that, in relational databases, query optimization overhead is negligible compared to query execution (*e.g.*, milliseconds for terabytes of data according to the TPC-H benchmark [27]), while LLM calls can take seconds to complete. Given the high cost of LLMs, ***is it worth using LLMs for query optimization?*** Our answer is yes. Unlike the query optimizations in relational databases that is based on data statics, *semantic query optimization can benefit markedly from the the capabilities of LLM, despite the associated costs.* Our logical plan optimizer can yield larger savings (in both runtime and monetary cost) compared to the optimization overhead. Our empirical studies show that the logical plan optimizer can reduce end-to-end execution time by 40.6% and processing price by 25.0% on average, respectively.

Table 1. Semantic operators currently supported in Nirvana.

Operator	Definition	Usage Example
map	Applies an LLM to perform a transformation on a column, producing a new column as output	df.semantic_map(user_instruction="NL-specified transformation", input_column="colA", output_column="colB")
filter	Uses an LLM to evaluate a natural language predicate on a column	df.semantic_filter(user_instruction="NL-specified predicate", input_column="colA")
reduce	Aggregates values in a column based on an NL-specified reduction function	df.semantic_reduce(user_instruction="NL-specified reducer", input_column="colA")
rank	Ranks values in a column according to an NL-specified ranking function	df.semantic_rank(user_instruction="NL-specified ranker", input_column="colA")

Question 2: Given logical plans for semantic data analytics, do physical plan optimizations exist? Service providers usually offer a variety of LLM options. For instance, OpenAI offers three sizes of its GPT-4.1 series: GPT-4.1, GPT-4.1 mini, and GPT-4.1 nano. These models exhibit heterogeneous performance and costs, where a more powerful LLM is inevitably more expensive. Each semantic operator in a plan can choose a variety of models to use. Among all LLM options, some are overkill for simple analytics requests, while others may underperform on complex ones. Nirvana defines a physical plan as a choice of execution strategy (*i.e.*, models in this work) for operators in a logical plan, and we apply physical plan optimization through model selection. Jo et al. [13] proposed Smart, a model selection method, which leverages a mix of LLMs to provide similar levels of output quality as the most powerful LLM but at a much lower expense. However, Smart only supports selecting LLMs for a single operator in a fixed task. Nirvana supports general query plans, which optimizes physical plans by assigning the most cost-effective model to each operator based on an improvement score, a novel measure of model cost-effectiveness. To improve optimization efficiency, we introduce two techniques, *computation reuse* and *evaluation pushdown*, to eliminate unnecessary computations. The empirical study (Section 5) finds that these optimizations reduce optimization overhead by 4× compared to Smart, while maintaining system effectiveness.

In summary, the key contributions are as follows:

- We discuss differences between semantic data analytics and relational data analytics, motivating an agent-based logical plan optimizer that uses a random walk search framework and an LLM-as-a-judge plan evaluator for logical plan exploration and exploitation.
- We design a physical plan optimizer that selects the most cost-effective LLM backend per operator in a logical plan, minimizing inference costs while preserving result quality.
- The core of the physical plan optimizer is a model selection algorithm based on the novel concept of an improvement score. To reduce optimization overhead, we propose optimization techniques — computation reuse and evaluation pushdown — with a model capability hypothesis.
- Extensive experiments on three real-world multi-modal benchmarks show that Nirvana can reduce end-to-end runtime by 10%—85% and monetary cost by 76%, compared to state-of-the-art systems. Each component plays its unique role in facilitating system's efficiency and scalability.

The remainder of the paper is organized as follows. Section 2 provides an overview of Nirvana. Section 3 presents the logical plan optimizer in Nirvana. Section 4 presents the physical plan optimizer. We report experimental results in Section 5. We discuss related work in Section 6 before concluding and discussing future directions in Section 7. Nirvana is open-sourced at https://github.com/JunHao-Zhu/nirvana.

2 System Overview

We first provide background on the data model and analytical queries supported by Nirvana. Then, we define the notions of logical and physical plans in the context of Nirvana and outline the optimizations involved.

2.1 Background: Data Model & Operators

Nirvana supports a data model consisting of tables with both structured and unstructured fields. Unstructured fields may contain heterogeneous content such as text, documents, images, or audio. Figure 1 presents an example of a movie dataset containing numeric, textual, and image fields. Internally, Nirvana represents unstructured fields as text that store file paths pointing to remote locations (*e.g.*, web URLs or object storage such as S3 [1]) where the unstructured data resides.

To support flexible and versatile semantic data analytics, Nirvana offers a suite of semantic operators. Table 1 summarizes the operators currently implemented in Nirvana: The map operator applies an LLM-driven transformation to a target column, producing a new derived column. The filter operator retains records based on a natural language predicate. The reduce operator applies a many-to-one aggregation across multiple data in a target column using a user-specified reducer expressed in natural language (NL). The rank operator assigns rankings to input values using an NL-defined ranking criterion. While Table 1 lists the currently supported operators, Nirvana is extensible with support for additional operators such as join and group by in this work. Users can construct their own analytical queries tailored to their data and tasks by employing semantic operators in Nirvana. We describe the system as multi-modal because it operates over multi-modal relational data, and semantic operators can interpret queries and generate results involving different modalities. The following is an example query to analyze common characteristics of crime movies with ratings between 8.5 and 9:

```
import nirvana as nv
df = nv.DataFrame(movie_data)
df.map(user_instruction="According to the movie overview, extract the genre of
    each movie.", input_column="Overview", output_column="Genre")
df.filter(user_instruction="The rating is higher than 7.", input_column="Rating")
df.filter(user_instruction="The rating is lower than 8.", input_column="Rating")
df.filter(user_instruction="The movie belongs to crime movies.", input_column="
    Genre")
df.reduce(user_instruction="Summarize the common points of their stories.",
    input_column="Overview")
```

Listing 1. Query example.

2.2 Query Optimization

At present, optimization in Nirvana aims to minimize system cost and latency under an accuracy constraint. This reflects common approximate-query practice where small, bounded deviations from the oracle result are acceptable in exchange for cost/latency improvements. In Nirvana, a user-specified query is compiled into data lineage (a directed acyclic graph), which serves as a logical plan in this paper. A logical plan captures the execution order of semantic operators, along with their predicates or instructions and their scopes of application, as illustrated on the left side of Figure 2, where a logical plan is derived from the query in Listing 1. A corresponding physical plan specifies the actual execution engines (e.g., computing functions or AI models) used to implement each operator in the logical plan. Next, Nirvana offers logical and physical plan optimizations. Inspired by the Spark SQL system [3], Nirvana separates query optimization into logical plan optimization and physical plan optimization.

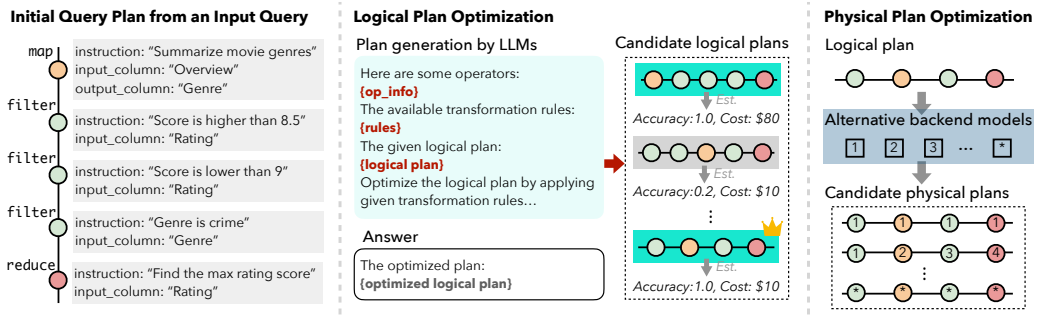


Fig. 2. System overview of Nirvana. Given an initial query plan, the logical optimizer samples and rewrites the logical plan to identify a (near-)optimal plan in the logical plan optimization phase. Then, the physical optimizer assigns the most cost-effective model to each operator in the logical plan.

Algorithm 1: Logical plan optimization

Input: Data samples D_s, an initial plan p_0, an plan rewriter m, the number of iterations N_{max}, error tolerance ϵ

Output: the optimized logical plan p^*

Initialize the set of candidate plans $\mathcal{P} \leftarrow \{p_0\}$
Initialize the best logical plan $p^* \leftarrow p_0$
for $i \leftarrow 1$ *to* N_{max} **do**
  $p_i \leftarrow sample(\mathcal{P})$
  $p^+ \leftarrow rewrite(p_i, m)$
  $p^+.acc, p^+.cost \leftarrow evaluate(p^+, p_0, D_s)$
  if $p^*.acc \geq \epsilon \cap p^+.cost \leq p_i.cost$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{p^*\}$;
$p^* \leftarrow \arg\min_p \{p.cost \mid p \in \mathcal{P}\}$

While logical plan optimization in Nirvana mirrors and extends traditional rule-based approaches, Nirvana devises a key innovation — an agentic optimizer — and introduce several transformation rules specific to semantic processing (*i.e.*, operator re-ordering, operator fusion, and operator replacement). As indicated in Figure 2, the agentic optimizer explores multiple candidate logical plans and estimates their result quality (measured by accuracy) and execution cost. It then selects the plan that minimizes the cost while meeting a quality constraint. The details of the logical plan optimizer are covered in Section 3. As shown to the right part in Figure 2, following the logical plan optimization, in Nirvana, the physical plan optimizer assigns appropriate backend engines (*i.e.*, LLMs) to each operator in the selected logical plan. The goal is to choose the most cost-effective configuration under the quality constraint. Details are presented in Section 4. Note that, the optimizers are modality-agnostic: optimizations apply uniformly across data types.

3 Logical Plan Optimizer

We begin by presenting the logical plan optimizer, which applies transformation rules to the initial (user-specified) plan to produce an optimized logical plan.

3.1 Optimization Workflow

Algorithm 1 outlines the skeleton of the logical plan optimizer, and Figure 3 illustrates the workflow with a running example.

Initialization. Given a user-specified plan as input, the optimizer initializes the set $\mathcal{P}$ of candidate logical plans with an empty set and sets the best logical plan p^* to the input plan p_0 (lines 1–2).

Plan rewrite. In each iteration, a plan is sampled from the candidate set for rewriting (line 4). We adopt a random walk approach [10] to sample a plan from the candidate set. Each plan is assigned a probability such that all candidates have a chance to be sampled. The random walk spans a tree, and each node represents a logical plan, as shown in Figure 3, where child nodes are generated from their parent node by rewriting plans. To avoid getting stuck in a local optimum, each plan is sampled from a distribution of a mixture of a uniform distribution and a cost-based weighted probability distribution. Specifically, a plan p_i is sample with the probability:

$$Pr(p_i) = \lambda \cdot \frac{1}{|\mathcal{P}|} + (1-\lambda) \cdot \frac{\exp(cost_{max} - p_i.cost)}{\sum_{p_j \in \mathcal{P}} \exp(cost_{max} - p_j.cost)}, \tag{1}$$

where λ (set to 0.2) controls the tendency to either continue optimizing the currently best observed plan or to explore new branches. This random walk search procedure interpolates between breadth-first search and depth-first search, thereby balancing plan exploration and plan exploitation.

The selected plan is then rewritten by an LLM rewriter using transformation rules expressed in natural language (line 5). The concept of LLM-driven plan rewriting has been validated and adopted in various domains, including in workflow generation [32, 37] and document analytics [24].

Plan evaluation. Once the sampled plan is rewritten, we check for semantic equivalence and evaluate the plan's cost. If the new plan is semantically equivalent to the input plan and incurs lower cost, it is added to the candidate set (lines 6–8).

Verifying semantic equivalence is quite tricky, especially for semantic queries. There are two representative query verification approaches: (1) *Formal verification*: converts SQL queries into logic formulas and checks the equivalence via algebraic solutions, *e.g.*, SQLSolver [7]. (2) *Execution consistency*: check if the execution result of a query matches that of the corresponding rewritten query, *e.g.*, SQLDriller [31]. Formal equivalence checking is attractive but difficult to apply in semantic analytics because many natural-language-driven operators lack straightforward logical encodings. Although one can use LLMs to perform formal equivalence checking, it creates a circular trust issue that it uses the same LLM to both rewrite and verify. To ensure an *objective* verification, Nirvana performs execution consistency checking that the execution results of rewritten plans are compared with those of the original plan. However, result equivalence can also be vague in semantic analytics, as LLM-executed semantic operators may produce syntactically different but semantically identical outputs for the same inputs. To address this, we use LLM-as-a-judge [33] as an equivalence verifier. We execute both the input plan and the rewritten plan on a data sample. With the outputs from both logical plans, we prompt an LLM to rate the similarity between the two plan outputs on a scale from 0 (completely different) to 10 (exactly same) based on the their semantic consistency. The normalized rating score is used as the plan's accuracy.

For plan cost, the dominant factor, the overall running time and the monetary cost for LLM inference are typically proportional to the number of LLM calls (*i.e.*, the number of data points to process) and the prompt length per data processing. Our cost estimator tracks the number of processed data items and prompt lengths per operator. The total cost is computed as the sum of all operator costs. We assign a selectivity parameter to each operator to monitor changes in the number of data points to be processed throughout the plan. By default, filter has a selectivity of 0.5, reduce has a selectivity of 0, and the other operators have one of 1. We track changes to operators

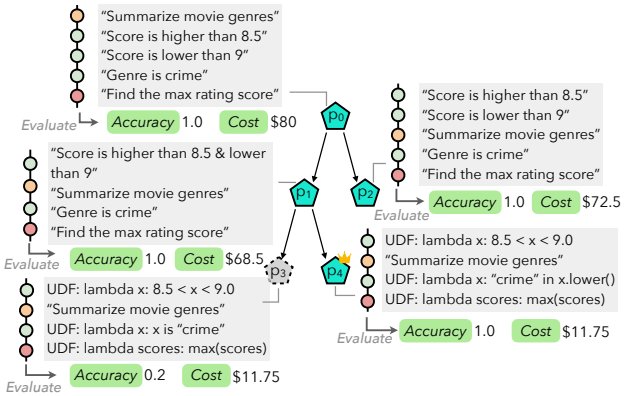


Fig. 3. A running example of logical plan optimization where, each node (a logical plan) is sampled by random walk.

in the plan and adjust their selectivity accordingly. For instance, when multiple operators (*e.g.*, two or three filters) are merged, the new operator's selectivity is adjusted accordingly (*e.g.*, reduced to half or one third, depending on how many filter operators are merged).

Although our system uses constant selectivity assumptions by default for now, operator selectivity can also be estimated using cardinality estimation techniques, similar to those in relational DBMSs. However, semantic data analytics depends not only on data distribution but also on natural language instructions, motivating a new line of research on semantic cardinality estimation. Nirvana can seamlessly integrate such semantic estimators once they become available. For instance, consider two filter operators in a query plan. Under the default optimizer, their order cannot be determined as they have identical selectivities. With a semantic cardinality estimator, however, the optimizer can prioritize execution by pushing down the filter with the higher selectivity, thereby reducing overall query cost. In addition, it also works for determining a join order.

At the end of the optimization, the logical plan optimizer returns the lowest-cost plan that satisfies the accuracy constraint (line 9). In the case of Figure 3, both p_3 and p_4 have low costs, but p_3 is discarded because it fails the semantic equivalence check.

3.2 Transformation Rules

Traditional query optimizers based on frameworks like Volcano [9] and Cascades [8] face several limitations in the context of semantic data analytics:

(1) *Missing optimization opportunities.* Traditional rule-based systems provide deterministic, hard-coded transformations that are hard to extend at query-time and hard to offer flexible, query-specific semantic interpretations and rewrites that are difficult to pre-enumerate.
(2) *Limited extensibility.* Transformation rules in existing query optimizers are usually hard-coded and tightly coupled with system internals [30]. Integrating a new transformation rule into a commercial or open-source database system is a non-trivial undertaking.

Our agentic logical plan optimizer addresses these issues. First, it leverages the semantic understanding of LLMs to enable more aggressive and intelligent plan rewrites. Second, the transformation rules are expressed in natural language, decoupling them from system internals.

The following three transformation rules are applied in our logical optimizer. Additional rules, formulated in natural language, can be added easily.

Algorithm 2: Naive backend model selection

Input: Data samples D_s, an operator o, a model set $\mathcal{M} = \{m_1, m_2, m_3, m^*\}$, an improvement margin ΔI_{min}
Output: the most cost-effective model m_o

Initialize improvement scores $I_{last} \leftarrow 0, I_{curr} \leftarrow 0$
Initialize the most cost-effective model $m_o \leftarrow m_1$
$R_{m_1} \leftarrow \{m_1(x) \mid x \in D_s\}$
$R_{m^*} \leftarrow \{m^*(x) \mid x \in D_s\}$
for m *in* $\mathcal{M}$ **do**
    if $m = m_1$ **then**
        $I_{curr} \leftarrow 0$
        continue
    $R_m \leftarrow \{m(x) \mid x \in D_s\}$
    $I_{curr} \leftarrow$ estimate model cost-effectiveness by computing the improvement score $I_{m_1 \to m}(D_s, m^*)$ on R_{m_1}, R_m, and R_{m^*}
    if $I_{curr} - I_{last} \geq \Delta I_{min}$ **then**
        $m_o \leftarrow m$
        $I_{last} \leftarrow I_{curr}$

- **Filter pushdown.** This transformation, widely used in existing DBMSs, moves a filter operator that does not rely on results of preceding operators to an earlier stage in the plan.
- **Operator fusion.** This transformation merges multiple operators on the same field into one operator. To maintain semantic equivalence, the predicate (or user instruction) will be rewritten for the new operator. For example, in Figure 3, two filter operations with predicates "Score is higher than 8.5" and "Score is lower than 9" can be merged into a single filter with predicate "Score is higher than 8.5 and lower than 9", yielding fewer LLM calls in processing a query.
- **Non-LLM replacement.** This transformation replaces an operator's NL instruction with an equivalent compute function. For example, in Figure 3, "Score is higher than 8.5 and lower than 9" can be interpreted as a comparison function: 8.5 < score < 9.

3.3 Local Rewrite Model

The cost overhead is one of the biggest obstacles when using LLMs for plan optimization, as each LLM call is expensive in both latency and price. Especially when using LLM services provided by cloud vendors, most of the time is wasted on network communication. One means of mitigating this problem is to use a local rewrite model as a substitute.

Training data collection. A dataset to be used for training a local rewrite model consists of pairs of un-optimized and rewritten logical plans. To assemble such pairs, we first use an LLM to convert natural language data analytics questions into analytical queries using the semantic operators in Nirvana, providing the question and operator interface documentation. Then, we use the LLM with transformation rules to rewrite the logical plan compiled from the generated queries in a last step. Collecting the training set, we fine-tune a small LLM (e.g., Qwen-3-8B, LLaMA-3-8B) using LoRA [11] in an offline setting to obtain the local rewriter.

Model inference. In the inference phase, the sampled logical plan is fed to the local rewrite model to generate an optimized version, *i.e.*, replacing the cloud LLM service with the local model in the optimization workflow. As cloud vendors improve LLM services in latency and others features, the local model's inference needs to be carefully implemented to achieve the same effect. For

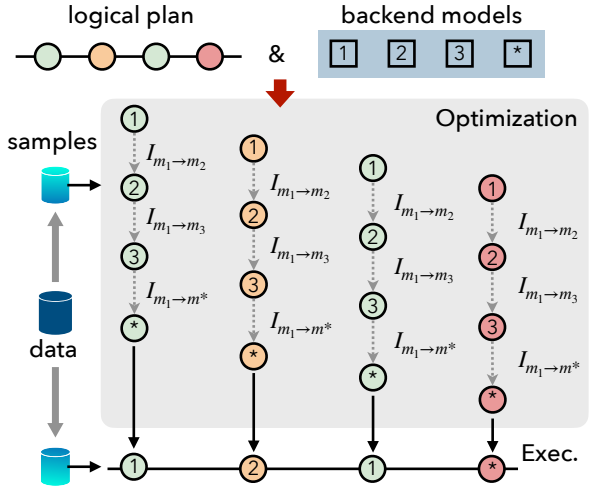


Fig. 4. Overview of physical plan optimization.

instance, to further accelerate generation, LLM serving systems (*e.g.*, vLLM [15], SGLang [34]) are recommended for local deployment.

4 Physical Plan Optimizer

In this work, a physical plan captures the assignment to semantic operators of backend models used as execution engines. Different models with varying scales offer different levels of intelligence, speed, and cost. Selecting the most appropriate model for each operator is essential for cost-effective data analytics, which is the goal of physical plan optimization. For readability, we discuss a case with a search space of four models[1], *i.e.*, $\mathcal{M} = \{m_1, m_2, m_3, m^*\}$. These models have the following properties:

$$\text{Intelligence (from weakest to strongest):} \quad m_1 \prec m_2 \prec m_3 \prec m^*$$
$$\text{Cost (from cheapest to most expensive):} \quad m_1 \succ m_2 \succ m_3 \succ m^*$$

Model m_1 is the weakest but fastest (and cheapest), while m^* is the most powerful but slowest (and most expensive).

4.1 Optimization Workflow

Given a logical plan with N operators and a choice of M models per operator, the search space of feasible physical plans is of size N^M, yielding a search space that is too large to efficiently find an optimal physical plan. Hence, we constrain the search space by making the following assumption:

Hypothesis 1 (Operator Independence). *Suppose that a model m_a is more intelligent and costly than a model m_b. Then, choosing m_a over m_b as the execution engine for an operator in a plan improves the overall result quality (but at higher cost), regardless of the models allocated to other operators in the plan.*

The hypothesis overlooks potential error propagation within the workflow — errors produced by one operator may cascade and amplify through subsequent operators. Fortunately, since inter-op

[1]Why do we choose a four-model setting to discuss? It can cover most use practices, where users may choose among a Tall-sized model (*e.g.*, BERT), a Gronde-sized model (*e.g.*, LLaMA3-8B), a Venti-sized model (*e.g.*, LLaMA3-70B), and a Trenta-sized model (*e.g.*, GPT-4.1). The discussion generalizes to cases with more models.

interactions are handled by the logical optimizer while the physical optimizer considers only intra-op optimization, the impact of hypothesis 1 is minimal. We adopt a greedy solution for model selection. Figure 4 provides an overview of the physical plan optimization process. The system selects the most cost-effective model for each operator from a pool of candidates. In the subsequent execution phase, the chosen model is used to process the remaining data. Algorithm 2 captures the model selection procedure for a single operator, presenting a naive baseline that forms the foundation for our optimizer. Given an operator in the logical plan, we evaluate the cost-effectiveness of each candidate model by computing an improvement score $I_{m_1 \to m}$ over a sampled dataset D_s. This score quantifies the gain in output quality obtained by replacing the baseline model m_1 with a candidate model m as the execution backend. If the score exceeds a user-defined threshold ΔI_{min}, the model is deemed sufficiently beneficial and is assigned to the operator. To estimate model cost-effectiveness, one always needs to run models m_1, m, and m^* on the entire sampled dataset, which can be expensive. To reduce this cost, we introduce several optimization techniques that decrease the amount of data that must be processed.

Next, we define the improvement score and present acceleration strategies based on it.

4.2 Improvement Score Computation

Since the ground-truth result of a query is unknown, we treat the output[2] of the most powerful model m^* as a proxy for the ground truth. We define the improvement score $I_{m_1 \to m}$ as the performance improvement gained from switching from the backend model m_1 to a stronger model m. The definition is as follows.

$$\begin{aligned} I_{m_1 \to m_2}(D, m^*) &\triangleq \mathbb{E}_{x \sim D}[m_2(x) = m^*(x), m_1(x) \neq m_2(x)] \\ I_{m_1 \to m_3}(D, m^*) &\triangleq \mathbb{E}_{x \sim D}[m_3(x) = m^*(x), m_1(x) \neq m_3(x)] \\ I_{m_1 \to m^*}(D, m^*) &\triangleq \mathbb{E}_{x \sim D}[m_1(x) \neq m^*(x)], \end{aligned} \tag{2}$$

where $m(x)$ denotes the response of the model m to an input x. And, $m_1(x) = m_2(x)$ (resp. $m_1(x) \neq m_2(x)$) denotes that the outputs from the models m_1 and m_2 are semantically equal (resp. different). For binary-output operators like filter, which are required to return True or False, the model outputs can be compared directly. For other ad-hoc queries, such as map that returns arbitrary values, the model outputs are compared using semantic similarity via semantic embeddings generated by a pre-trained language model, like Sentence-BERT [22].

As shown in Eq. 2, an improvement score $I_{m_1 \to m}$ behaves as an expected proportion of inputs from a dataset D to which the responses of model m are consistent with the surrogate ground truth $m^*(x)$ while responses of the baseline model m_1 are not.

For simplicity, we omit D and x, since models always process the same data with respect to a given operator.

Next, to compute exact improvement scores, it is necessary to invoke the most expensive model m^* on all inputs. To reduce this potentially large overhead, we make the following transformation that reduces the number of m^* invocations.

$$\begin{aligned} I_{m_1 \to m_2} &\triangleq \mathbb{E}[m_2 = m^*, m_1 \neq m_2] \\ &= Pr(m_2 = m^*, m_1 \neq m_2) \\ &= Pr(m_2 = m^* \mid m_1 \neq m_2) Pr(m_1 \neq m_2) \end{aligned} \tag{3}$$

This transformation allows us to execute expensive evaluations $m_2 = m^*$ on only the subset of records that meet the condition $m_1 \neq m_2$, reducing the number of evaluations involving the

[2]we use terms "model output" and "response" interchangeably in this paper.

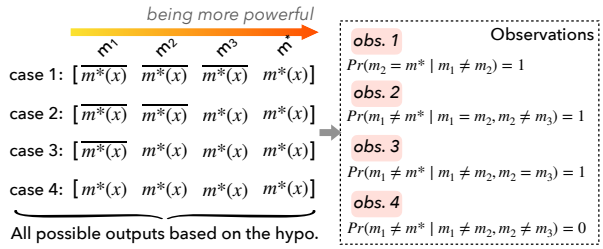


Fig. 5. Possible responses of different models to a query, based on the model capability hypothesis. $m^*(x)$ denotes the response of the most powerful model m^* to the input x, while $\overline{m^*(x)}$ denotes the opposite response to the same input.

Table 2. Comparison of GPT-4.1 and GPT-4.1-nano outputs for the task of extracting amenities from estate descriptions.

#Aligned	#Misaligned	#GPT-4.1-is-right	#GPT-4.1-nano-is-right
424	76	69	7

expensive model, m^*. We refer to this optimization as "*evaluation pushdown*" because its core idea is similar to "predicate pushdown" in DBs.

The improvement score $I_{m_1 \to m_3}$ can be transformed in a similar way, *i.e.*, $I_{m_1 \to m_3} = Pr(m_3 = m^* \mid m_1 \neq m_3) Pr(m_1 \neq m_3)$. However, we can further reduce the number of invocations of the best model m^* by applying the law of total probability:

$$\begin{aligned} I_{m_1 \to m_3} &= Pr(m_3 = m^*, m_1 \neq m_3) \\ &= Pr(m_3 = m^*, m_1 \neq m_3, m_2 = m_3) + Pr(m_3 = m^*, m_1 \neq m_3, m_2 \neq m_3) \\ &= I_{m_1 \to m_2} + Pr(m_3 = m^*, m_2 \neq m_3, m_1 = m_2) \end{aligned} \tag{4}$$

The last term in Eq. 4 can be expanded as follows.

$$\begin{aligned} &Pr(m_3 = m^*, m_2 \neq m_3, m_1 = m_2) = \\ &Pr(m_3 = m^* \mid m_2 \neq m_3, m_1 = m_2) Pr(m_2 \neq m_3 \mid m_1 = m_2) Pr(m_1 = m_2) \end{aligned}$$

From Eq. 4, the number of m^* invocations in the evaluation of $m_3 = m^*$ can be reduced by the conditions $m_2 \neq m_3$ and $m_1 = m_2$, where $m_1 = m_2$ was already evaluated when computing $I_{m_1 \to m_2}$. We also note that the computing $I_{m_1 \to m_3}$ directly reuses the result of $I_{m_1 \to m_2}$. Therefore, we call this technique "*computation reuse*".

These conclusions also apply to the computation of $I_{m_1 \to m^*}$. For brevity, we only show the expansion of $I_{m_1 \to m^*}$ using the law of total probability:

$$\begin{aligned} I_{m_1 \to m^*} &= Pr(m_1 \neq m^*) \\ &= Pr(m_1 \neq m^*, m_1 = m_2, m_2 = m_3) + Pr(m_1 \neq m^*, m_1 = m_2, m_2 \neq m_3) \\ &+ Pr(m_1 \neq m^*, m_1 \neq m_2, m_2 = m_3) + Pr(m_1 \neq m^*, m_1 \neq m_2, m_2 \neq m_3) \end{aligned} \tag{5}$$

Although *evaluation pushdown* and *computation reuse* help reduce unnecessary computations, considerable computation is still required for evaluations involving the slowest and most expensive model m^*. In the following, we introduce techniques to accelerate this computation.

Table 3. Dataset characteristics.

Dataset	#Records	#Attributes	Modalities
Movie	250	22	numerical values, text, images
Estate	1041	4	images, long text
Game	18891	21	dates, numerical values, images, text

4.3 Improvement Score Approximation

To achieve acceleration, we first introduce a model capability hypothesis that enables the elimination of slow and expensive m^* invocations when computing improvement scores.

Hypothesis 2 (Model Capability Hypothesis). *Given the ground-truth result y to a query x, if a model m returns a correct answer y to x,* i.e., *$m(x) = y$, any model m^+ that is more powerful than m also returns the right answer y,* i.e., *$m^+(x) = y$.*

To showcase the generality of this hypothesis, we compare the outputs of GPT-4.1 and GPT-4.1-nano on the same task: extracting amenities from the detailed estate descriptions. We sample 500 records from the Estate dataset for evaluation. As shown in Table 2, among the cases where the outputs of GPT-4.1 and GPT-4.1-nano differ, over 90% of GPT-4.1 responses are correct. Although the hypothesis do not hold universally, it is empirically valid in most practical cases.

Under this hypothesis, we can characterize the possible responses of models in the candidate set as illustrated on the left side of Figure 5, where we treat the output from the best model m^* as the ground truth. In addition, several observations can be made on combinations of model outputs, as shown in Figure 5. For example, $m_2 = m^*$ always holds when $m_1 \neq m_2$ (observation 1 in Figure 5).

Using observation 1, we refine Eq. 3 and obtain the following approximation:

$$\begin{aligned} I_{m_1 \to m_2} &= Pr(m_2 = m^* \mid m_1 \neq m_2) Pr(m_1 \neq m_2) \\ &\approx Pr(m_1 \neq m_2), \end{aligned} \tag{6}$$

where the evaluations of $m_2 = m^*$ are eliminated completely thanks to the observations based on the hypothesis.

The same idea applies to $I_{m_1 \to m_3}$. From Figure 5, we observe that $m_3 = m^*$ always holds when both $m_2 \neq m_3$ and $m_1 = m_2$ are true. This leads to the following approximation of $I_{m_1 \to m_3}$:

$$I_{m_1 \to m_3} \approx I_{m_1 \to m_2} + Pr(m_2 \neq m_3 \mid m_1 = m_2) Pr(m_1 = m_2) \tag{7}$$

Note that since $Pr(m_1 = m_2) = 1 - I_{m_1 \to m_2}$, this computation can reuse the result of Eq. 6 so that $Pr(m_2 \neq m_3 \mid m_1 = m_2)$ is the only term to be determined when computing $I_{m_1 \to m_3}$. Compared to the full evaluation in Eq. 4, the evaluations involving the expensive m^* invocations are entirely eliminated in Eq. 7, and the evaluation of m_3 is constrained to records satisfying $m_1 = m_2$, preserving the benefits of "evaluation pushdown".

Similarly, we can derive an approximation of $I_{m_1 \to m^*}$ as follows. This result also follows observations from the hypothesis and is inferred by algebraic transformations (details are provided in the extended version [36]).

$$\begin{aligned} &I_{m_1 \to m^*} \approx Pr(m_1 \neq m^* \mid m_1 = m_2, m_2 = m_3)(1 - I_{m_1 \to m_3}) + \\ &I_{m_1 \to m_3} - I_{m_1 \to m_2} + Pr(m_2 = m_3 \mid m_1 \neq m_2) Pr(m_1 \neq m_2) \end{aligned} \tag{8}$$

This approximation reduces the number of invocations of m^* considerably. Specifically, evaluations involving m^* are required only on the subset of records that pass the conditions $m_1 = m_2$ and $m_2 = m_3$. Compared to Eq. 5, the approximation reduces the computations by 3× when computing $I_{m_1 \to m^*}$.

5 Experimental Evaluation

We evaluate Nirvana and several existing proposals on three real-world benchmarks. We also analyze systematically the advantages and limitations of design choices in Nirvana.

5.1 Experimental Setup

5.1.1 Datasets. We use three well-established, publicly available multi-modal datasets for the evaluation; see Table 3. (1) The **Movie**[3] dataset includes 250 movie records sourced from OMDB, each with 22 attributes, including titles, directors, ratings, posters, and plot descriptions. It covers numerical, textual, and image modalities. (2) The **Estate**[4] dataset contains 1,041 estate entries, each with 4 attributes, including property images, project descriptions, locations, and estate details. The modalities include images and long text. (3) The **Game**[5] dataset contains 18,891 video game records with PEGI ratings from Steam. Each record has 21 attributes, including game titles, cover images, release dates, prices, and game summaries. The modalities span dates, numerical values, images, and long text.

5.1.2 Workloads. For each dataset, we construct a query workload comprising 12 queries. These queries are categorized into three groups based on the number of operations involved: (a) *Small*: Each query contains a single operator. *Small* queries usually process all records in a table, and they are too limited for optimization by traditional optimizers, whereas in Nirvana, they can be further optimized through non-LLM operator replacement and physical plan optimization. (b) *Medium*: Each query consists of two to three operators. *Medium* queries present moderate optimization potential for both traditional systems and Nirvana. (c) *Large*: Each query has four or more operators. *Large* queries provide increased opportunities for optimization, and they usually produce a single-value result, allowing us to evaluate result quality across systems. The workloads are available in the extended version [36].

5.1.3 Systems. We compare Nirvana against three existing data analytics systems:

- GPT-4.1. A baseline that prompts the LLM directly by feeding the query in natural language and entire table into the model to obtain analytical results.
- TableRAG [5]. A Retrieval-Augmented Generation (RAG) framework specifically designed for LM-based table understanding.
- Table-LLaVA [35]. A tabular multimodal large language model (MLLM), which treats tables as images and combines table images with user queries as LLM inputs. The model is deployed locally on an Nvidia A100 40G GPU.
- Palimpzest[6]. An LLM-powered analytical processing system that implements several operators to execute analytical queries. It employs a Cascade optimizer to enable logical and physical plan optimization tailored to semantic operators.
- Lotus[7]. A Pandas-like programming framework with LLM-powered semantic operators, each of which can be executed using a standard algorithm or an approximate algorithm.

5.1.4 Implementation. In the experiments, we employ the following OpenAI models as execution backends for semantic operators: `gpt-4.1-nano`, `gpt-4o-mini`, `gpt-4.1-mini`, `gpt-4.1`.The `gpt-4o` model is employed for logical plan optimization. Model pricing follows the official OpenAI pricing policy [20]. For TableRAG, we set the cell encoding budget to 10,000 and the retrieval

[3]https://huggingface.co/datasets/moizmoizmoizmoiz/MovieRatingDB
[4]https://huggingface.co/datasets/Binaryy/multimodal-real-estate-search
[5]https://github.com/triesonyk/data-analysis-steam-games
[6]https://github.com/mitdbg/palimpzest/releases/tag/0.7.7
[7]https://github.com/lotus-data/lotus/releases/tag/v1.1.0

Table 4. Runtime and monetary cost of Nirvana, Table-LLaVA, TableRAG, Palimpzest, and Lotus on the Movie, Estate, and Game benchmarks. Mean values across queries are reported.

Datasets		Movie			Estate			Game		
		Small	*Medium*	*Large*	*Small*	*Medium*	*Large*	*Small*	*Medium*	*Large*
Table-LLaVA	Time (s)	6.103	1.320	1.596	-	-	-	-	-	-
TableRAG	Time (s)	18.048	15.452	17.463	22.178	21.516	24.449	18.725	16.485	15.961
	Cost ($)	0.035	0.028	0.086	0.123	0.153	0.146	0.035	0.042	0.047
Palimpzest	Time (s)	81.754	82.685	163.115	363.671	581.923	918.673	≥ 5.5h	≥ 5.5h	≥ 5.5h
	Cost ($)	0.220	0.126	0.390	1.479	1.660	2.677	-	-	-
Lotus	Time (s)	40.283	62.326	76.178	149.945	252.182	260.099	1375.920	2491.667	3664.475
	Cost ($)	0.170	0.107	0.383	0.748	1.527	1.564	4.116	7.419	10.925
Nirvana	Time (s)	19.963	16.956	21.169	114.196	193.317	233.679	213.767	478.463	728.443
	Cost ($)	0.091	0.011	0.014	0.612	0.524	0.417	0.084	0.187	0.249
	Δ Time	↓ 50.44%	↓ 72.79%	↓ 72.21%	↓ 23.84%	↓ 23.34%	↓ 10.16%	↓ 84.46%	↓ 80.80%	↓ 80.12%
	Δ Cost	↓ 46.48%	↓ 89.50%	↓ 96.31%	↓ 18.16%	↓ 65.90%	↓ 73.37%	↓ 97.96%	↓ 97.48%	↓ 97.72%

Table 5. Quality evaluation of Nirvana, its competitors, and its variants across the three benchmarks.

	Movie	**Estate**	**Game**
GPT-4.1	✗	✗	✗
Table-LLaVA	0%	✗	✗
TableRAG	0%	0%	0%
Palimpzest	58.3%	25.0%	/
Lotus	8.3%	16.7%	25.0%
Nirvana	75.0%	41.7%	50.0%
Nirvana w/o Logical optim.	58.3%	33.3%	41.7%
Nirvana w/ GPT-4.1	58.3%	33.3%	33.3%
Nirvana w/o all optim.	66.7%	41.7%	41.7%

limit 50. To mitigate the impact of LLMs varying in performance, we restrict the LLM choices of all systems to the GPT-4.1 family to ensure fair comparison. For logical plan optimization, the number of optimization iterations N_{max} is set to 3. The parameter λ in Eq. 1 is set to 0.2. The error tolerance ϵ is set to 0.8. For physical plan optimization, the improvement margin ΔI_{min} is set to 20%. The number of data samples is set to 5. To improve throughput, Nirvana leverages concurrent programming with 16 coroutines at the most.

5.2 Overall Results

5.2.1 Settings. We compare Nirvana to two state-of-the-art bulk semantic processing systems, Palimpzest and Lotus, in terms of runtime, monetary cost, and analytical quality. For fair comparison, Palimpzest is configured with access to `gpt-4.1`, `gpt-4.1-nano`, `text-embedding-3-small`, and `clip-ViT-B-32`. All built-in implementation rules in Palimpzest are enabled, and its optimization policy is set to `MinTime` on Movie and Estate and `MinCost` on Game. For Lotus, we use `gpt-4.1` as the backend model and `gpt-4.1-nano` as the helper model. Both Palimpzest and Lotus are configured to use concurrent programming, with the number of workers set to 16.

5.2.2 System runtime. Table 4 presents runtime and monetary cost comparisons across all three datasets. Nirvana consistently outperforms Palimpzest and Lotus in runtime, reducing the execution time by 65.15%, 19.11%, and 81.87% on Movie, Estate, and Game, respectively. Notably, the runtime of Palimpzest on Game exceeds 5.5 hours, making it impractical for real-world deployment. The runtime improvements are attributed to both the logical and the physical optimizations in

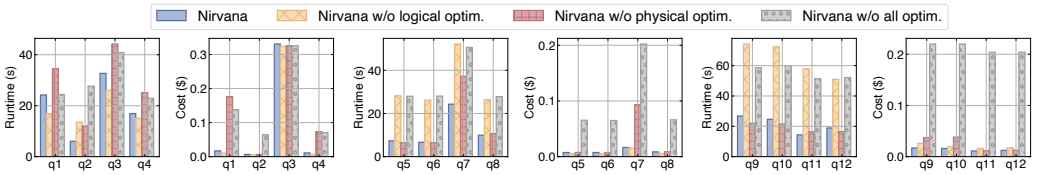


Fig. 6. Ablation study of Nirvana in terms of runtimes and monetary costs on Movie.

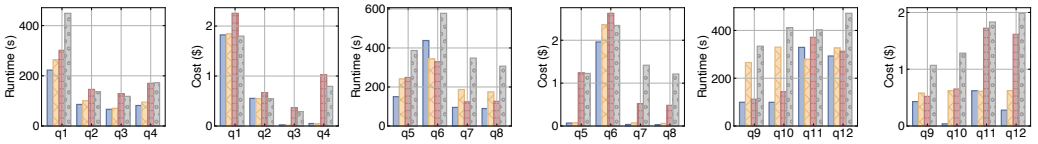


Fig. 7. Ablation study of Nirvana in terms of runtimes and monetary costs on Movie on Estate.

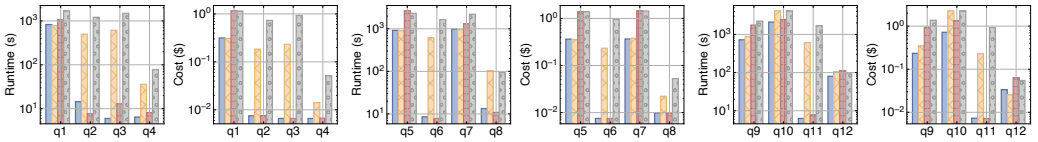


Fig. 8. Ablation study of Nirvana in terms of runtime and monetary costs on Game. The y-axes use a logarithmic scale.

Nirvana. Compared to Lotus, which supports proxy inference using smaller models, Nirvana additionally reduces LLM computations through operator merging, reordering, and replacement. While Palimpzest also supports operator reordering, Nirvana further enhances performance through collaborative execution between large and small models. Although Table-LLaVA performs tableQA efficiently (completing queries in just a few seconds), the image size can easily exceed the input limit as table length grows. TableRAG achieves nearly constant runtime regardless of table size because it retrieves only a fixed number of relevant records. However, as discussed later, both Table-LLaVA and TableRAG exhibit extremely poor answer quality.

5.2.3 System cost. In terms of monetary cost, Nirvana reduces monetary cost significantly compared to Palimpzest and Lotus. The reasons are two-fold: First, the three transformation rules in our logical plan optimizer effectively reduce LLM invocations. Second, the physical optimizer assigns cheaper, smaller models where appropriate. Since this cost reduction applies at the per-record level, the more data is processed, the larger the savings.

5.2.4 Result quality. Table 5 shows the analytical accuracy of Nirvana and the five other systems on the three datasets. The quality is measured as the proportion of queries answered correctly, where the ground truth of each query is labeled manually. The LLM, GPT-4.1, fails to answer any query across the datasets because the table content exceeds the maximum context length. Even on a smaller version of the smallest dataset, Movie, where only 5 out of 22 columns are selected, it fails. Table-LLaVA also fails to produce correct answers, as the image quality is inadequate even on Movie. For Estate (and also Game), the generated image sizes (245,668,890 px) exceed the model's limit of 178,956,970 px. TableRAG achieves an accuracy of 0 because it cannot process data outside its retrieval scope and does not support aggregation operations. Nirvana achieves better accuracy than Palimpzest and Lotus across most datasets. However, all systems struggle on queries involving relatively complex mathematical computations over string-like values, such as calculating the average price of a large group of properties.

Table 6. Comparison w.r.t. cost of processing query q10 on the Estate dataset between Nirvana's logical optimizer and the Cascades optimizer in Palimpzest (PZ). Times are in seconds; monetary costs are in USD.

	Opt. time	Opt. cost	Exec. time	Exec. cost
PZ (Cascades)	~ 0.0	0.0	995.3	2.4
Nirvana	9.8	$8.2E^{-3}$	99.1	0.038

5.3 Logical Plan Optimizer Evaluation

5.3.1 Settings. We perform an ablation analysis to evaluate the effectiveness of Nirvana's logical optimizer. Two variants are considered: (1) Nirvana w/o logical optim. that disables the logical plan optimizer; and (2) Nirvana w/o all optim. that disables both the logical optimizer and physical optimizer.

5.3.2 Runtime and cost. Figures 6, 7, and 8 show runtime and monetary costs for Nirvana and its variants across all queries. Generally speaking, across almost all datasets and workloads, both the logical and physical optimizers independently contribute to performance gains, reducing runtime and cost compared to the variant with no optimizations. Thus, logical and physical plan optimizers play their unique roles in query optimization, reducing both runtime and cost considerably. Runtime and cost differences become more pronounced for larger workloads (*e.g.*, queries q9–q12) and larger datasets (*e.g.*, Game), where optimization opportunities increase. Comparing Nirvana to Nirvana w/o logical optim., we observe that the logical optimizer accelerates query execution and lowers cost notably in most cases, especially for "large" queries (q5–q12), where optimizations like filter pushdown effectively prune irrelevant data early. For smaller workloads (q1–q4), the benefits of logical optimizations are minimal, as these queries usually involve only a single operator. In these cases, the logical plan optimizer in Nirvana has little to do given a single operation, except applying transformation like non-LLM replacement. Indeed, only a few queries — q2 on both Movie and Game — qualify for such replacement. Interestingly, in some cases (*e.g.*, q1 and q3 in Figure 6, and q1 in Figure 8), Nirvana incurs higher runtime and cost than the variant without logical optimization for analytical processing. This is due to the high runtime and cost of the logical plan optimizer that invokes multiple LLMs during the optimization process.

5.3.3 Comparison with Palimpzest. We report the overhead of the logical optimizer in Table 6 as a case study. We see that, processing query q10 on Estate requires 9.8 seconds and 0.82 cents for logical optimization alone, compared to the zero-cost, near-instant optimization performed by the Cascades optimizer in Palimpzest. Reducing this overhead presents an important direction for future research on LLM-powered query optimization.

5.3.4 Result quality. We assess the impact of the optimizers on result quality using the results in Table 5. Nirvana with full optimization achieves the same number of correct answers as its no-optimization variant (Nirvana w/o all optim.), while having considerably lower runtime and cost. However, disabling logical optimization (*i.e.*, Nirvana w/o logical optim.) decreases accuracy slightly. This is likely due to a higher likelihood of inference errors when more records are processed by a weaker model, as fewer filters are applied early. And also, an LLM might make more mistakes than a computing function if non-LLM replacement is not enabled. However, the difference is marginal with typically only one or two additional incorrect results out of 12 results if only using either logical or physical plan optimizer.

5.3.5 Reliability of LLM-as-a-Judge. We collect all query rewrites and their rating scores assigned by the LLM-as-a-Judge from the optimization process. We categorize these rewrites into four groups:

Table 7. Reliability of the LLM-as-a-Judge and its average cost across queries.

	Success rate	Precision	Recall	Cost
Movie	81.6%	85.2%	93.8%	0.0024
Estate	90.0%	95.6%	91.6%	0.0026
Game	86.7%	84.6%	100%	0.0016

Table 8. Logical optimizer w/ semantics v.s. w/o semantics. Times are in seconds and costs are in $\times 10^{-2}$ USD.

	Movie			**Estate**		
	S	M	L	S	M	L
Optim. time (w/ sem)	6.57	7.32	8.20	7.45	8.88	9.49
Optim. time (w/o sem)	6.81	7.49	8.05	7.03	7.87	8.76
Optim. time (2 step)	6.54	7.15	8.20	7.44	8.85	8.17
Overall time (w/ sem)	19.96	12.06	21.17	114.19	193.32	205.08
Overall time (w/o sem)	34.13	32.95	50.82	101.71	192.28	227.92
Overall time (2 step)	19.96	12.06	21.17	114.19	193.32	205.08
Optim. cost (w/ sem)	0.67	0.78	0.83	0.63	0.72	0.89
Optim. cost (w/o sem)	0.53	0.61	0.82	0.57	0.65	0.69
Optim. cost (2 step)	0.52	0.60	0.81	0.55	0.64	0.67
Overall cost (w/ sem)	9.14	1.00	1.41	61.19	52.39	34.21
Overall cost (w/o sem)	9.21	1.22	2.00	61.00	51.69	44.69
Overall cost (2 step)	9.13	0.99	1.40	61.17	52.38	34.19

(a) True Positives (TP): correct rewrite & rating $\geq \epsilon$ (0.8); (b) False Positives (FP): incorrect rewrite & high rating; (c) False Negatives (FN): correct rewrite & low rating; and (d) True Negatives (TN): incorrect rewrite & low rating. In the evaluation, the correctness of a query rewrite is determined by a human expert. We use the following metrics to measure the reliability of the LLM-as-a-Judge: (1) Success rate: $\frac{\#TP+\#TN}{\#rewrites}$; (2) Precision: $\frac{\#TP}{\#TP+\#FP}$; (3) Recall: $\frac{\#TP}{\#TP+\#FN}$. Table 7 reports the results and the cost of verification. The LLM-as-a-Judge achieves $>$ 80% success rate, $>$ 85% precision, and $>$ 90% recall across three benchmarks. These results demonstrate near-human verification quality at a reasonable cost. We also note that failures occur in $\sim$ 10% of cases (failing to abort incorrect rewrites) and $\sim$ 8% of cases (failing to accept correct rewrites), mainly due to low coverage of data samples for validation and heterogeneous operator outputs that are too vague to verify.

5.3.6 Optimizer w/ semantics v.s. optimizer w/o semantics. To assess the effectiveness of semantics in logical optimization, we conduct ablation studies to isolate the impact of semantic-aware transformations (*i.e.*, non-LLM replacement v.s. filter pushdown and operator fusion). Table 8 reports the average runtimes and costs of both optimization and end-to-end execution across workloads. incorporating semantic-aware transformations consistently reduces overall system runtime and cost, albeit with minor optimization overhead, particularly on Movie, where many user-defined operations can be implemented directly as UDFs. In addition, we compare our optimizer with a "two-step" hybrid optimization that first applies basic transformations (*i.e.*, filter pushdown and operator fusion) and then applies semantic-aware ones(*i.e.*, Non-LLM replacement). As shown, the total costs and end-to-end runtimes of the two approaches are nearly identical.

Table 9. Overhead comparison of processing q3 on Estate using Smart and the physical optimization in Nirvana.

	Optim. time (s)	Exec. time (s)	Ratio
Smart (exhaustive)	59.06	626.77	9.42%
Smart (efficient)	53.80	620.95	8.66%
Smart (multi-model)	56.27	625.73	8.99%
Nirvana (Synchronous)	13.11	674.56	1.94%
Nirvana (Asynchronous)	4.12	66.47	5.82%

5.4 Physical Plan Optimizer Evaluation

5.4.1 Settings. We conduct an ablation study of the physical optimizer, comparing Nirvana with two variants: (1) Nirvana w/o physical optim., which disables the physical optimizer, and (2) Nirvana w/o all optim., which disables both optimizers.

5.4.2 Runtime and cost. The physical optimizer provides a robust strategy for reducing runtime and monetary cost. Nirvana w/o physical optim. outperforms the version with no optimization, offering evidence of the physical optimizer's effectiveness. Further, we observe that Nirvana w/o physical optim. often incurs lower costs than Nirvana w/o logical optim. We conclude that the physical optimizer plays a dominant role in cost reduction. Interestingly, we also find that in certain queries — such as q5 and q6 in Figure 6 — logical plan optimization alone yields larger runtime and cost savings than when using also physical plan optimization. Another interesting finding is that logical optimization can bring more reductions in runtime and cost than physical plan optimization in certain cases where a semantic operator can be replaced by a computing operator (*e.g.*, q5 and q6 in Figure 6). This happens when semantic operators are replaced by traditional computing operators, showcasing that the logical optimizer can sometimes apply more aggressive transformations that yield larger runtime reductions.

5.4.3 Result quality. We evaluate the effect of optimizers on result quality; see Table 5. The findings align with the findings in Section 5.3: the impact of optimization on answer quality is minimal.

5.4.4 Comparison with Smart [13]. Model selection is at the core of physical plan optimization. Smart [13] is a recent system that proposes a cost-aware model selection algorithm with accuracy guarantees. However, it focuses on a single filter operator and executes in a non-parallel fashion. It offers three variants: exhaustive, efficient, and multi-model. We compare Nirvana's physical optimizer in both its non-parallel (Synchronous) and parallel (Asynchronous) modes against all three Smart variants where the candidate LLMs are same as those available to Nirvana. Results are shown in Table 9. First, in the synchronous setting, Nirvana and Smart yield comparable execution times, suggesting that both systems make similar model selection decisions during query planning (*i.e.*, assigning GPT-4.1-nano to all operators as the execution backend by both Smart and our physical optimizer). However, Nirvana's optimizer requires significantly less time for optimization — about one-fourth that of Smart. In the asynchronous setting, Nirvana drastically reduces both optimization and execution time, showing that parallel execution can boost throughput.

5.5 Breakdown Analysis

5.5.1 Cost breakdowns. We provide detailed cost breakdowns of Nirvana (including both runtimes and API expenses) on all workloads. Figure 9 shows the runtime and cost distributions across the three main components of Nirvana: the logical optimizer, the physical optimizer, and the query executor. The proportions of optimization time relative to total query evaluation time are 50.7%,

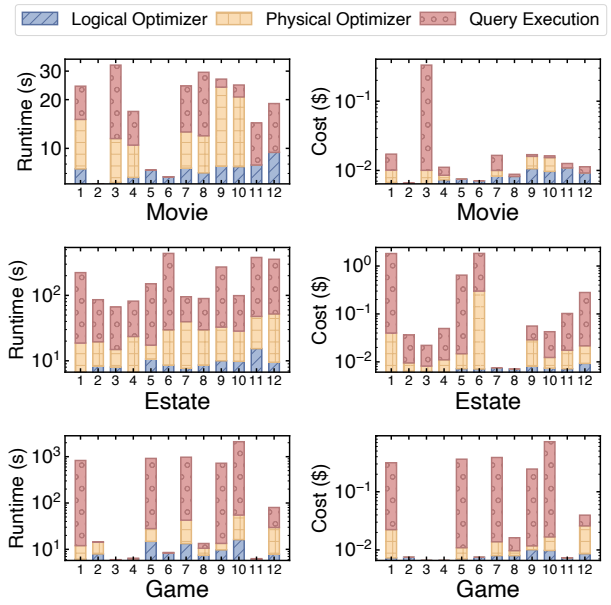


Fig. 9. Breakdown of Nirvana in terms of the runtime and the cost per phase. The y-axes use a logarithmic scale.

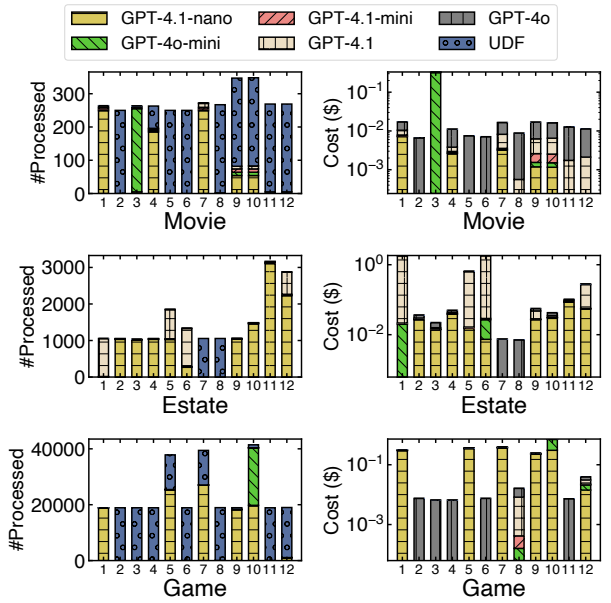


Fig. 10. Breakdown of Nirvana in terms of the number of records processed and prices per models on all workloads.

6.7%, and 42.7% for the Movie, Estate, and Game benchmarks, respectively. By analyzing individual queries, we find that the impact of optimization depends on query execution: for queries using LLM-executed data operations (*e.g.*, Estate), the time taken on optimization is a small fraction of total processing time; for queries executed via UDFs generated by the optimizer (*e.g.*, q5, q6 in Movie;

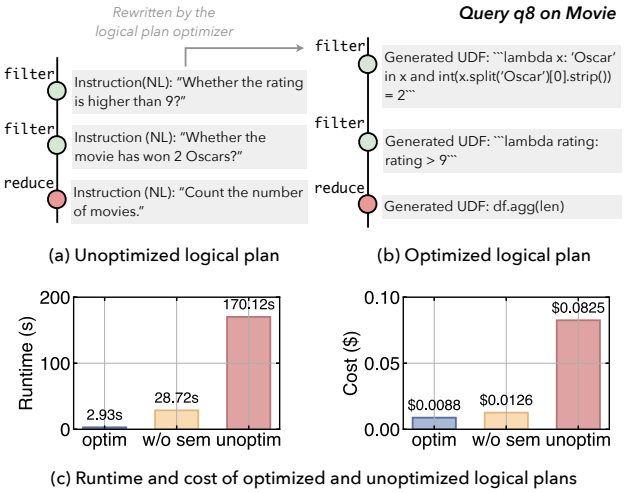


Fig. 11. Case study of query optimization for q8 on Movie. The system runtime and cost when adopting the optimizer, the optimizer w/o semantics, and no optimizer are reported.

q2, q8 in Game), optimization becomes the dominant factor in query processing. Furthermore, the ablation studies in Figures 6, 7, and 8 show that optimization significantly reduces end-to-end runtime costs (*e.g.*, runtimes are reduced by 53.7%, 23.5%, and 44.5% on three datasets). Another notable observation is that the runtime overhead of the logical optimizer remains nearly constant across datasets, incurring only 7–9 seconds on average (Movie: 7.31s, Estate: 9.17s, Game: 9.17s). This stability arises because the logical optimization cost depends primarily on the number of operators in a query rather than the dataset size. These results demonstrate that the benefits of LLM-based optimization outweigh the additional overhead.

5.5.2 Model breakdowns. Figure 10 provides a detailed view of Nirvana in terms of the number of records processed by each model/executor and their corresponding costs. As observed, with the physical optimizer, Nirvana seamlessly transitions to utilizing LLMs with varying intelligence and prices while keeping a high accuracy. For instance, on q1 in Movie, GPT-4.1-nano is responsible for the majority of processing instead of the default GPT-4.1 model, reducing a great deal of cost. Moreover, different LLMs are selectively assigned to different operators within a query to balance accuracy and efficiency. For instance, on q8 in Game, Nirvana employs a combination of GPT-4.1-nano, GPT-4.1-mini, GPT-4o-mini, and GPT-4.1 for query processing.

5.6 Case Studies

5.6.1 Successful cases. Figure 11 illustrates the case of processing q8 on Movie. We notice that the logical optimizer uses two transformation rules to rewrite the initial logical plan: (a) it randomly reorders two filter operators, as the optimizer has no knowledge of their selectivities; and (b) it substitutes the LLM-powered operations with non-LLM operations executed by computing functions. The LLM leverages query semantics to perform complex query rewrites, *e.g.*, converting the instruction "whether the movie has ever won 2 Oscar" into a UDF lambda x: 'Oscar' in x and int(x.split('Oscar')[0].strip())==2. With optimizations, the runtime and cost are reduced by approximately 98% and 89%, respectively.

5.6.2 Failure cases. Figure 12 illustrates representative failure cases in the logical and physical plan optimization. As shown in Figure 12(b), the optimizer generates incorrect UDFs for the map

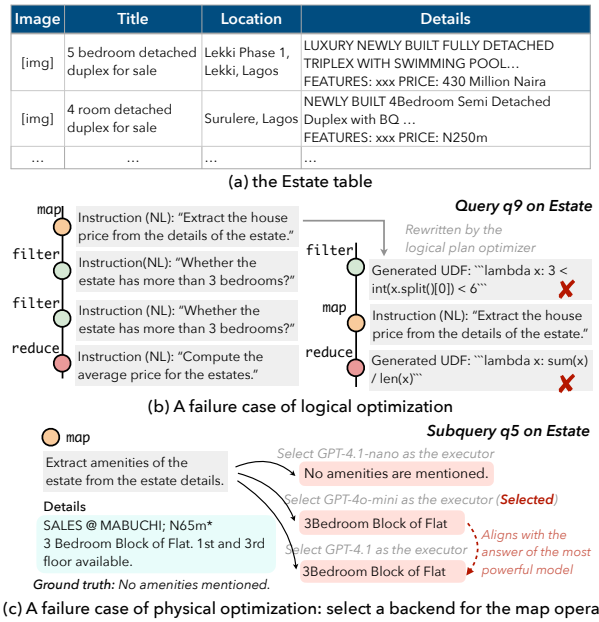


Fig. 12. Failure cases of logical and physical optimization.

and reduce operations. Specifically, the map function works correctly only on a subset of the data records, while reduce cannot process string values. This failure occurs because the optimizer lacks knowledge of the table contents to guide plan rewrites, and the LLM-as-a-Judge fails to detect and abort incorrect rewrites due to insufficient coverage of sampled data. Figure 12(c) demonstrates a failure in physical plan optimization, where GPT-4o-mini is mistakenly assigned to an operator. This occurs because GPT-4o-mini produces the same incorrect output as the most powerful model, GPT-4.1, which is used as a proxy for the unknown ground truth.

6 Related Work

We proceed to review related work and explain how Nirvana differs from existing approaches.

AI-augmented analytical processing systems. The most relevant line of work involves building semantic data analytical processing systems from scratch, where semantic operators are powered by LLMs [2, 6, 16, 17, 19, 21, 24]. Representative examples include Lotus [21] and Palimpzest [17], which share similar motivations with Nirvana. Lotus adopts an optimization strategy that selects an appropriate execution algorithm for each operator under accuracy constraints. Palimpzest introduces the Abacus optimizer [23], inspired by the Cascades optimizer [8] from relational systems. However, its plan cost estimation and analysis quality evaluation remain coarse-grained. Compared to them, Nirvana attempts to explore conceptually distinct optimization strategies and discuss their pros and cons through experimental evaluations.

Another line of research focuses on integrating AI-powered semantic operators into mature relational DBMSs [14, 26, 28]. For example, ThalamusDB [14] adds a semantic filter operator powered by pretrained models for unstructured data (*e.g.*, text, images, audio) into DuckDB. CAESURA [28] augments traditional SQL operators (*e.g.*, joins, aggregations) with semantic operators such as TextQA and VisualQA to support reasoning over text and images. A key challenge is the difficulty of co-optimizing relational and semantic operators within conventional relational architectures.

Workflow generation by LLMs. Workflow generation for reasoning tasks via LLMs is conceptually related to logical plan optimization. Aflow [32] samples reasoning workflows from scratch to achieve high accuracy on tasks such as math and commonsense QA. ADAS [12] treats workflow generation as code generation and leverages LLMs to generate reasoning programs. GPTSwarm [37] models workflow generation as computation graph construction and proposes a graph generation strategy. Unlike these systems, Nirvana focuses on optimizing logical plans for semantic data analytical processing. It draws from traditional database principles such as rule-based transformations and cost-based optimization, while balancing analysis quality and system cost.

Optimizations for compound AI systems. Several recent efforts aim to optimize compound AI systems using techniques such as model selection and query reordering. QuestCache [18] reduces LLM invocation costs for semantic data analytics by reordering table rows and columns to improve KV-cache reuse. Smart [13] reduces the cost of semantic processing by selectively replacing powerful LLMs with smaller, cheaper models while maintaining similar output quality. While sharing a similar goal, Nirvana's physical plan optimizer adopts a distinct strategy. It introduces a novel cost-effectiveness metric and incorporates optimization techniques to reduce both runtime and monetary cost. Doctopus [4] focuses on extracting data (e.g., database schema, relations) from unstructured documents, which optimizes the extraction accuracy under a user-specified budget.

7 Conclusion

With the goal of enabling semantic analytical processing over multi-modal data, we present Nirvana, a new LLM-powered semantic data analytics system equipped with semantic operators. To minimize system costs, Nirvana incorporates two novel query optimization techniques: an agentic logical plan optimizer based on random walk search, and a physical plan optimizer that allocates cost-effective models to operators in a query plan. Extensive experiments show that Nirvana is able to consistently outperform existing semantic data analytics systems in terms of both runtime and monetary cost.

Acknowledgments

We thank three anonymous reviewers for their helpful comments on our manuscript. This work was supported in part by the NSFC under Grants No. (62472377, 62502434, 62402422) and Ningbo Yongjiang Talent Introduction Programme (2022A-237-G). Lu Chen is the corresponding author of the work.

References

[1] Amazon. 2025. S3: Cloud Object Storage. https://aws.amazon.com/s3/.

[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. 2024. The Design of an LLM-powered Unstructured Analytics System. *CoRR* abs/2409.00847 (2024).

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). 1383–1394.

[4] Chengliang Chai, Jiajun Li, Yuhao Deng, Yuanhao Zhong, Ye Yuan, Guoren Wang, and Lei Cao. 2025. Doctopus: Budget-aware Structural Table Extraction from Unstructured Documents. *Proc. VLDB Endow.* 18, 11 (2025), 3695–3707.

[5] Si-An Chen, Lesly Miculicich, Julian Eisenschlos, Zifeng Wang, Zilong Wang, Yanfei Chen, Yasuhisa Fujii, Hsuan-Tien Lin, Chen-Yu Lee, and Tomas Pfister. 2024. TableRAG: Million-Token Table Understanding with Language Models. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2024/hash/88dd7aa6979e352fda7c4952ca8eac59-Abstract-Conference.html

[6] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Binding Language Models in Symbolic Languages. In *ICLR*. https://openreview.net/forum?id=lH1PV42cbF

[7] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proc. ACM Manag. Data* 1, 4 (2023), 227:1–227:26.

[8] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[9] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. 209–218.

[10] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *SIGKDD*. 855–864.

[11] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*. https://openreview.net/forum?id=nZeVKeeFYf9

[12] Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated Design of Agentic Systems. In *ICLR*. https://openreview.net/forum?id=t9U3LW7JVX

[13] Saehan Jo and Immanuel Trummer. 2024. SMART: Automatically Scaling Down Language Models with Accuracy Guarantees for Reduced Processing Fees. *CoRR* abs/2403.13835 (2024).

[14] Saehan Jo and Immanuel Trummer. 2024. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *Proc. ACM Manag. Data* 2, 3 (2024), 186.

[15] 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 *SOSP*. 611–626.

[16] 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).

[17] Chunwei Liu, Matthew Russo, Michael J. Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael J. Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *CIDR*.

[18] 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 *MLsys*.

[19] Samuel Madden, Michael J. Cafarella, Michael J. Franklin, and Tim Kraska. 2024. Databases Unbound: Querying All of the World's Bytes with AI. *Proc. VLDB Endow.* 17, 12 (2024), 4546–4554.

[20] OpenAI. 2025. Pricing policy. https://openai.com/api/pricing/.

[21] Liana Patel, Siddharth Jha, Parth Asawa, Melissa Pan, Carlos Guestrin, and Matei Zaharia. 2024. Semantic Operators: A Declarative Model for Rich, AI-based Analytics Over Text Data. *CoRR* abs/2407.11418 (2024).

[22] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP-IJCNLP*. 3980–3990.

[23] Matthew Russo, Sivaprasad Sudhir, Gerardo Vitagliano, Chunwei Liu, Tim Kraska, Samuel Madden, and Michael Cafarella. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. *CoRR* abs/2505.14661 (2025).

[24] Shreya Shankar, Aditya G. Parameswaran, and Eugene Wu. 2024. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *CoRR* abs/2410.12189 (2024).

[25] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2023/hash/77c33e6a367922d003ff102ffb92b658-Abstract-Conference.html

[26] MindsDB Team. 2025. MindsDB: AI's query engine. https://mindsdb.com/.

[27] TPC-H. 2025. TPC-H V3 Top Performance Results, 30-Jun-2025 at 9:01 AM. https://www.tpc.org/tpch/results/tpch_perf_results5.asp?resulttype=all&version=3.

[28] Matthias Urban and Carsten Binnig. 2024. CAESURA: Language Models as Multi-Modal Query Planners. In *CIDR*. https://www.cidrdb.org/cidr2024/papers/p14-urban.pdf

[29] Jiayi Wang, Guoliang Li, and Jianhua Feng. 2025. iDataLake: An LLM-Powered Analytics System on Data Lakes. *IEEE Data Eng. Bull.* 49, 1 (2025), 57–69.

[30] Yifan Wang, Haodi Ma, and Daisy Zhe Wang. 2024. No more optimization rules: LLM-enabled policy-based multi-modal query optimizer. *CoRR* abs/2403.13597 (2024).

[31] Yicun Yang, Zhaoguo Wang, Yu Xia, Zhuoran Wei, Haoran Ding, Ruzica Piskac, Haibo Chen, and Jinyang Li. 2025. Automated Validating and Fixing of Text-to-SQL Translation with Execution Consistency. *Proc. ACM Manag. Data* 3, 3 (2025), 134:1–134:28.

[32] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025. AFlow: Automating Agentic Workflow Generation. In *ICLR*. https://openreview.net/forum?id=z5uVAKwmjf

[33] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html

[34] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html

[35] Mingyu Zheng, Xinwei Feng, Qingyi Si, Qiaoqiao She, Zheng Lin, Wenbin Jiang, and Weiping Wang. 2024. Multimodal Table Understanding. In *ACL*. 9102–9124.

[36] Junhao Zhu, Lu Chen, Xiangyu Ke, Ziquan Fang, Tianyi Li, Yunjun Gao, and Christian S. Jensen. 2025. Beyond Relational: Semantic-Aware Multi-Modal Analytics with LLM-Native Query Optimization. *CoRR* abs/2511.19830 (2025). https://doi.org/10.48550/arXiv.2511.19830

[37] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. GPTSwarm: Language Agents as Optimizable Graphs. In *ICML*. https://openreview.net/forum?id=uTC9AFXIhg

Received July 2025; revised October 2025; accepted November 2025