Unify: An Unstructured Data Analytics System

Jiayi Wang
Tsinghua University
jiayi-wa20@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

Jianhua Feng
Tsinghua University
fengjh@tsinghua.edu.cn

***Abstract*—Despite unstructured data constituting over 80% of the information available today, no specialized analytics system currently exists to process this type of data. The lack of a predefined schema in unstructured data renders traditional structured query languages, such as SQL, unsuitable for unstructured data analytics. A seemingly straightforward solution is to utilize natural language for crafting queries, thereby making analytics more accessible to users without technical expertise. However, understanding and executing queries posed in natural language presents significant challenges. A brute-force approach involves relying on users to manually derive solutions, tapping into their knowledge and experience. However, this method of generating query plans by human intervention is neither scalable nor efficient. Therefore, a pertinent question arises:** *how can we automate unstructured data analytics?*

To address these challenges, this paper introduces `Unify`, an innovative system leveraging the capabilities of large language models (LLMs) to automatically generate, optimize, and execute query plans for unstructured data analytics, where queries are articulated in natural language. `Unify` initializes by defining common operators used in unstructured data analytics and creates both pre-programmed implementations and LLM-based implementations for physically executing these operators. It then guides LLMs to devise logical plans by methodically deconstructing queries into smaller steps, ensuring accurate logic by aligning with suitable operators. For translating a logical plan into an optimal physical plan, we further introduce a technique for physical plan optimization that employs a semantic cost model alongside semantic cardinality estimation. Comprehensive tests conducted on real-world datasets demonstrate that `Unify` can expedite query processing by up to $40\times$, while preserving high accuracy, thus positioning `Unify` as an effective tool for large-scale unstructured data analytics.

***Index Terms*—data analytics, unstructured data.**

I. Introduction

Unstructured data, such as text, PDFs, videos, and audio, accounts for over 80% of the information available today [21]. This type of data is prevalent across various domains, including social media, healthcare, and legal documentation, and harbors significant potential for insight extraction. However, in contrast to structured data that easily aligns with tables and predefined schemas, unstructured data does not have a specific structure, thereby posing challenges for analysis using conventional methods. Therefore, the need for innovative approaches in unstructured data analytics is critical.

The first obstacle lies in expressing the objective of unstructured data analytics, particularly in crafting the query. Given the absence of a schema in unstructured data, traditional structured queries, such as SQL, are not applicable. A logical strategy uses natural language for formulating queries, making them compatible with unstructured data and more user-friendly for users without technical expertise.

For example, consider a scenario of analyzing a collection of Web pages from "Sports Stack Exchange", and an analytical query, "Among questions with over 500 views, which sport involving a ball has the highest ratio of number of injury-related questions to number of training-related questions?" This example illustrates that natural language queries specify the desired outcome without detailing how to attain it.

However, the second obstacle is how to execute the query, because answering the query necessitates a sophisticated reasoning process. For instance, a possible plan of answering this query involves *filtering* documents (*e.g.,* excluding non-ball sports, *selecting* questions with more than 500 views), *grouping* documents by *extracted* sports, *filtering* injury-related and training-related questions for *computing* the ratio for each group, and then *comparing* these ratios to determine which sport has the highest ratio. A natural strategy to answer this query involves allowing users to manually create execution plans, relying on them to craft solutions based on their expertise and knowledge.

Nevertheless, relying on humans to create query plans is neither scalable nor efficient. Therefore, a pertinent question arises: *how can we make this process automatic?*

Central Problem. The central problem we aim to address is: *Given a collection of unstructured data and an analytics query formulated in natural language, is it possible to automatically generate an optimized execution plan for this query that can be executed efficiently to produce accurate results?*

Fortunately, large language models (LLMs) [33], [26] have gained recognition as promising instruments owing to their impressive capabilities in semantic comprehension. They excel in interpreting natural language and reasoning, allowing us to use them to generate an execution plan for a given query.

Key Idea. Our core idea stems from the insight that execution plans for unstructured data analytics, as crafted by humans, are methodically structured assemblies incorporating standard predefined operators (*e.g.,* filter, selection, count, comparison). Therefore, our initial step involves identifying and predefining these common operators. Upon receiving a query, we proceed to construct the execution plan by systematically matching segments of the query with the most relevant operators. This iterative process breaks down the query into increasingly refined smaller, more manageable sub-queries, until it is fully composed of predefined operators. Throughout this iterative process, we generate a series of potential execution plans,

subsequently refining them to enhance both accuracy and efficiency. This meticulous optimization leads to the selection of the most effective execution plan for implementation.

Challenges. However, answering unstructured data analytics queries faces the following challenges:

C1. Plan Accuracy. Accurately answering these queries requires complex multi-step reasoning. For instance, as demonstrated in the previous example, it entails the integration of diverse filtering, grouping, and aggregation operations, with the planning process heavily dependent on a semantic comprehension of the query. Yet, contemporary LLMs face difficulties in formulating precise and complex plans for such tasks. Consequently, the first challenge lies in devising a logical workflow for a given query that accurately mirrors the required reasoning to yield correct responses.

C2. Plan Efficiency. LLMs are computationally expensive, especially for large input sizes. The inefficiency is compounded by repeated invocations of LLMs, leading to substantial computational overhead. Moreover, the interdependencies among operators within the plan typically necessitate sequential execution, which further decelerates the process. Therefore, optimizing the generation of plans to minimize the total execution time represents the second challenge.

Our Approach. We propose `Unify`, the first system for unstructured data analytics in natural language. Our method consists of three key steps: generating logical plans that ensure correct reasoning, transforming logical plans into optimized and efficient physical plans, and executing the plans to compute the final answer.

Logical Plan Generation. Unlike traditional logical plan generation in databases, which mandates that queries strictly adhere to specific syntactic and semantic requirements, ensuring operators within the query precisely align with predefined operators, our system introduces a more flexible approach. It operates without stringent syntactic and semantic constraints, allowing for operators in the query to semantically correspond with predefined operators, albeit not always exactly. This flexibility, however, introduces complexity, making the accurate generation of logical plans a challenging endeavor. To address this challenge (for C1), the query is decomposed iteratively, breaking it down operator by operator. This process involves aligning the query with predefined operators to progressively reduce its complexity. In each iteration, the most suitable operator is selected to refine the query through a two-step strategy. This decomposition continues until the query is fully decomposed, and the sequence of matched operators constructs a logical plan capable of effectively addressing the query. The first step uses a semantic embedding similarity check to filter out operators deemed irrelevant. The second step instructs an LLM to re-rank the remaining operators, prioritizing them according to their relevance to the query.

Physical Plan Generation. Different from traditional query optimization in databases, where a cost model and statistical cardinality estimation built over structured data are employed, unstructured data presents more challenges. In structured environments, cost estimation relies on predefined statistical models, such as histograms, to estimate cardinalities. However, these methods are not applicable to unstructured data, which involve more complex operators dealing with semantic information. For example, cardinality estimation for semantic queries cannot depend on purely statistical techniques, as they must account for the content-based relationships within the data. To address this challenge and generate efficient physical plans for execution (for C2), we analyze the operators for unstructured data and design a cost model tailored for unstructured data analytics operators. Besides, to provide accurate cost estimation, we formally define the problem of cardinality estimation over unstructured data and propose an importance-sampling-based method that leverages the semantic relevance of data and query in the embedding space. This technique allows us to predict the size of intermediate results at various steps of the plan, thus enabling a suitable ordering of operators based on the expected cost and selecting the most appropriate physical implementations for each operator. In this way, we can ultimately choose the physical plan that offers the highest expected accuracy and performance.

Contributions. Overall, we make the following contributions.

(1) We propose `Unify`, the first system for automatic query plan generation and execution for supporting natural language analytics over unstructured data.

(2) We propose a logical plan generation algorithm that constructs logical plans capable of solving complex queries through correct logical reasoning.

(3) We propose physical plan optimization techniques to transform logical plans into efficient physical plans, based on a novel cost model and semantic cardinality estimation.

(4) We evaluate our system `Unify` on real-world benchmarks for unstructured data analytics, demonstrating that `Unify` generates optimized plans and reduces execution time by up to $40\times$, while achieving high analytics accuracy.

II. Preliminary

A. Problem Definition

Problem Formulation. Given a document collection consisting of N documents and a natural language (NL) analytics query, the objective is to answer the query based on these unstructured documents. The brute-force method involves experts manually converting the query into an execution plan and then executing this plan on the documents. We aim to automate this process, targeting both high accuracy and low latency.

Supported Data Types. We assume that documents have been pre-processed into plain text, enabling us to sidestep issues related to document formatting and concentrate on the complexities of unstructured data analytics.

Supported Query Types. We support a wide range of NL analytical queries. These queries enable operations analogous to SQL, such as selection, projection, and aggregation of documents. Beyond traditional SQL-like capabilities, `Unify` incorporates advanced semantic functions, *e.g.,* semantic filtering and grouping. This combination of both conventional and semantic features empowers users to derive deeper insights.

TABLE I
COMPARISON BETWEEN CONVENTIONAL RELATIONAL DATABASES FOR STRUCTURED DATA AND `Unify` FOR UNSTRUCTURED DATA.

Key Components	Conventional Relational Databases (Structured Data Analytics)	Unify (Unstructured Data Analytics)
Logical Operators	Limited to structured operations (*e.g.,* select, projection, join, aggregate)	Both structured and semantic operators (*e.g.,* semantic filtering, extraction, and other LLM-driven operations)
Physical Operators	Programmed function based implementation	Both pre-programmed implementations and LLM-based implementation
Logical Plan Generation	Map SQL queries with rigid schema and strict syntactic into operators	Break down natural language queries, without a schema, to semantically aligned operators step by step
Physical Plan Generation	Cost model for predefined functions with statistical cardinality estimation	Cost model for both pre-programmed and LLM-based implementations with semantic cardinality estimation
Query Execution	Execute predefined operators sequentially following the plan	Interactive execution with dynamic replanning based on intermediate results
Optimization Strategy	Fixed query optimization rules and cost models for structured data	Adaptive optimization with semantic reasoning and LLM feedback

B. Related Work

Unstructured Document Analytics. Recent advancements in large language models (LLMs) have greatly expanded the potential for exploiting the value of large and complex unstructured data [24], [27], [22], [21], [9], [35], [8]. `LOTUS` [27] enables semantic queries across both structured and unstructured data by converting unstructured data into tables using LLMs. However, it still requires manually written pandas-like code to orchestrate the analytical pipeline. `ZENDB` [21] indexes templatized documents by leveraging their inherent text structures, but its reliance on document templates limits flexibility for general document formats. `DocETL` [32] optimizes complex document processing pipelines using LLM-powered agents to automatically rewrite and evaluate pipelines for improved accuracy through rewrite directives and sub-plan optimization. However, the optimization process can be computationally expensive and time-consuming, particularly for large datasets, due to the need for extensive LLM calls and validation steps. `Evaporate` [9] extracts tables from documents by LLM-generated code and synthesized rules, yet analysis still requires SQL queries, constraining semantic analytics on the original unstructured data. `PALIMPZEST` [22] optimizes both logical and physical plans for large-scale tasks like information extraction and multimodal analytics, but it relies on user-defined schemas and logical plans. Contrary to these approaches, `Unify` eliminates the need for predefined document structures or user-written code, allowing users to perform analytics directly through natural language queries.

Retrieval Augmented Generation (RAG). RAG amplifies the knowledge of LLMs for answering queries by retrieving semantically relevant information from a text corpus [14], [36], [10]. Both RAG and `Unify` involve retrieving information from unstructured data. However, RAG is limited to *point lookups* [27], assuming that a query can be answered by returning relevant documents. This assumption does not hold for unstructured data analytics, which requires handling complex patterns, such as aggregating across multiple documents and performing sophisticated reasoning over intermediate results.

NL2SQL. NL2SQL translates natural language (NL) queries into structured SQL queries for executing data analytics over relational databases [20], [23], [34]. Both NL2SQL systems and `Unify` take the NL query as input, but NL2SQL adheres to strict syntactic and schema requirements of databases. This enables NL2SQL methods to break down the SQL generation into sub-tasks aligned with these constraints [13], [29]. In contrast, `Unify` poses additional challenges due to the absence of well-defined structural boundaries.

III. `Unify` FRAMEWORK

The `Unify` framework, illustrated in Figure 1, consists of three key components. First, `Unify` pre-processes the unstructured data and operators offline for answering queries efficiently (Section III-A). Then, given a query, `Unify` automatically generates and refines an execution plan to answer the query both accurately and efficiently (Section III-B). Lastly, `Unify` judiciously executes the optimized plan to ensure high performance and robustness (Section III-C).

A. Preprocessing

Before executing a query, `Unify` indexes both operators and unstructured data to ensure efficient planning and execution.

Operators. `Unify` utilizes a set of manually defined operators, which are frequently used in unstructured data analytics, such as `Filter`, `Extract`, `Compare`, and support adding new operators to handle uncovered cases. Each operator has a predefined input, output, and execution specification. For example, the `Filter` operator takes a list of documents as input and outputs only those that meet a specified condition, such as "2000 $\leq$ MovieMade $\leq$ 2010". To align operators with natural language, we establish *logical representations* for usage patterns of each operator, describing its functionality in natural language.

Definition 1 (Logical Representation): A logical representation is a structured natural language expression template designed to encapsulate the semantic essence of NL expressions. It abstracts essential elements into placeholders, such as Entity, Condition, etc., each denoting distinct semantic roles.

For instance, the logical representation "[Entity] that [Condition]" for the `Filter` operator can match phrases like "Movies that were made in the 2000s" or "Movies that were produced between 2000 and 2010." This flexible representation allows `Unify` to identify operators involved in a query, regardless of the specific condition or phrasing. Note that each operator may have multiple logical representations. For instance, the `Filter` operator has multiple logical representations such as "*[Entity] having [Condition]*", "*[Entity] that satisfies [Condition]*", etc.

Additionally, each operator can have one or more physical implementations of two categories: **pre-programmed implementations** and **LLM-based implementations**. For instance, the aforementioned `Filter` operator could be implemented by filtering documents with keyword-matching functions or processing each document by prompting an LLM to evaluate whether it meets the filtering criteria.

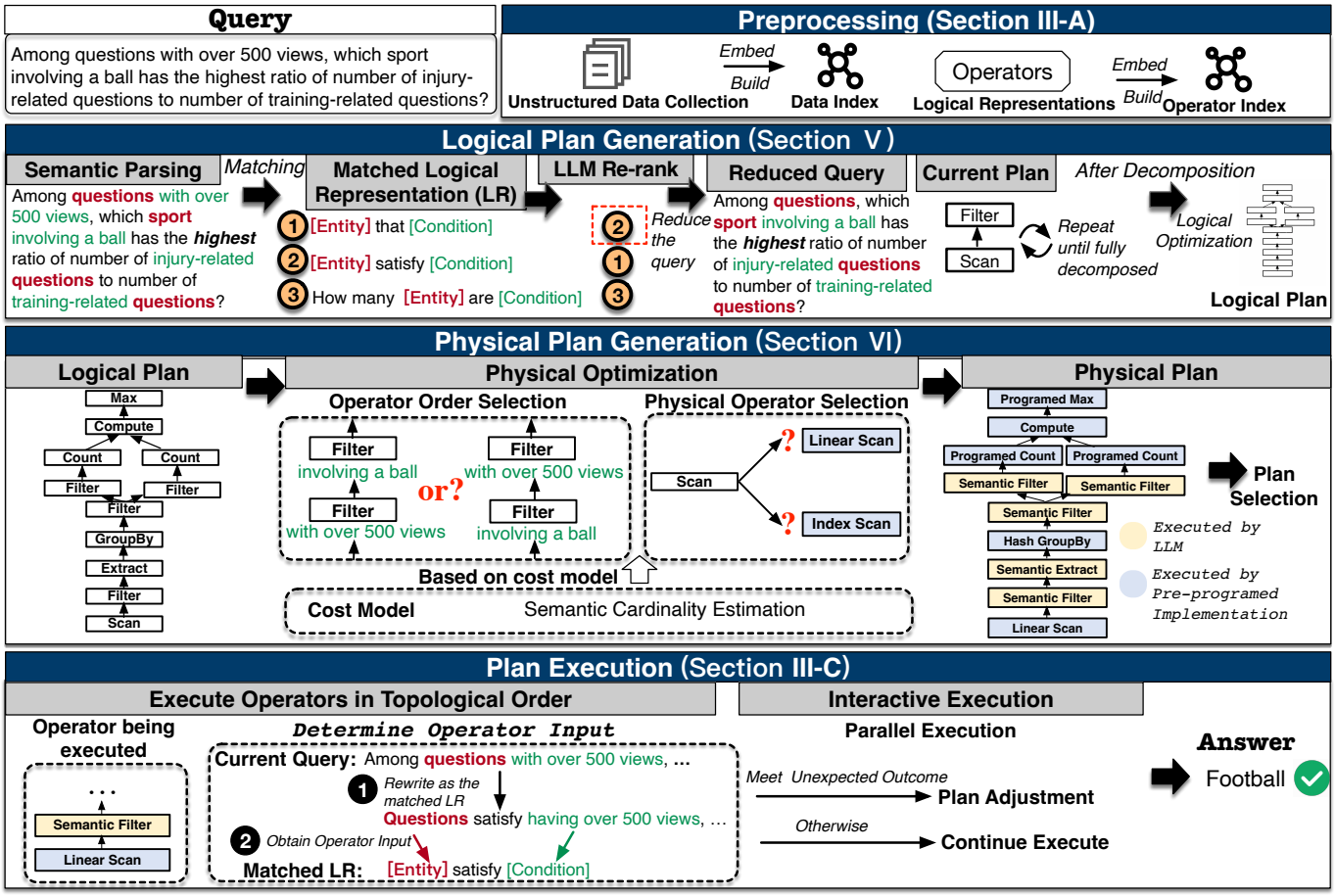


Fig. 1. Framework of Unify.

Indexing. To effectively extract pertinent information from extensive unstructured datasets, Unify employs a text embedding model to convert text sentences into vectors and then organizes these vectors using a vector index like HNSW [25] for efficient retrieval. In addition, the text embedding model also converts logical representations of the operators into embeddings for efficient matching of operators with a query, thus aiding in the efficient construction of execution plans (Section III-B).

B. Planning Engine

Upon receiving a natural language query, the Planning Engine in Unify generates an optimized execution plan with two steps: *Logical Plan Generation* and *Physical Plan Generation*.

1) Logical Plan Generation: The logical plan is constructed by progressively identifying appropriate pre-defined logical operators and reducing the query with the operators.

Semantic Parsing. To accurately capture the intent of the query and align it with appropriate operators, the LLM is prompted to identify the semantic elements of the NL query, *e.g.,* Condition, Entity, for extracting the logical representation of the query (*e.g.,* the converted query in Figure 1, where the red and green texts denote values for Entity and Condition, respectively). This process simplifies the query, thus enabling accurate planning without being misled by specific values.

Logical Plan Generation Algorithm. Unify constructs the logical plan by recursively reducing the query in a depth-first search (DFS) manner, following two key steps:

Step 1: Operator Matching. Unify first identifies appropriate operators for query reduction in two steps: (1) computes embedding distances between the logical representations of the query and the operators to select candidates with the smallest distances; (2) prompts the LLM to check the applicability of each candidate operator and reorders them based on the LLM output. Since logical representations semantically encapsulate the query logic and operator logic, the initial embedding distance filtering efficiently narrows down relevant operators and minimizes the cost of subsequent LLM evaluations.

Step 2: Query Reduction. Unify then iterates the operator list to solve part of the query. For each operator, Unify prompts the LLM to check whether the prerequisites of the operator are satisfied. If so, the LLM is prompted to rewrite the query by reducing the matched segment according to the logical representation of the operator. For example, in Figure 1, the query is reduced by removing the condition *with over 500 views* by the logical representation "[Entity] satisfy [Condition]" of the Filter operator. The reduced query will then be taken as input to the above process until it is fully

reduced.

Error Handling. If all candidate operators cannot reduce the query, `Unify` backtracks to the query before the latest reduction. In cases where no reduction path can fully decompose the query, `Unify` appends a `Generate` operator to the top of the plan and instructs the LLM to solve the remaining parts.

Logical Plan Optimization. The generated plan is further optimized by analyzing the dependencies between operators to reorganize it into a directed acyclic graph (DAG) form for parallel operator execution, as will be discussed in Section V-C.

2) Physical Plan Generation: Once a logical plan is determined, `Unify` converts it into an optimized physical plan, detailing how each operator will be executed.

Cost Model. Unlike relational databases, where operator costs are typically driven by I/O factors, `Unify` focuses on computational costs due to the frequent involvement of resource-intensive LLMs. For LLM-related operators, costs are predominantly determined by token count and the number of LLM calls. For pre-programmed operators, costs depend on input size and computational complexity. `Unify` builds a unified cost model that estimates time consumption for different physical operators (Section VI-A), which is crucial for selecting efficient operator order, and appropriate physical operator, thereby optimizing the overall plan, as will be introduced in Section VI.

Semantic Cardinality Estimation. `Unify` employs a semantical importance-aware sampling method (Section VI-B) for estimating the size of intermediate results generated by each operator. This is crucial for physical plan generation since the cost heavily depends on the cardinality of intermediate outputs.

C. Execution Module

Determining Operator Input. Each operator in the plan contains placeholders representing its input in the logical representation. During execution, these placeholders must be filled. To achieve this, `Unify` instructs the LLM to rewrite the matched query segment into the form of the matched logical representation. For example, the query *"Among questions with over 500 views, ..."* is rewritten as *"Questions satisfy having over 500 views, ..."* to align with the logical representation *"[Entity] satisfy [Condition]"*. Using regular expressions, `Unify` extracts the actual values for these placeholders. In this case, "Questions" and "having over 500 views" are filled as the [Entity] and [Condition] inputs for the `Filter` operator. This method is employed across all operators, facilitating the plan's execution.

Parallel Topological Execution. To maximize efficiency, execution follows a bottom-up, parallelized way based on the topological order of the plan. Leveraging the DAG-structured plan, operators with no interdependencies are executed in parallel until all operators are completed, producing the final result. By improving resource utilization through parallel execution, the overall execution efficiency is improved.

Plan Adjustment During Execution. If an operator fails to produce the expected result, `Unify` dynamically adjusts the plan by continuously replanning to handle intermediate queries, avoiding a complete restart.

IV. Unstructured Data Analytics Operators

This section introduces the operators used to compose the query plan. In total, we have extracted 21 operators.

A. Logical Operators

Traditional relational operators are insufficient for unstructured data analytics, as they lack semantic processing capabilities. To address this, we manually identify a set of operators to support more comprehensive analytics. Each operator is defined with input, output, and the predefined execution process, and a set of logical representations, as shown in Table II. While some operators have functionality similar to traditional database operators, they are extended for unstructured data and equipped with semantic analytics capabilities, *e.g.,* support filtering with semantic checking of natural language conditions.

B. Physical Operators

We describe the physical implementations of the logical operators introduced in Section IV-A. Each logical operator can be realized by one or more physical implementations. We categorize physical operators into two types: (1) **Pre-programmed Implementations**, which are implemented by pre-programmed algorithms, similar to database operators, and (2) **LLM-based Implementations**, which utilize LLMs to handle semantic operators that require reasoning, *e.g.,* filtering sports documents based on whether the sport requires teamwork. Most logical operators feature two types of physical implementations: pre-programmed functions for simple tasks and LLM-based approaches for operations that demand a thorough semantic comprehension. This dual approach for each logical operator significantly increases `Unify`'s adaptability in handling unstructured data.

1) Pre-programmed Implementations: Pre-programmed implementations employ fixed and predefined execution processes. These implementations are suitable for operations that do not require semantic understanding, such as exact matching or basic computations. For instance, the `GroupBy` operator can be implemented using either `HashGroupBy`, which leverages a hash table, or `SortGroupBy`, if the data is already sorted by the grouping attribute. Similarly, the `Extract` can use keyword search or regular expressions to retrieve specific information.

2) LLM-based Implementations: For tasks requiring semantic understanding or complex processing of unstructured data, LLM-based implementations are used. We call these "semantic" operators since they utilize LLMs to produce results tailored to specific tasks through prompts. Every operator is associated with a pre-defined prompt template, which is dynamically filled with particular query values. For example, the `SemanticFilter` operator constructs a prompt describing the filtering conditions and applies it to each element in the input list by invoking the LLM. As these operators rely on

TABLE II
THE LOGICAL OPERATORS, THEIR INPUTS, OUTPUTS, CORRESPONDING PHYSICAL OPERATORS, AND EXAMPLE LOGICAL REPRESENTATIONS.

Operator	Input	Output	Pre-programmed Implementation	LLM-based Implementation	Example Logical Representation
`Scan`	List	List	Linear Scan, Index Scan	-	documents satisfy [Condition]
`Filter`	List	List	Exact condition filtering	Semantic filtering	[Entity] that [Condition]
`Compare`	A, B, Condition	A/B	Standard comparison, *e.g.,* $>$, $<$	Semantic comparison	larger in [Entity] and [Entity]
`GroupBy`	List	List of List	Grouping by exact attributes	Semantic grouping	aggregate [Entity] by [Attribute]
`Count`	List	Number	Standard aggregation (Count)	Semantic count	number of documents [Condition]
`Sum`	List	Number	Standard aggregation (Sum)	Semantic sum	the total sum of [Entity]
`Max`	List	Number	Standard aggregation (Max)	Semantic max	the maximum of [Entity]
`Min`	List	Number	Standard aggregation (Min)	Semantic min	the minimum of [Entity]
`Average`	List	Number	Standard aggregation (Average)	Semantic average	the mean of [Entity]
`Median`	List	Number	Standard aggregation (Median)	Semantic median	the median of [Entity]
`Percentile`	List	Number	Standard aggregation (Percentile)	Semantic percentile	the k-th percentile for [Entity]
`OrderBy`	List	List	Numerical/lexicographical sort	Semantic sorting	Sort [Entity] [Condition]
`Classify`	Text	Class	Rule-based/ML-based classification	Semantic classification	The type of [Entity]
`Extract`	Text	Text	Keyword/Regex extraction	Semantic extraction	get [Entity] from documents
`TopK`	List	List	Numeric ranking	Semantic ranking	the top [Number] [Entity]
`Join`	List, List	List	Join by key	Semantic join	[Entity] that also occurs in [Entity]
`Union`	Set, Set	Set	Standard set union	Semantic set union	set union of [Entity] and [Entity]
`Intersection`	Set, Set	Set	Standard set intersection	Semantic set intersection	in set [Entity] and in [Entity]
`Complementary`	Set, Set	Set	Standard set complementary	Semantic set complementary	in set [Entity] not in [Entity]
`Compute`	List	Number	Programmed Mathematical Equation	Semantic computation	sum of squares of [Entity]
`Generate`	Text	Text	-	LLM invocation	explain the result

LLM invocations for each input, they tend to be more computationally expensive. However, these operators are essential for handling tasks that require semantic inference, which pre-programmed implementations cannot address.

3) Discussion: We introduce some representative operators that are different from traditional database operators.

Index Scan. For semantic filtering conditions, `LinearScan` is expensive when scanning large datasets. To this end, we implement a vector-based `IndexScan` operator, which can efficiently identify relevant data points with smaller embedding distances to the query and avoid a full scan of the entire data.

Semantic Aggregation. These operators perform aggregations over an unstructured data list, *e.g.,* `SemanticCount` can tally the number of rule violations mentioned across a series of sports match reports.

Semantic Set Operations. These operators handle set operations that involve semantic checks. For example, `SemanticSetUnion` can merge a set of *entity matching* methods from abstracts of academic papers, producing a union of all entity matching methods found either in the original set or within the paper abstracts.

Extensibility for Adding Other Operators. If the provided operators do not meet specific requirements, additional operators can easily be added by defining their logical representations for planning and physical implementations for execution.

V. LOGICAL PLAN GENERATION

In `Unify`, the logical plan is built by recursively applying operators to simplify the query. This section covers operator identification for query reduction (Section V-A), query reduction with selected operators (Section V-B), plan construction during reduction (Section V-C), and the recursive plan generation algorithm (Section V-D).

A. Operator Matching

Overview. Given a query, operator matching aims to efficiently select the suitable operator that can solve a segment of the query. A naive approach is to rely entirely on the LLM to select the operator by organizing all operator descriptions to the LLM within the prompt. However, this is neither efficient nor accurate due to two reasons: (1) high computational cost caused by extensive token usage and (2) high risk of incorrect selections caused by the large number of operators [38]. To address this, we minimize LLM involvement, and use LLM as an auxiliary support when absolutely necessary.

One direct problem for operator matching is accurately interpreting the intent of the natural language query. We address this by also converting the query into a **logical representation** using the LLM, with a few-shot prompt like "Please parse the following question to extract the entities, conditions, attributes, and the return type." As discussed in Section III-A, this involves replacing specific values in the query with semantic placeholders like [Condition], [Entity], and [Attribute]. By simplifying the query to this essential form, we can focus on its logical structure and reasoning, rather than being distracted by specific values, enabling a more precise understanding of its intent.

Our **key insight** is that *an operator is relevant to a query if its logical representation is semantically similar to the query's logical representation*. Therefore, we use semantic similarity to filter out irrelevant operators and focus on those with high potential to match the query, thereby improving both accuracy and efficiency. Based on this idea, `Unify` performs operator matching through a two-stage process. First, it eliminates irrelevant operators by assessing the semantic similarity between the logical representations of the query and operators. Next, the remaining operators are evaluated by the

LLM, which categorizes them according to their capability to address the query, ranging from fully solving, partially solving, to not solving at all.

Semantic Matching. Although the logical representations of queries and operators share a similar form, they are not directly comparable as they are still in natural language. To address this, we transform both into semantic embedding vectors that capture their underlying meanings. The semantic correlation between a query and an operator can then be measured by the distance between their respective embedding vectors. Specifically, embeddings for the logical representations of operators are precomputed and stored. The query's logical representation is generated by prompting the LLM, and its semantic embedding is computed via the embedding model. Operator matching is then performed by computing the embedding distances, allowing us to assess the semantic similarity between the query and available operators. The operators with the highest semantic similarity, along with their corresponding logical representations, are returned as candidates. For example, as shown in Figure 1, the top three operators based on similarity are selected.

Reranking Operators. Although semantic matching identifies a set of candidate operators, not all are guaranteed to solve parts of the query. For instance, an operator selected based on semantic similarity might require unavailable inputs. To refine the operator selection, we maintain a set of available variables, which represent processed data starting from the unstructured dataset. As new variables are generated (*e.g.,* through filtering), their descriptions (generated by prompting the LLM) are added to this set. Only operators whose inputs are all within the available variable set can be considered for further operations.

Specifically, `Unify` uses the LLM to perform a more detailed evaluation of each operator's ability to solve the query. Specialized prompts like "Please check whether the operator can solve any part of the query, if so, output the degree of solution (fully solving, partially solving), otherwise output not solving" are used. The query, available variables, the candidate operator's logical representation, and few-shot examples are provided in the prompt. The LLM categorizes the operators as fully solving, partially solving, or not solving based on its analysis. Candidate operators are then reranked, prioritizing operators more likely to solve the query, using the solving degree as the primary criterion and semantic similarity as the secondary criterion. This produces a sorted list of operators, which is then used to reduce the query and generate the logical plan.

B. Query Reduction

Reduction Process. Once an appropriate operator is selected, `Unify` attempts to reduce the query logically by applying the operator to the matched query segment. Intuitively, this reduction is to replace the matched query segment with the expected operator output. However, due to the inherent flexibility and variability of natural language expressions, strict keyword matching and replacement are insufficient. Therefore, the reduction process in `Unify` is facilitated by an LLM using a predefined few-shot prompt that incorporates placeholders for the query, the matched logical representation of the operator, and the expected output. The structure of the prompt is as follows: *"Given the query [Query] and a matched logical representation [LR] of operator [OP] with expected output [OUTPUT], please rewrite the query with the operator by reducing the matched segment of the logical representation."*

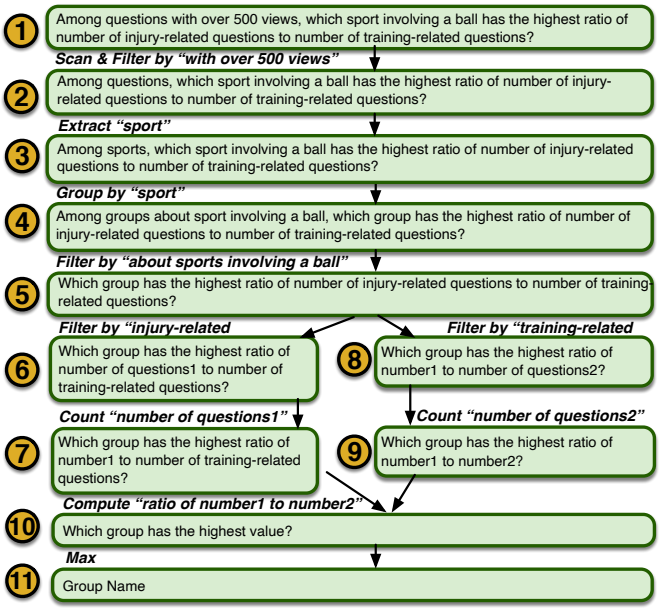


Fig. 2. Query Reduction Process for an Example Query.

Figure 2 illustrates the query reduction process for the example query from Section III, with each reduction step numbered by the reduction order. In the first step, the `Filter` operator is applied to reduce the query by the logical representation "[Entity] satisfy [Condition]" and the expected output "[Entity]". This `Filter` operator is feasible because its required input is the initial unstructured dataset, which is already available. With `Filter`, the original query is simplified by reducing "*Among questions with over 500 views*" to "*Among questions*" and meanwhile generating an intermediate variable "Questions with over 500 views" that can be used in subsequent steps. The reduced query is then recursively processed through the aforementioned semantic parsing, operator matching, and further reduction steps using the same methodology. This recursive process continues until the query is fully solved as will be introduced next.

End of Reduction. The query is gradually simplified until it reaches a final form, which we define as a minimal semantic unit containing only an irreducible element, such as a single entity, numerical value, or Boolean condition. The final state is determined through LLM evaluation using prompts with few-shot examples like: *"Check whether the initial query [Original query] has been fully resolved given the [Variable set description] and the current reduced query [Reduced query]."* This approach ensures that the query is systematically reduced completely, obtaining a logical reasoning path to obtain the

final answer.

C. Plan Construction

Overview. While query decomposition at the logical level (as described in Sections V-A-V-B) breaks down the query, it does not directly produce an execution plan. A straightforward approach is to use the operator sequence applied in decomposition as the plan, but this would impose strict serialization, where each operator must wait for all preceding ones to finish. This is inefficient because many operators can run in parallel without waiting for others to complete. For example, in Figure 2, the `Filter` and `Count` for "training-related questions" and "injury-related questions" can be executed concurrently since they are independent. Therefore, a key question arises: *How can we accurately identify dependencies between operators to optimize their execution order?*

Dependency between Operators. Dependencies between operators O_1 and O_2 are defined by their input-output relationships. Operator O_2 depends on O_1 if it takes O_1's output as an input, or if any of its inputs are derived from an operator dependent on O_1. Thus, identifying dependencies requires analyzing the input and output structure of each operator.

Dependency Check. Given an operator O^* and a sequence of preceding operators $O_1, O_2, \cdots, O_c$, where dependencies among preceding operators are determined, the task is to determine which operators O^* depends on. The dependency is checked in a reverse order, starting from O_c and moving backward to O_1. For each operator, we first check whether it is a prerequisite for an existing prerequisite of O^*; if so, it is directly determined as a prerequisite by the transitivity property of dependency, *i.e.,* if operator A depends on B, and B depends on C, then A also depends on C. If not, we check whether its output is an input for O^*. This check is conducted by instructing LLMs using a prompt in the form of "Check whether the output of O_i is an input for conducting the operator O^*", filled with the outputs of O_i and the required inputs of O^*. After enumerating all preceding operators, all prerequisites of O^* that it depends on are identified.

Plan Construction Algorithm. The plan is constructed as follows. (1) During each step in query decomposition, an operator O is selected; (2) the dependency between O and existing operators in the plan is checked; (3) O is added to the plan by connecting it to its direct prerequisites. This process is repeated iteratively until the full plan is constructed, as shown in Figure 1 and Figure 2.

Example. In the query decomposition from Figure 2, consider the dependency check for the `Filter` operator on "training-related questions." This operator only requires input on questions about specific ball sports with over 500 views, and thus, it has no dependency on the `Filter` for "injury-related questions." Consequently, these two filters can be executed in parallel, as depicted in Figure 1. In contrast, the `Compute` operator depends on the results of both `Count` operators and is connected to them by edges in the final plan. This illustrates that the plan forms a *Directed Acyclic Graph (DAG)*, where prerequisites of an operator may share dependencies.

Algorithm 1 Logical Plan Generation

Input: A query Q, operator index I, top k candidate operators
Output: A logical plan or `None` if no valid plan to generate.

function GENPLAN(Q, $plan$, I, k)
  /* End of Reduction (Section V-B) */
  if SIMPLEQUESTION(Q) **then**
    return $plan$
  /* Operator Matching (Section V-A) */
  $opList \leftarrow$ OPERATORMATCH(Q, I, k)
  for each op in $opList$ **do**
    /* Query Reduction (Section V-B) */
    $Q' \leftarrow$ reduce(Q, op)
    /* Plan Construction (Section V-C) */
    $nextPlan \leftarrow$ ADDOPTOPLAN(Q, Q', op, $plan$)
    $plan' \leftarrow$ GENPLAN(Q', $nextPlan$, I, k)
    if $plan'$ is not `None` **then**
      return $plan'$
  return `None`
$LogicalPlan$ = GENPLAN(Q, $emptyPlan$, I, k)

D. Overall Logical Plan Generation Algorithm

We first introduce the generation of a single plan, followed by error handling and an exploration to create multiple plans.

Generating Single Plan. Algorithm 1 describes the process for generating a logical plan. The algorithm employs a depth-first search (DFS) strategy, where `Unify` recursively applies the operators that best match the query to simplify the query. At each step, the query is reduced by the selected operator, progressing toward a simpler form until it reaches a final state, as described in Section V-B, containing only an irreducible semantic element. During this process, the logical plan is incrementally constructed, with each operator applied being incorporated into the final plan that is stored in the $state$ variable and ensures that the plan maintains an efficient directed acyclic graph (DAG) structure.

Error Handling. If, at any reduction step, no suitable operator is found to simplify the query further, `Unify` backtracks to explore alternative search paths. If after examining all possible search paths the query remains unresolved, `Unify` restores the most complete plan identified so far and prompts the LLM to select one of the following strategies for resolving the remaining query components: (1) Append a `Generate` operator to produce an answer based on collected information via the LLM (fallback to RAG). (2) Instruct the LLM to generate Python code for solving the remaining task (fallback to code generation). The choice of strategy is determined by the LLM based on the complexity of the unresolved task. This error-handling mechanism ensures `Unify` can robustly address ad-hoc queries and incomplete decompositions. Meanwhile, encountered errors are also collected and can be used to build new operators tailored for the specific application scenario.

Generating Multiple Plans. While generating a single plan is straightforward, it is prone to execution failures. In practice, producing multiple candidate plans increases the likelihood of

finding the correct solution. To achieve this, the DFS algorithm is adapted to explore multiple potential plans. Specifically, the search continues after identifying a fully decomposed plan, either until all possible paths are explored or until a predefined number of plans, denoted by a hyperparameter n_c, have been generated. To promote diversity among the candidate plans, the search process may be adjusted to prevent an overly thorough investigation of individual paths. This adjustment is controlled by a second hyperparameter, τ, which ranges from 0 to 1 and dictates the fraction of search paths to be pursued prior to initiating backtracking, *i.e.,* how thoroughly individual search paths are explored. A higher τ allows for a deeper exploration of each path before backtracking, while a lower τ encourages earlier backtracking to explore alternative paths, preventing generating too many results that share the same initial operators (prefixes). When $\tau = 1$, the search becomes exhaustive. Once the target number of candidate plans are generated, the plan generation stops and `Unify` proceeds to optimize them.

VI. Physical Plan Generation

Although the logical plan outlines a feasible method for addressing the input question, it omits the execution details for each step. Nonetheless, the selection of a physical implementation greatly influences query performance, compelling us to choose from several alternatives. To address this, we propose a cost model for selecting the physical plan.

A. Cost Estimation

In `Unify`, the cost of each operator is defined as its *execution time*[1]. As discussed in Section IV, operators can be categorized into two types: *pre-programmed implementations* and *LLM-based implementations*. We estimate the costs for these two categories separately.

LLM-based Implementations. For operators implemented using LLMs, the majority of time is spent on the LLM invocation. As noted in [3], the time cost of LLMs is proportional to the number of output tokens (since input tokens only take a tiny proportion of time cost, about 1% to 5% [3], it can be ignored). Therefore, if we can estimate the number of output tokens, we can accurately predict the cost. The few-shot prompt templates in `Unify` enforce a fixed output format, aiming to keep the number of output tokens constant. However, we observe variations in time consumption and average number of outputs across different LLMs, even for the same operator. This variability makes it impractical to predefine a universal constant for all operators.

Instead, we propose estimating these parameters based on historical execution data. This allows us to compute an average time-per-token constant μ, as well as the average number of output tokens out_{op} for each operator op. The cost can then be estimated as $card \cdot \mu \cdot out_{op}$, where $card$ represents the input cardinality, *i.e.,* the number of elements to be processed.

Pre-programmed Implementations. For pre-programmed implementations, which are implemented using static logic within the code, the computation cost generally depends on the size of the input and the function complexity. By utilizing historical data, we calibrate this function to forecast the computational expense. The projected cost for each operator op can be computed as $f_{op}(card)$, where $f_{op}()$ is the calibrated function and $card$ is the input's cardinality.

Based on above methods, we can estimate the cost for all operator types. Notably, the cost for both LLM-based and pre-programmed operators is closely tied to input cardinality, making accurate cardinality estimation crucial for reliable cost predictions.

B. Semantic Cardinality Estimation

Semantic Cardinality Estimation. In database systems, cardinality estimation (CE) is a well-established problem with a significant impact on cost estimation [16], [18]. However, traditional CE techniques are designed for structured SQL queries and are not directly applicable to semantic queries over unstructured data. Specifically, given an unstructured dataset containing N text records, the task of **semantic cardinality estimation (SCE)** is to predict the result size of a semantic predicate θ without executing the query. Each semantic predicate is expressed as an NL condition, such as *related to sports*, *involves a ball game*. SCE presents unique challenges due to the flexible nature of unstructured data, which lacks a well-defined schema and does not support indexes for CE like the commonly used histograms [28], [31] in relational databases. Moreover, queries over unstructured data are often expressed in natural language with semantic predicates, making them harder to analyze. From Table II, we can observe that many operators have fixed result cardinality (*e.g.,* `Compare`, `Classify`, `Extract`, `Compute`, `Generate`, `TopK` and aggregation operators), or preserve input cardinality (*e.g.,* `OrderBy`). However, remaining operators like `Filter`, `Scan`, `GroupBy`, set operators, and `Join` affect cardinality by selecting data based on certain conditions. Therefore, the focus of SCE is primarily on estimating the cardinality after filtered by the conditions.

Estimation by Sampling. A straightforward approach to SCE is sampling [17], [11], [37], [12]: executing the query on a small subset of sampled data and estimating the proportion of data that satisfies the predicate. However, uniform sampling is inefficient, as it requires large sample sizes for accurate estimations, which is costly when each sample must be evaluated by costly LLMs. Stratified sampling is not directly applicable because unstructured data lacks predefined attributes. Similarly, histograms [28], [31], [12], widely used in relational databases, are infeasible for SCE because unstructured data lacks predefined attribute distributions.

Importance Sampling Optimization. To improve efficiency, we leverage the observation that *data points satisfying the query often have small embedding distances with the query*.

[1]Minimizing total execution time and total execution cost are two different objectives in unstructured data analytics. In this paper, we focus on optimizing the total execution time. It should be noted that the method in this paper is also suitable for optimizing the total cost, just by modifying the cost function accordingly.

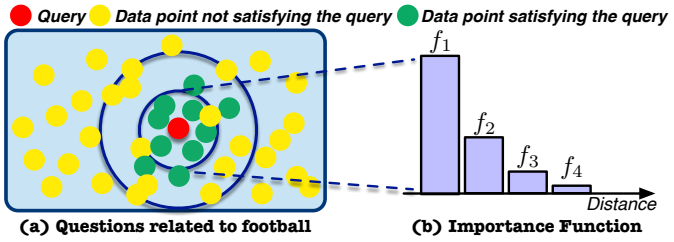


Fig. 3. Illustration of high correlation between embedding distance and probability of satisfying the query predicate.

As illustrated in Figure 3(a), for the example query *"Questions related to football"*, data points closer to the query embedding are more likely to satisfy the query, while the likelihood decreases with increasing distance. Uniform sampling fails in such cases because relevant data points make up only a small portion of the entire dataset and may not be involved in the uniform samples. Therefore, we propose importance sampling, which can focus more on data points closer to the query vector. This method improves the accuracy of cardinality estimation even with a small number of samples.

Importance Function. To implement importance sampling, we define an importance function f, which inputs a distance to the query and outputs the normalized probability of satisfying the query for this distance. Since the right importance function depends on the distribution of vectors, f must be tailored to each dataset individually. Specifically, f in `Unify` is a piecewise scalar function, similar to a histogram, where each piece f_i denotes the *importance* of the i-th group of vectors that are within a specific distance range, with $\sum_i f_i = 1$. During estimation, `Unify` samples data points from groups of different distances proportional to f_i and estimates the result as $\sum_i \frac{n_i(\sum_{x\in S_i}\theta(x))}{n_s \cdot f_i}$ where S_i represents the samples in the i-th group, n_s denotes the total number of samples, n_i denotes the total number of data points in the i-th group and f_i is the importance value for that group. This function f, tailored to the dataset's vector distance distribution, is learned from historical queries. Compared with uniform sampling, a special case for $f_i = \frac{n_i}{N}$, `Unify` samples based on an approximation distribution of satisfied data, thus improving accuracy under the same sample size, *e.g.,* in Figure 3, the data points likely to be omitted by uniform sampling can be captured by `Unify`.

Estimation by Importance Sampling. Given a query with predicate θ, `Unify` estimates its cardinality as follows. (1) Compute embedding vector distances between the query and data; (2) Group data based on the distance ranges specified in the importance function; (3) Samples $n_s f_i$ data points within each group, where n_s denotes the total number of samples; (4) Check whether the samples satisfy θ as $\theta(x)$ using LLMs; (5) Estimate the result as $\sum_i \frac{n_i(\sum_{x\in S_i}\theta(x))}{n_s \cdot f_i}$.

C. Physical Plan Selection

Unlike traditional databases, in `Unify`, the ordering of basic operators such as `Filter` also significantly impacts execution efficiency due to the involvement of LLM computations. The need for extra semantic understanding makes it difficult to rely solely on cost when choosing physical operators.

Operator Order Selection. When a query involves multiple operators, *e.g.,* multiple `Filter` operators, the order in which they are applied affects efficiency. To maximize performance, filters eliminating more data should be applied earlier to reduce the dataset size quickly. For example, the query in Figure 1 contains several filters, and by reordering these filters, we can generate multiple candidate plans. `Unify` evaluates the cost of each plan using the cost model and our SCE method, selecting the plan with the lowest cost for execution.

Physical Operator Selection. As discussed in Section IV-B, a logical operator may have multiple physical implementations, each with varying efficiency. To choose the appropriate physical operator, we apply the cost model from Section VI-A. In cases where multiple physical operators are feasible, we compare their estimated costs and select the most efficient option. When specific operator requirements are present, such as semantic understanding, `Unify` bypasses the cost model and directly selects the operator based on these requirements.

Plan Selection. When multiple physical plans are available, the goal of physical plan selection is to select the most efficient one for execution. Using the cost model from Section VI-A, we estimate the cost of each candidate plan and choose the one with the lowest estimated cost. This ensures `Unify` to select the plan that provides accurate results while minimizing expected execution time.

VII. EXPERIMENTS

Our experiments aim to answer the following key questions. (1) Can `Unify` generate reasonable plans that accurately answer the input queries? (Section VII-B) (2) How efficient are the plans optimized by `Unify`? (Section VII-B) (3) How accurate is the proposed semantic cardinality estimation method? (Section VII-C) (4) How effective are the logical and physical optimization techniques? (Sections VII-D-VII-E)

A. Experimental Settings

Dataset. We evaluate our system using four real-world datasets, widely utilized in prior research, representing a diverse range of content and document lengths. The datasets include informal (Stack Exchange) and formal texts (Wikipedia) with document counts ranging from 1,000 to 5,000.

(1) `Sports` [4]: 3,898 web pages from *Sports Stack Exchange*. For all datasets from Stack Exchange [5], we restrict to questions with at least 3 upvotes to ensure data quality.
(2) `AI` [1]: 5,137 web pages from *AI Stack Exchange*.
(3) `Law` [2]: 2,053 web pages from *Law Stack Exchange*.
(4) `Wiki` [7]: A sample of 1,000 web pages from the English Wikipedia.

Baselines. We compare `Unify` against the following methods:
(1) `RAG` [14]: The basic retrieval-augmented generation (RAG) pipeline that retrieves related document sentences based on embedding similarity and generates answers using the LLM.
(2) `RecurRAG` [36]: An extension of `RAG` that decomposes the query iteratively and retrieves information for decomposed queries.

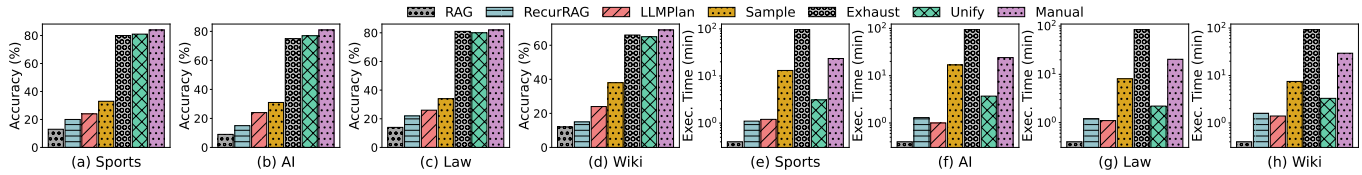


Fig. 4. Performance of different methods on different datasets: Accuracy (a)-(d); Latency (e)-(h).

(3) `LLMPlan`: Instructs the LLM to generate a plan using operator descriptions, then executes the plan by instructing LLM with prompts for each operator.

(4) `Sample`: Uses the LLM to enumerate a proportion of all data (20% in the experiments) with prompts describing the query, and iteratively outputs intermediate results, cumulating the final answer.

(5) `Exhaust`: Exhaustively searches all possible execution plans and selects the best based on LLM feedback, acting as an extreme variant of `Unify`.

(6) `Manual`: Manually creates and executes physical plans by experts. The planning time cost for this method is calculated based on the time spent designing the plan and debugging for execution.

Hyper-parameter Setting. We employ Llama-3.1-70B (quantized to 8-bit floats) for plan generation and Llama-3.1-8B for LLM-based operators since plan generation and operator execution may require different levels of semantic understanding capability. The number of candidate operators for semantic matching is set to $k = 5$, with $n_c = 3$ candidate plans and a plan diversity parameter $\tau = 0.75$. We use Sentence Transformer [30] to embed sentences into embeddings and use the hnswlib [25] library for vector indexing. The top 100 relevant sentences are used in the retrieval step for RAG-based baselines. Execution is parallelized when possible across 4 local Llamas. LLM invocation is batched when possible.

Test Workloads. We generate 100 queries per dataset, derived from 20 templates that are manually designed based on the queries on StackExchange Data Explorer [6]. Each template produces 5 queries by sampling literals from the data. To increase diversity in the natural language expression of the queries, we instruct LLM to generate equivalent variants for each query as the test query and manually verify their equivalence. The ground truths are computed manually.

Evaluation Metrics. We assess result quality using accuracy and measure the end-to-end execution time of answering a query through latency. For semantic cardinality estimation, we measure accuracy by q-error [16], [15], defined as q-error $= \max\{\frac{\text{Estimation}}{\text{GroundTruth}}, \frac{\text{GroundTruth}}{\text{Estimation}}\}$, which compares the estimated cardinality against the actual value. A smaller q-error indicates a more accurate estimation.

Environment. Experiments are conducted on an Ubuntu server with an Intel Xeon 6242R CPU, 6 Nvidia 4090 GPUs, and 2TB RAM.

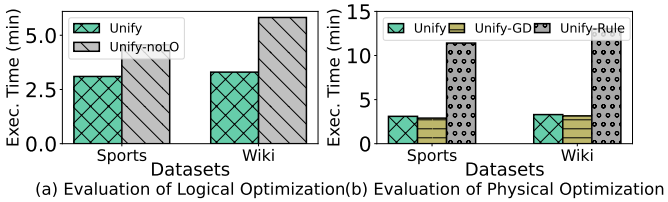
B. Overall Evaluation

1) Comparison of Accuracy: Figure 4(a)-(d) shows the accuracy of different methods across datasets. The results demonstrate that `Unify` significantly outperforms all baselines, with accuracy improvements up to 68%. In addition, `Unify` achieves accuracy similar to exhaustive search methods (`Exhaust`) and manual plan selection (`Manual`). For example, on the `Sports` dataset, `Unify` achieves 81% accuracy, compared to 13% (`RAG`), 20% (`RecurRAG`), 24% (`LLMPlan`), 33% (`Sample`), 80% (`Exhaust`), and 84% (`Manual`). The low accuracy of `RAG` and `RecurRAG` is due to their inability to handle complex queries that require complex reasoning and data aggregation since they only retrieve pieces of information. Similarly, `LLMPlan` struggles due to its reliance on LLMs for both plan generation and execution, which is limited by the limited reasoning abilities of LLMs. `Sample`, while capable of processing data subsets, relies on LLM for complex reasoning and only analyzes the sample subset, thus degrading its accuracy. In contrast, `Unify` uses operator-based query decomposition, which generates logically correct plans and retrieves the necessary information, thus leading to much higher accuracy. Though `Exhaust` achieves high accuracy through an exhaustive search, it is extremely costly and LLM does not always select the optimal plan. Notably, `Unify` obtains an accuracy comparable to `Manual`, which, while accurate, requires extensive time for manual coding and debugging.

2) Comparison of Efficiency: Figure 4(e)-(h) compares end-to-end execution time of different methods. `Unify` answers each query in a few minutes with an average of 0.8 minutes for planning. This execution time is competitive with RAG-based methods and is significantly faster than `Exhaust` and `Manual`. For example, on `Sports`, `Unify` has an average latency of 3.1 minutes, compared to 0.4 minutes (`RAG`), 1.1 minutes (`RecurRAG`), 1.2 minutes (`LLMPlan`), 13.1 minutes (`Sample`), 96.2 minutes (`Exhaust`), and 23.5 minutes (`Manual`). Although `RAG` and `RecurRAG` are faster, their simplistic processing does not consider the aggregation information of all data and limits their accuracy. While `LLMPlan` is also faster, its fully prompt-based planning and execution are incapable of solving complex reasoning required by the queries. `Sample` takes a longer time due to its need to process large data subsets. `Exhaust` and `Manual` are significantly slower since they either exhaustively search all possible plans or involve human efforts. Though involving multi-step planning, optimization, and execution, `Unify` achieves competitive execution time due to optimizations like involving LLMs for planning only

TABLE III
Q-ERRORS OF SEMANTIC CARDINALITY ESTIMATION METHODS.

Sampling Method	Sports				AI			
	50th	95th	99th	Max	50th	95th	99th	max
Uniform	6.95	33.2	94.2	351	6.47	35.0	82.5	409
Stratified	5.82	28.1	79.3	302	6.63	31.5	70.8	345
AIS	7.21	36.8	102.4	387	7.39	48.1	91.2	432
Unify	**1.79**	**6.45**	**12.4**	**22.5**	**1.95**	**5.22**	**9.15**	**25.0**



(a) Evaluation of Logical Optimization(b) Evaluation of Physical Optimization

Fig. 5. Evaluation of: (a) logical optimization; (b) physical optimization.

when necessary, maximizing operator independence degree for parallel execution and appropriate physical operator selection.

In summary, our system, Unify*, achieves significantly higher accuracy compared to state-of-the-art baselines, with improvements ranging from approximately 48% to 68%. It also achieves approximately the same accuracy as manually designed and exhaustively searched plans, while reducing execution latency by up to 40 times compared to* Exhaust *and 10 times compared to* Manual.

C. Semantic Cardinality Estimation Evaluation

We evaluate the effectiveness of our semantic cardinality estimation method (Section VI-B) against other methods on the Sports and AI datasets. Filtering conditions in queries from Section VII-B are used for evaluation, and all methods are constrained to the same number of sampled data points (1% of all data). Baselines include:

(1) Uniform Sampling: uniformly sampling from all data, which is also adopted in PALIMPZEST [22].

(2) Stratified Sampling: grouping data by embedding distance to the query, then sampling proportionally to group size.

(3) Adaptive Importance Sampling (AIS) [19]: dividing the sampling process into multiple iterations. Starting from a uniform distribution, AIS iteratively refines the importance function to improve sampling accuracy. We restrict to two iterations due to the limited number of total samples.

Table III shows that Unify outperforms all baselines. On Sports, the median q-error of Unify is 1.79, compared to 6.95 (Uniform), 5.82 (Stratified), and 7.21 (AIS). Both uniform and stratified sampling fail to account for the distribution of data points that satisfy query conditions. AIS attempts to improve accuracy through iterative refinement but struggles with limited samples. Unify achieves superior accuracy by effectively capturing the distribution of data points that satisfy the query with an accurate importance function.

D. Logical Optimization Evaluation

We evaluate the effectiveness of the proposed logical optimization on the Sports and Wiki datasets by comparing Unify, which employs logical optimization to generate a DAG-structured plan with parallel execution when possible, against Unify − noLo, which executes operators sequentially without optimization. Figure 5(a) shows the latency of the two methods. The results show that logical optimization significantly reduces execution time, with average reductions of 32% and 45% (the sequential plans of some queries offer limited or no optimization potential). This demonstrates that Unify accurately captures operator dependencies, and reducing these dependencies leads to significant efficiency improvement.

E. Physical Optimization Evaluation

To evaluate the effectiveness of the proposed physical optimization methods, including operator ordering and physical operator selection based on the cost model introduced in Section VI, we compare Unify with two baselines: Unify-Rule, which performs no physical optimization and randomly selects physical operators based on semantic requirements, and Unify-GD, which uses ground truth cardinality for cost-based physical optimization. The results, shown in Figure 5(b), demonstrate that Unify significantly reduces execution time compared to Unify-Rule and achieves similar efficiency to Unify-GD. The performance improvement of Unify over Unify-Rule stems from the ability of our optimization techniques to select operator orders that minimize intermediate result sizes early in the execution, thereby reducing the overall computational cost. Additionally, Unify can choose more efficient physical operators when applicable, such as IndexScan. The comparable efficiency between Unify and Unify-GD is due to the accuracy of Unify in estimating semantic cardinality, which allows it to accurately predict plan costs and select the most efficient execution plan.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we proposed Unify, a novel system designed to automate unstructured data analytics by utilizing LLMs. To address the limitations of traditional analytics systems that rely on predefined schemas, we presented an algorithm for generating logical plans that can tackle complex queries with accurate logical reasoning. We designed effective methods for optimizing physical plans, turning logical plans into efficient executions based on an innovative cost model and semantic cardinality estimation. Experiments demonstrated that Unify not only significantly improved accuracy but also enhanced query processing speed for analyzing unstructured data.

ACKNOWLEDGMENT

Guoliang Li is the corresponding author. This work was supported by National Key R&D Program of China (2023YFB4503600), Shenzhen Project (CJGJZD20230724093403007), NSF of China (62232009), Zhongguancun Lab, Huawei, and Beijing National Research Center for Information Science and Technology (BNRist).

REFERENCES

[1] Ai stackexchange data. https://archive.org/download/stackexchange/ai.stackexchange.com.7z.

[2] Law stackexchange data. https://archive.org/download/stackexchange/law.stackexchange.com.7z.

[3] Openai latency optimization. https://platform.openai.com/docs/guides/latency-optimization.

[4] Sports stackexchange data. https://archive.org/download/stackexchange/sports.stackexchange.com.7z.

[5] Stackexchange data archive. https://archive.org/download/stackexchange.

[6] Stackexchange data explorer. https://data.stackexchange.com/stackoverflow/queries.

[7] Wiki data archive. https://dumps.wikimedia.org.

[8] E. Anderson, J. Fritz, A. Lee, and et al. The design of an llm-powered unstructured analytics system. *arXiv:2409.00847*, 2024.

[9] S. Arora, B. Yang, S. Eyuboglu, A. Narayan, A. Hojel, I. Trummer, and C. Ré. Language models enable simple systems for generating structured views of heterogeneous data lakes. *VLDB*, 17(2):92–105, 2023.

[10] A. Asai, Z. Wu, Y. Wang, A. Sil, and et al. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *ICLR 2024*, 2024.

[11] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *SIGMOD*, pages 263–274. ACM Press, 1999.

[12] G. Cormode, M. N. Garofalakis, and et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases*, 4(1-3):1–294, 2012.

[13] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.*, 17(5):1132–1145, 2024.

[14] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, Q. Guo, M. Wang, and H. Wang. Retrieval-augmented generation for large language models: A survey. *CoRR*, abs/2312.10997, 2023.

[15] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*. www.cidrdb.org, 2019.

[16] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *VLDB*, 2015.

[17] V. Leis, B. Radke, A. Gubichev, and et al. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[18] V. Leis, B. Radke, A. Gubichev, and et al. Query optic mization through the looking glass, and what we found running the join order benchmark. *VLDB J.*, 27(5):643–668, 2018.

[19] G. P. Lepage. Adaptive multidimensional integration: vegas enhanced. *J. Comput. Phys.*, 439:110386, 2021.

[20] B. Li, Y. Luo, C. Chai, G. Li, and N. Tang. The dawn of natural language to sql: Are we fully ready? *VLDB*, 17(11):3318–3331, 2024.

[21] Y. Lin, M. Hulsebos, R. Ma, and et al. Towards accurate and efficient document analytics with large language models. *CoRR*, 2024.

[22] C. Liu, M. Russo, M. J. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. J. Franklin, T. Kraska, S. Madden, and G. Vitagliano. A declarative system for optimizing AI workloads. *CoRR*, abs/2405.14696, 2024.

[23] X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Luo, Y. Zhang, J. Fan, G. Li, and N. Tang. A survey of nl2sql with large language models: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109*, 2024.

[24] S. Madden, M. J. Cafarella, M. J. Franklin, and T. Kraska. Databases unbound: Querying all of the world's bytes with AI. *Proc. VLDB Endow.*, 17(12):4546–4554, 2024.

[25] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

[26] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[27] L. Patel, S. Jha, C. Guestrin, and M. Zaharia. LOTUS: enabling semantic queries with llms over tables of unstructured and structured data. *CoRR*, abs/2407.11418, 2024.

[28] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, pages 294–305. ACM Press, 1996.

[29] M. Pourreza and D. Rafiei. DIN-SQL: decomposed in-context learning of text-to-sql with self-correction. In *Neurips*, 2023.

[30] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *EMNLP*, pages 3980–3990. Association for Computational Linguistics, 2019.

[31] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34. ACM, 1979.

[32] S. Shankar, T. Chambers, T. Shah, A. G. Parameswaran, and E. Wu. Docetl: Agentic query rewriting and evaluation for complex document processing. *arXiv preprint arXiv:2410.12189*, 2024.

[33] H. Touvron, T. Lavril, G. Izacard, X. Martinet, and et al. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[34] I. Trummer. Generating succinct descriptions of database schemata for cost-efficient prompting of large language models. *VLDB*, 2024.

[35] M. Urban and C. Binnig. CAESURA: language models as multi-modal query planners. In *CIDR*, 2024.

[36] S. Yao, J. Zhao, D. Yu, N. Du, and et al. React: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

[37] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *SIGMOD*, pages 1525–1539. ACM, 2018.

[38] C. Zheng, H. Zhou, F. Meng, J. Zhou, and M. Huang. Large language models are not robust multiple choice selectors. In *ICLR*, 2024.