AOP: Automated and Interactive LLM Pipeline Orchestration for Answering Complex Queries

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

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

ABSTRACT

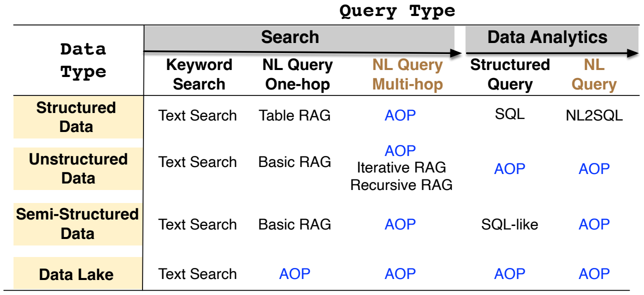
Current data lakes are limited to basic put/get functions on unstructured data and analytical queries on structured data. They fall short in handling complex queries that require multi-hop semantic retrieval and linking, multi-step logical reasoning, and multi-stage semantic analytics across unstructured, semi-structured, and structured data in data lakes. The introduction of large language models (LLMs) has significantly transformed the landscape of traditional data search and analytics across different fields due to their semantic comprehension and reasoning skills. Utilizing LLMs opens up new opportunities to efficiently handle these complex queries for data search and analytics, spanning structured, semi-structured, and unstructured data types in data lakes. However, LLMs struggle with complex queries that require complex task decomposition, pipeline orchestration, pipeline optimization, interactive execution, and self-reflection.

In this work, we propose AOP, the first systematic system for automated pipeline orchestration in LLMs for answering complex queries on data lakes. AOP pre-defines standard semantic operators crucial for building execution workflows, such as semantic retrieval, filtering, aggregation, and validation. Then given an online query, AOP extracts relevant operators and uses these operators to automatically and interactively compose optimized pipelines with the assistance of LLMs. This enables AOP to adaptively and accurately address diverse and complex queries on data lakes. To further improve efficiency, we introduce query optimization techniques, including prefetching and parallel execution, to enhance overall efficiency without sacrificing accuracy. Through extensive experiments on real-world datasets, we demonstrate that AOP significantly improves the accuracy for answering complex queries. For instance, on a challenging test set, AOP increases answer accuracy by 45%.

1 INTRODUCTION

Existing data lakes primarily offer simple put/get operations for unstructured data and support analytical queries for structured data. However, they are not equipped to process complex queries that demand advanced semantic retrieval and linkage, intricate logical reasoning, and sophisticated semantic analytics across the spectrum of unstructured, semi-structured, and structured data in data lakes.

The advent of large language models (LLMs) has dramatically revolutionized traditional data search and analytics across various domains [26, 32] due to their strong semantic understanding and reasoning capabilities. By harnessing LLMs, we have opportunities to effectively process semantic queries for data search and analytics across structured, semi-structured and unstructured data, as illustrated in Figure 1. Taken together, LLMs bring the potential to explore and interrogate the vast, diverse heterogeneous data in data lakes, which turns traditional static storage-oriented data lakes into dynamic intelligent analytical data lakes, thereby unlocking the full value of data lakes.



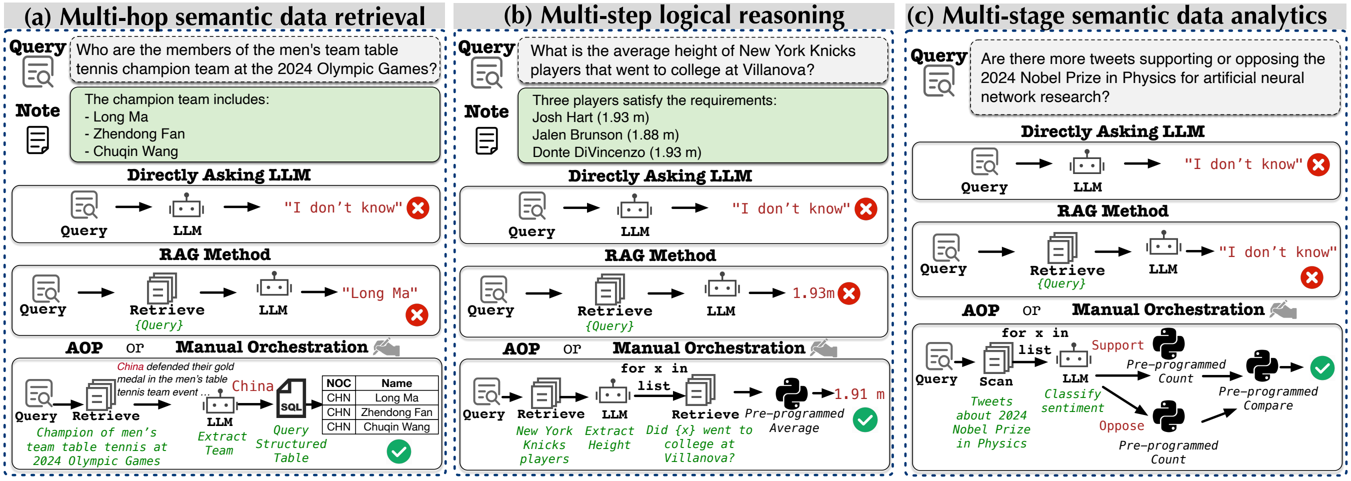
Query Type

Data Type	Search			Data Analytics	
	Keyword Search	NL Query One-hop	NL Query Multi-hop	Structured Query	NL Query
Structured Data	Text Search	Table RAG	AOP	SQL	NL2SQL
Unstructured Data	Text Search	Basic RAG	AOP Iterative RAG Recursive RAG	AOP	AOP
Semi-Structured Data	Text Search	Basic RAG	AOP	SQL-like	AOP
Data Lake	Text Search	AOP	AOP	AOP	AOP

Figure 1: Roadmap of data search and data analytics methods.

Complex Queries. However, LLMs often struggle with processing *complex queries that necessitate multi-hop semantic data retrieval and linking, multi-step logical reasoning, and multi-stage semantic data analytics across heterogeneous data formats*, such as structured tables and unstructured documents in data lakes. For such complex queries, directly asking LLMs or using some static human-craft execution pipelines, *e.g.,* retrieval-augmented generation (RAG) methods, cannot get correct answers, because they demand capabilities in understanding, reasoning, orchestrating complex processes, and reflection to produce accurate answers.

We provide several examples to demonstrate the types of complex queries that data lakes are currently unable to process as illustrated in Figure 2. First, consider a natural language (NL) query that requires multi-hop retrieval and linking across unstructured data and structured data. Given a query: "Who are the members of the men's team table tennis champion team at the 2024 Olympic Games", the winning team needs to be identified from news in text documents while the player list should be obtained from structured tables. Second, consider a query that necessitates multi-step reasoning: "What is the average height of New York Knicks players that went to college at Villanova?". As illustrated in Figure 2, directly asking this query to an LLM receives an answer of "I don't know" due to the absence of relevant information in the LLM knowledge. Even when using the Retrieval-Augmented Generation (RAG) method to obtain external knowledge, the query is still not answered correctly. This is because accurately answering the query requires not only precise retrieval of relevant information (*i.e.,* identifying all the New York Knicks players who went to college at Villanova), but also



Who are the members of the men's team table tennis champion team at the 2024 Olympic Games?

The champion team includes:
- Long Ma
- Zhendong Fan
- Chuqin Wang

NOC	Name
CHN	Long Ma
CHN	Zhendong Fan
CHN	Chuqin Wang

What is the average height of New York Knicks players that went to college at Villanova?

Three players satisfy the requirements:
Josh Hart (1.93 m)
Jalen Brunson (1.88 m)
Donte DiVincenzo (1.93 m)

Are there more tweets supporting or opposing the 2024 Nobel Prize in Physics for artificial neural network research?

Figure 2: Complex query examples: (a) multi-hop data retrieval; (b) multi-step reasoning; (c) multi-stage semantic data analytics.

performing correct logical analytics (extracting the heights of the players and summing up them to compute the average height). Third, consider a query that involves multi-stage semantic analytics: "Are there more tweets supporting or opposing the 2024 Nobel Prize in Physics for artificial neural network research?". This requires first identifying relevant tweets about the 2024 Nobel Prize in Physics via scan, then accurately conducting semantic analytics to classify each as supportive or opposing to count the number of tweets for each category respectively, and lastly comparing the results to compute the final answer. For such multi-stage data analytics, failure at any stage can result in an incorrect or incomplete answer. This requires multi-stage reasoning and reflection to get a correct answer.

In this paper, *we explore the challenges of handling complex queries posed in natural language within data lakes and aim to develop an intelligent data lake ecosystem.* We address complex queries that, unlike traditional SQL queries with their rigid structure and limitation to structured data, necessitate semantic operator execution (*e.g.,* semantic filter for identifying all the New York Knicks players who went to college at Villanova, and semantic aggregations of their heights), semantic pipeline (plan) generation with multi-step decomposition, physical operator implementation with multi-hop retrieval across multiple stages, and interactive and parallel physical pipeline execution from diverse data formats and sources.

Traditionally, such complex queries are addressed by manually orchestrated execution pipelines involving LLMs, as demonstrated in Figure 2. These pipelines break down the query into sub-tasks and combine pre-programmed steps, retrieval steps, and prompt-based subtasks to obtain accurate answers [1, 4]. However, manual pipeline orchestration has notable limitations:

Human Costs (P1): Manual orchestration incurs significant human costs, as an effective pipeline depends heavily on user expertise. In addition, effective pipelines are typically complex, *e.g.,* comprising hundreds of steps [2], which further increases human cost since more time and expertise are needed.

Static Pipeline (P2): Intermediate operations in LLM pipelines may fail to obtain the expected outcomes. However, manually written pipelines cannot dynamically make adjustments to these failures. For example, if a retrieval step yields irrelevant results, subsequent steps should be adjusted accordingly. Otherwise, the final answer based on unrelated information is likely to be incorrect.

Limited Functionality (P3): Human-designed pipelines are tailored to specific queries and do not work for our complex queries, which boast an almost limitless range of expression. As a result, many queries are not covered by existing pipelines. For these queries, only relatively basic pipelines can be used, resulting in low accuracy.

Central Problem. Therefore, the central problem we aim to address is *whether we can automate pipeline orchestration for processing complex queries on data lakes*. We aim to design a framework that not only automates pipeline orchestration but also enables dynamic, interactive execution with intermediate adjustment and self-reflection based on real-time results.

Key Idea. In this work, we propose AOP, the first framework that automates the orchestration and optimization of execution pipelines for complex queries over data lakes. *The key idea is that human-crafted pipelines are essentially well-constructed assemblies of standard semantic operators. We can predefine these standard semantic operators, and offer programmed and LLM-driven implementations for these operators. Then given an online complex query, we harness LLMs to automatically identify the necessary semantic operators, orchestrate semantic pipelines with multi-step logical reasoning, optimize the pipeline with multiple objectives, and iteratively execute the pipeline with self-reflection.*

Challenges. Automating pipeline orchestration and optimization introduces some challenges.

Flexible Semantic Operator Definition and Extraction (C1). Identifying standard common semantic operators within LLM-driven pipelines is fundamental but challenging. The operator set must be flexible and adaptable to support diverse query scenarios.

Automatic Semantic Pipeline Orchestration (C2). Orchestrating a pipeline automatically is challenging due to the numerous possible compositions of operators. Optimizing these pipelines for efficiency is also difficult since predicting the performance of each composition before execution is hard given the diverse data sources and complex reasoning paths.

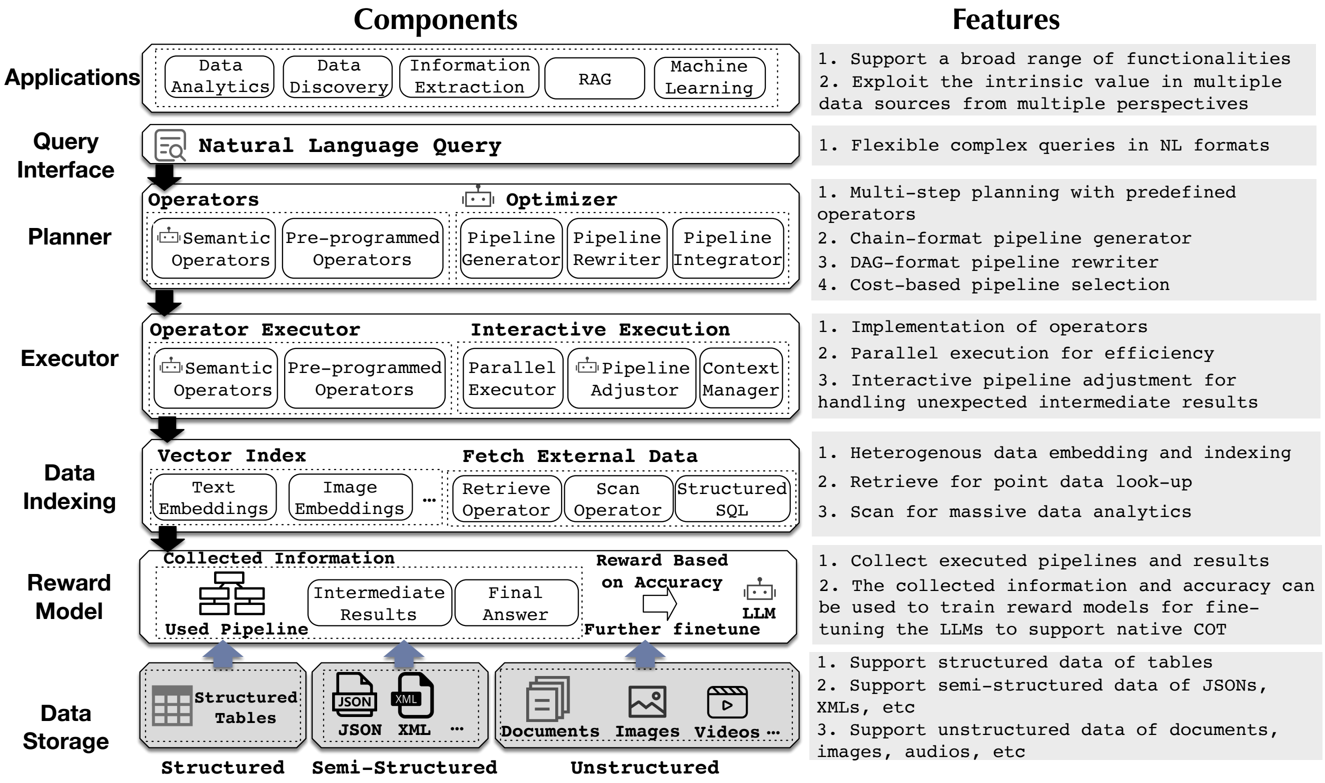

Components
Features
Applications
Data Analytics
Data Discovery
Information Extraction
RAG
Machine Learning
1. Support a broad range of functionalities
2. Exploit the intrinsic value in multiple data sources from multiple perspectives
Query Interface
Natural Language Query
1. Flexible complex queries in NL formats
Planner
Operators
Semantic Operators
Pre-programmed Operators
Optimizer
Pipeline Generator
Pipeline Rewriter
Pipeline Integrator
1. Multi-step planning with predefined operators
2. Chain-format pipeline generator
3. DAG-format pipeline rewriter
4. Cost-based pipeline selection
Executor
Operator Executor
Semantic Operators
Pre-programmed Operators
Interactive Execution
Parallel Executor
Pipeline Adjustor
Context Manager
1. Implementation of operators
2. Parallel execution for efficiency
3. Interactive pipeline adjustment for handling unexpected intermediate results
Data Indexing
Vector Index
Text Embeddings
Image Embeddings
...
Fetch External Data
Retrieve Operator
Scan Operator
Structured SQL
1. Heterogenous data embedding and indexing
2. Retrieve for point data look-up
3. Scan for massive data analytics
Reward Model
Collected Information
Used Pipeline
Intermediate Results
Final Answer
Reward Based on Accuracy
LLM
Further finetune
1. Collect executed pipelines and results
2. The collected information and accuracy can be used to train reward models for fine-tuning the LLMs to support native COT
Data Storage
Structured Tables
JSON
XML
...
Documents
Images
Videos ...
Structured
Semi-Structured
Unstructured
1. Support structured data of tables
2. Support semi-structured data of JSONs, XMLs, etc
3. Support unstructured data of documents, images, audios, etc

Figure 3: Architecture overview of AOP.

Streamlined Pipeline Execution with Self-assessment (C3). Achieving efficient end-to-end execution while preserving high accuracy is challenging. High accuracy often requires composing complex, multi-step pipelines that demand significant planning and execution time with self-reflection and online adjustment.

Contributions. In summary, we make the following contributions:

(1) We propose AOP, the first framework for Automated Orchestration and optimization of LLM Pipelines over data lakes, designed to handle complex queries with high flexibility. AOP supports interactive pipeline adjustments during execution, allows for addition/deletion/update of custom operators, and handles heterogeneous data.

(2) We identify and implement key standard semantic operators essential for building effective LLM pipelines and implement them within AOP, supporting a wide range of query scenarios.

(3) We design an automatic pipeline orchestration method to generate an effective pipeline for a given NL complex query.

(4) We propose pipeline optimization techniques, such as prefetching, parallel execution, and cost-based pipeline selection, to enhance the overall efficiency of AOP without sacrificing accuracy.

(5) We have implemented AOP and validated AOP on real-world datasets, demonstrating that AOP can automatically generate accurate pipelines for complex tasks in scenarios like unstructured data analytics. For example, AOP achieves up to a 45% improvement in answer accuracy on a challenging test set, compared with directly asking LLMs with the query.

2 SYSTEM ARCHITECTURE

The AOP system is designed to address complex queries over structured, semi-structured, and unstructured data in data lakes, as shown in Figure 3. AOP is equipped with predefined standard semantic operators to support a wide range of functionalities on data lakes, including semantic data discovery and linking, multi-hop retrieval-augmented generation (RAG), multi-stage semantic data analytics, and machine learning. Given an online natural language (NL) query, AOP identifies the relevant semantic operators involved in the query, orchestrates interactive and efficient pipelines of processing the query, and executes the pipelines with self-reflection.

Query Interface. AOP initiates processing through a natural language query interface, which allows users to input complex, unstructured queries, supporting a broader range of queries with deeper semantic analytics than structured query languages like SQL.

Operators. AOP identifies and implements a collection of standard operators commonly used in data analytics, *e.g.,* `Scan`, `Filter`, `GroupBy` and `Aggregate`. Operators are divided into two categories based on their implementations: pre-programmed operators and semantic operators (executed by LLMs with prompts), each executing a specific task. Together, the operators can form accurate execution pipelines to address complex queries.

Optimizer. Based on the input natural language query, the optimizer automatically generates an optimized execution pipeline.

Pipeline Generator. This component generates multiple chain-format execution pipelines based on predefined operators and the input NL query. Leveraging the randomness in the output generation of LLMs, multiple candidate pipelines with diverse reasoning paths can be generated.

Pipeline Rewriter. The rewriter refines each chain-format execution pipeline into a Directed Acyclic Graph (DAG) structure to enable parallel execution of independent operators and enhance efficiency.

Pipeline Integrator. This component consolidates multiple execution pipelines into a single one to integrate different reasoning paths to obtain the final answer.

Cost-Based Pipeline Optimization. When choosing among multiple potential pipelines, the optimizer picks the most cost-effective option by estimating the computational cost for each operator. This cost estimation process greatly differs from traditional database methods, given the lack of pre-established attributes and predicates to guide the pipeline assessment.

Pipeline Executor. The executor judiciously runs the selected optimized pipeline to generate the final answer.

Operator Executor. Each operator is executed according to its defined implementation, either programmed functions or LLM-based implementations.

Parallel Execution. The operators can be executed in parallel where possible, with techniques like prefetching used to further improve efficiency.

Interactive Execution. During execution, `AOP` can dynamically adjust the pipeline based on intermediate results, allowing it to adjust the pipeline to handle unexpected outcomes, such as retrieval failures, or unexpected intermediate results.

Context Management. To manage intermediate results efficiently within LLM context length constraints, `AOP` employs a context manager that uses the `Summarize` operator, condensing information to support large-scale, multi-step processes, while also reducing the LLM overhead.

Indexing. `AOP` computes embeddings for various data types and builds vector indexes on them for efficient retrieval and scanning. `AOP` also employs SQL and other access tools (e.g., dataframe) for structured data in tables.

Reward Model. To effectively adjust the pipelines at runtime, we design a reward model to provide a self-reflection for pipeline execution. The reasoning process of LLMs can be modeled as a Markov Decision Process (MDP), where each operator functions as an action that the LLM policy can select at each state to guide reasoning steps toward the solution. Effective training LLM policies to have high logical reasoning abilities relies on an accurate Process Reward Model (PRM) [17, 22], which assigns rewards based on the relevance and impact of each operator in its given state. In the `AOP` framework, the sequence of chosen pipelines for each query, along with intermediate and final outcomes, is recorded to provide reward signals at each step. These records serve as training data for the PRM, which then fine-tunes the LLM to reinforce its native chain-of-thought capabilities. As a result, `AOP` can gain improved effectiveness in answering questions by iteratively refining the LLM's reasoning ability to generate and optimize efficient, accurate pipelines.

Data Storage. `AOP` manages diverse data formats in data lakes, including structured data like tables and CSV files, semi-structured data like JSON/XML files, and unstructured content like documents, images, and videos.

3 SEMANTIC OPERATORS

In this section, we first introduce the standard semantic operators (Section 3.1), then discuss how to execute the operators (Section 3.2) and finally present how to flexibly add other operators to the operator set (Section 3.3).

3.1 Definition of Semantic Operators

We have elaborately designed 22 semantic operators to answer complex queries, and other operators can be easily added into our system. Similar to relational database operators, the operators can be categorized into *logical* and *physical* levels.

Retrieve. The `Retrieve` operator enhances LLMs by fetching up-to-date or specific information not contained within the model. It searches external databases, documents, or knowledge bases on data lakes to find relevant information based on a query. `Retrieve` has various physical implementations, such as searching documents over text embeddings or querying a knowledge graph.

Example

In Figure 2, the `Retrieve` operator in the RAG pipeline fetches documents related to the query to enhance response accuracy.

Scan. The `Scan` operator loads and enumerates all entries within a data source for processing, often applying filtering or initial transformations as it scans. `Scan` is crucial for efficiently handling large data lakes, as it serves as the initial step to retrieve raw data before any further processing. Physical implementations include both linear scans over files and indexed scans that speed up the process when an index is available. Different from `Retrieve` that only extracts several pieces of data, `Scan` goes through a large data collection.

Example

In a pipeline querying user profiles, the `Scan` operator is employed to load all records containing user data, allowing subsequent filtering of the query based on specified criteria.

Filter. The `Filter` operator removes information that does not meet specified criteria from retrieved or generated data. It is similar to the selection (σ) operator in relational databases but can also operate on unstructured text data through the assistance of LLMs.

Example

Consider a query that wants to analyze the post-2020 information. It is necessary to filter out the pre-2020 parts in the retrieved results because they would mislead the LLM response.

OrderBy. The `OrderBy` operator sorts retrieved documents or information based on given criteria, which is similar to `Order By` in SQL, but with the additional capability of semantic ordering.

Example

For the example query in Figure 2(c), the tweets can be sorted by their sentiment degree. In this way, tweets with a clear sentiment tendency can be determined directly, while ambiguous tweets can also be further examined to obtain more accurate results.

Summarize. The `Summarize` operator condenses longer pieces of text into shorter, more manageable summaries while retaining key

Table 1: Summary of Semantic Operators

Operator	Description
`Retrieve`	Fetch relevant data from external sources to answer a query, enhancing LLM responses with up-to-date or specific details.
`Scan`	Load documents or tables and enumerate their elements.
`Filter`	Remove non-relevant information based on specific criteria, similar to the selection operator in SQL.
`OrderBy`	Order retrieved information based on criteria, similar to SQL's `ORDER BY` operator.
`Summarize`	Condense lengthy text into shorter summaries to optimize context consumption and readability.
`Generate`	Produce coherent text based on input, often used to generate final responses.
`Refine`	Adjust input text to enhance clarity or precision, beneficial for subsequent processing.
`Classify`	Categorize or label entities, using either the LLM or external ML models for classification.
`Translate`	Convert text between languages, implemented using the LLM or external tools.
`Transform`	Transform data from one format into another, *e.g.,* transform a structured table into text descriptions.
`Evaluate`	Assess the quality or relevance of information based on criteria, often through log probabilities or binary outputs from the LLM.
`Explain`	Provide explanations or justifications for decisions, enabling model reflection.
`Integrate`	Combine information from multiple sources into a cohesive response.
`Conceptualize`	Identify key concepts in text, simplifying complex queries by pinpointing main ideas.
`Extract`	Isolate specific information relevant to the query, similar to the projection operator in SQL.
`Plan`	Automatically orchestrate a pipeline of operators for executing complex queries.
`Link`	Identity and construct linking across data related to the query, *e.g.,* obtaining relevant tables and documents in the data lake.
`Set`	Set operations over sets of data, *e.g.,* `Union`, `Intersection`, and `Complementary`.
`Validate`	Verify the accuracy of generated information by searching with the answer itself within external sources.
`GroupBy`	Organize data into subgroups based on specific attributes or conditions, allowing the computation of summary statistics for each group.
`Compare`	Assess two input values based on a specified operand and return the value that satisfies the comparison criteria.
`Aggregate`	Compute aggregation results, *e.g.,* sum, average, max, from input data.

information. This reduces context consumption and makes the information easier to process and understand.

Example

When approaching the LLM context length limit, it is important and necessary to summarize the previous content to continue the pipeline execution.

Generate. The `Generate` operator produces coherent text based on the input, such as retrieving relevant information via RAG and generating the final response by LLMs. It also serves as a flexible fallback when other operators cannot meet the requirements.

Example

Pipelines in Figure 2 can use `Generate` to generate the final reply.

Refine. The `Refine` operator improves or adjusts the input text to better meet requirements or improve coherence. It can rewrite vague or poorly structured queries to be more precise and less ambiguous, benefiting subsequent operators.

Example

Queries that contain text irrelevant to the question can be stripped away with the `Refine` operator so that the subsequent pipeline can focus on answering the question without being misled.

Classify. The `Classify` operator assigns categories or labels to the input text or entities within it. It performs simple classification tasks described by natural language. Its physical implementation can be either the LLM itself or an external specialized trained ML model.

Example

The example query in Figure 2(c) involves classifying the sentiment of the tweets.

Translate. The `Translate` operator converts text from one language to another. Its physical implementations include both the LLM and pre-programmed implementations.

Example

Consider a query entered in English, but specified to output the results in French. The results here need to be translated.

Evaluate. The `Evaluate` operator assesses the quality of different inputs based on specified criteria. Its physical implementations include either leveraging the output of the LLM or utilizing log probabilities inside the LLM for options `True` and `False`.

Example

To evaluate whether the `Retrieve` results are relevant to the original query, the retrieved paragraphs can be input to the `Evaluate` operator for evaluation. The relevance can be judged by instructing the LLM with certain prompts. Optionally, the relevance can also be judged by enforcing the LLM to output from `True` and `False`, and comparing the probability of these two tokens.

Explain. The `Explain` operator provides explanations or reasoning for a response or decision made by the model. It enables reflection [8], allowing the model to identify and correct uncertain results.

Example

Consider a query where the LLM gets different answers through different reasoning. In order to get a final answer, each reasoning needs to explain why it gets the answer.

Integrate. The `Integrate` operator receives information from multiple sources, judges the correctness of them, and combines them into a cohesive response.

Example

In the above example for `Explain`, to get the final answer, these candidate answers and their explanations need to be aggregated using the `Integrate` operator.

Conceptualize. The `Conceptualize` operator extracts and identifies key concepts described in the text, simplifying complex information by pinpointing the main idea. Its physical implementations include directly querying the LLM or retrieving from external knowledge.

Example

Converting the concept in the query "Find the occupations of all the wives of the presidents of the United States" into "first lady" simplifies subsequent process of retrieving relevant information.

Extract. The `Extract` operator identifies and pulls out specific pieces of information from data, isolating crucial details for answering the query. It typically follows the `Retrieve` operator to extract important information related to the query and is similar to the Projection (π) operator in SQL.

Example

As shown in Figure 2(b), after retrieving "New York Knicks players", player heights are extracted before averaging.

Plan. The `Plan` operator automatically orchestrates a pipeline composed of the operators for the input query. Its detailed process is introduced in Section 4.1.

Example

Given a query, such as the one in Figure 2(a), the `Plan` operator can be called to orchestrate the pipeline for execution.

Link. To process heterogeneous data collections, such as data lakes, it is essential to identify and link data that contains relevant information for the current query. The `Link` operator handles this by identifying entire related data items (*i.e.,* complete tables and text files) and their relationships. For instance, schema linking [16, 20] is a common technique used to align natural language queries with structured table content in NL2SQL, but the `Link` operator also extends to unstructured sources by linking relevant documents and identifying cross-format relationships for integrated analytics.

Example

As illustrated in Figure 2(a), accurately answering the example query requires information from both an unstructured document that details the winning team and a structured table that lists team members. If these data items are not annotated, `Link` identifies and connects them within the data lake, enabling a coherent answer to the query.

Transform. The `Transform` operator supports transforming data from one type into another, which is frequently used for analytics over heterogeneous data.

Example

The example query in Figure 2(a) needs to transform the structured table results into text descriptions if it requires text output.

Set. The `Set` operator conducts set computations on data, *e.g.,* `Union`, `Intersection`, and `Complementary`. It enables the combination or differentiation of data sets to meet specific query requirements.

Example

For a query seeking combined data on people who are either New York residents or have New York employment history, the `Set` would union the two sets: New York residents and New York employees.

Validate. The `Validate` operator checks the accuracy and reliability of generated or retrieved information. Unlike `Retrieve`, which searches using the query, `Validate` searches using the generated answer, ensuring that the search item is within the same space as the data (documents) being searched.

Example

After obtaining the final answer through the retrieved relevant information and logical reasoning, we can execute `Validate`, and use the generated answer to search the external knowledge supporting or opposing the answer.

GroupBy. The `GroupBy` operator partitions data into subgroups according to specified attributes, such as grouping by age or location. `GroupBy` allows further operators to compute summary statistics or apply other transformations specific to each group, similar to the `GROUP BY` operation in SQL.

Example

In a sentiment analysis query that examines reviews of different products, the `GroupBy` operator can cluster reviews by product categories, enabling an aggregate analysis of sentiment trends for each product.

Compare. The `Compare` operator evaluates two input values using a specified operand (such as <, >, or =) and returns a boolean result indicating whether the comparison criteria are met. It is commonly used for evaluating conditional expressions, applying thresholds in analytics, or determining the semantic relationship between inputs.

Example

When comparing the count of two categories, `Compare` can perform numerical comparisons based on the condition. Besides, in analyzing text content, it is common to compare different natural language expressions to determine whether they are describing the same thing.

Aggregate. The `Aggregate` operator computes summary statistics such as `Count`, `Sum Min`, `Max`, `Median`, and `Percentile` over the data. These statistics are useful for generating insights and analyzing distributions within datasets.

Example

For the example query in Figure 2, the `Aggregate` operator takes a list of heights as input and computes their average through numerical computations.

3.2 Operator Execution

The execution of each operator depends on its physical implementation, which can be classified into two types: *prompt-based* (LLM-based) and *pre-programmed*. Similar to relational databases, each operator can have multiple physical implementations.

Prompt-based LLM Execution. Prompt-based operators involve instructing the LLM to complete specific tasks via carefully designed prompts. For basic operators such as `Summarize` and `Extract`, execution consists of inputting the operator and its description as a prompt, guiding the LLM to perform the operation and return the result. We use the following prompt template: *The next operator is {Operator}. This operator is to {Operator Description}. Please execute it following the instructions and output the results.* Some operators, such as semantic `Filter` or `GroupBy`, require data retrieval before instructing the LLM to execute the operation. For example, the query shown in Figure 2(c) first retrieves tweet data from the data lake and then applies a semantic filter to retain only tweets about the 2024 Nobel Prize in Physics, guided by LLM-based filtering instructions.

Pre-programmed Execution. Pre-programmed operators, which do not involve LLMs, are executed directly using pre-defined programs. For instance, the `Retrieve` operator might perform a nearest neighbor search on document embeddings or use BM25 for keyword searches. Similarly, the `Filter` operator may apply programmed rules—such as filtering by attribute values. In all cases, the selected physical operator follows a specific execution workflow defined in its pre-programmed implementation.

3.3 Support of Adding New Operators

In practical applications, the predefined operators introduced in Section 3.1 may not encompass all use cases, highlighting the need for incorporating custom operators. The addition of custom operators into our framework is easy by its inherent flexibility. Specifically, the pipeline orchestration of `AOP` only leverages the explanation of each operator (Section 4.1). Therefore, adding a custom operator only requires providing its explanation and implementation. For prompt-based operators, minimal coding is needed, as it primarily involves writing the operator description and prompt in natural language.

4 LLM PIPELINE ORCHESTRATION FRAMEWORK

To answer an input query, `AOP` first takes the query and detailed operator descriptions (*e.g.,* descriptions in Table 1) as input and selects the appropriate operators to automatically orchestrate an initial pipeline (Section 4.1). It then executes the pipeline, making interactive adjustments based on intermediate results (Section 4.2), until the entire process is completed and the final result is obtained.

4.1 Automated Pipeline Orchestration

If the LLM could optimally orchestrate the pipeline for the query on its own, using the provided operators, we would achieve a pipeline that is both highly accurate and efficient. However, this pipeline orchestration task exceeds the current reasoning capabilities of LLMs, as the optimal orchestration requires complex, multi-step reasoning or "chain of thought" processes [13], as well as the data information in data lakes to orchestrate efficient pipelines. Although LLMs can identify the necessary operators, they struggle with correctly decomposing queries and orchestrating operators on its own due to challenges in operator dependency recognition, ordering, and physical operator selection.

To address these limitations, we decompose the complex orchestration task into three simpler steps that current LLMs are capable of: (1) generating candidate chain pipelines; (2) rewriting the pipelines into Directed Acyclic Graph (DAG) structures; (3) integrating the DAGs into a single comprehensive pipeline. Steps (1) and (2) are achieved through tailored prompts given to the LLM, while step (3) is accomplished using manually designed rules. Figure 4 provides a concrete example of this process, which we will explain next.

Note: Although LLMs cannot directly output the optimal pipeline, the above decomposed steps simplify the original complex pipeline orchestration problem into easy, solvable sub-problems, which makes LLMs possible to well conduct pipeline orchestration by solving each step and offering reflection at every decision point.

Generate Initial Pipelines. First, we use the query and operator explanations as input and instruct the LLM to generate candidate pipelines with chain structures. For instance, the Chain Pipelines in Figure 4 are the generated candidate pipelines. The generation step uses the following prompt format: *Query: {Query}. The operators for planning the workflow are as follows:{Operator Explanations}. Please provide some effective and efficient pipelines consisting only of the operators above for answering the query and explain each pipeline.*

In this step, multiple candidate pipelines are generated by calling the LLM multiple times, leveraging the randomness in the LLM generation process.

Rewrite Pipelines. Although the chain-structured pipelines generated in the previous step can answer the query, they are inefficient due to their linear execution. To improve efficiency, we rewrite the pipelines as DAGs, allowing parallel execution of operators where possible. We propose to rewrite by the LLM using the following prompt: *Certain operators can be executed in parallel to enhance efficiency. Please rewrite the pipeline as a directed acyclic graph (DAG) to maximize parallel execution without compromising the result quality.*

Combine Pipelines. Different pipelines explore the answer to the query in different ways. Since we cannot predict which way will yield the most accurate results, we combine their results to improve accuracy. Specifically, we add an `Explain` operator to the final step of each DAG pipeline to output the result with an explanation. As shown in Figure 4, these outputs are then fed into an `Integrate` operator, which consolidates the results of multiple DAGs to produce the final answer. This approach is inspired by the Best-of-N [31] method that selects the best from N generations. The difference is that the explanations here can help LLMs better integrate the answers

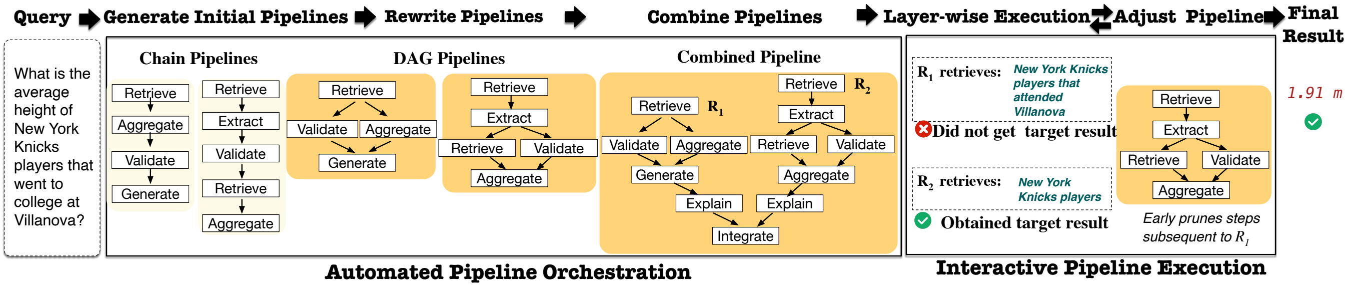


Figure 4: Framework of AOP that answers queries accurately with automated pipeline orchestration and interactive pipeline execution.

with higher accuracy, since the LLM can judge the reasonableness of each answer based on the explanation to filter out unreasonable answers and focus on the answers more likely to be correct.

4.2 Interactive Pipeline Execution

Layer-wise Execution. After automated pipeline orchestration, AOP executes the pipeline layer by layer from top to bottom following its topological structure. This ensures that all operators with completed prerequisites are executed simultaneously, thus maximizing parallelism and efficiency. For example, the first operators to execute in Figure 4 are the Retrieve operators R_1 and R_2 that attempt to obtain external knowledge from different perspectives to assist in answering the query.

Pipeline Adjustment. Each time a layer of operators is executed, intermediate results are evaluated to determine if adjustments to the pipeline are necessary. For example, if a Retrieve operation fails to obtain the desired information or if a Validate operation identifies a problem, subsequent operations are halted. The pipeline is then adjusted using the method described in Section 4.1, with the gathered execution results used as inputs to the LLM to guide the pipeline generation. For instance, as shown in Figure 4, R_1 fails to obtain the target information while R_2 succeeds. Based on these results, AOP adjusts the pipeline by early stopping further execution of the steps subsequent to R_1 and focusing on the pipeline of R_2. This dynamic adjustment ensures that AOP explores the answer in the correct direction, thus obtaining higher accuracy and efficiency.

Upon completing the entire pipeline, an accurate answer to the query can be obtained based on the gathered information and reasoning. For example, in Figure 4, the correct answer, *1.91 m*, supported by the retrieved information, can be generated.

5 QUERY OPTIMIZATION

To improve the efficiency of the orchestrated pipeline without compromising accuracy, AOP employs a cost-based model to select the efficient execution pipeline prior to execution (Section 5.1). Besides, during execution, AOP employs additional optimizations to further improve efficiency (Section 5.2).

5.1 Cost-Based Optimization

Cost Model. AOP employs a cost model for the operators to improve execution efficiency, especially for costly operations, such as Scan. Costs for both pre-programmed and LLM-based operators can be modeled as functions based on their inherent computational complexity and input size (cardinality). While the functional relationship between cost and input cardinality is known, *e.g.,* pre-programmed operators have fixed complexity and LLM-based operator costs scale approximately linearly with output size [5], specific parameters of these functions require to be determined. In AOP, these parameters are tuned using a sample workload.

Cardinality Estimation. Estimating cardinality is crucial for cost modeling, especially for data lake analytics or unstructured data analytics, where intermediate results can vary significantly in size. Due to the complexity of semantic-related operators and lack of schema for unstructured data, traditional cardinality estimation methods in relational databases such as histograms or learned data distribution-based methods [34, 35] cannot be directly adopted. Instead, AOP uses uniform sampling (*e.g.,* 1% of the data), and executes the query over the samples to approximate selectivity, thereby estimating the cardinality of results with a relatively small time consumption.

Cost-based Pipeline Optimization. The AOP optimizer estimates the computational cost for multiple candidate pipelines and predicts intermediate data cardinality to select the most efficient option. Additionally, this cost model enables AOP to reorder operators within the pipeline to further improve efficiency without affecting result accuracy.

5.2 Execution Optimization

Unlike relational databases, following a certain execution logic will always produce a correct result. The complex data analytics involving semantics in AOP face execution failures, require different or multiple computation devices for different operators and is constrained by the context length limit of LLMs. Except for cost-based optimizations before execution, AOP employs additional optimizations during execution that include reducing delays from retrieval failures (prefetching), exploiting parallelism to reduce latency (parallel execution), and handling LLM context constraints (context management).

Prefetch. Retrieval failures, *i.e.,* cases where the Retrieve steps do not yield the desired results, can severely degrade efficiency due to the delay caused by repetitive planning and retrieval. Since retrieval failures are unpredictable, we propose a prefetching mechanism to mitigate their impact by utilizing idle computational resources to fetch potentially useful information for subsequent steps in the background. Consider the example query in Figure 2, which has several retrieval options:

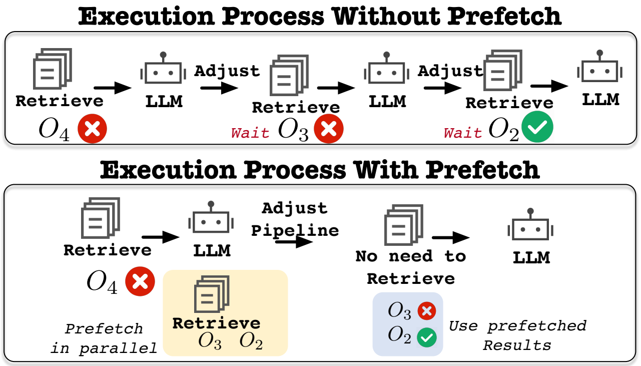


Figure 5: An example of execution with and without prefetch.

O_1: Villanova NBA players
O_2: New York Knicks players
O_3: New York Knicks players that attended Villanova
O_4: Average height of New York Knicks players that attended Villanova

These queries vary in granularity; for instance, O_4 directly targets the desired answer, while others are broader and require further filtering and computation. When manually orchestrating the pipeline, one can leverage domain knowledge to select the most appropriate queries and avoid ineffective ones. In contrast, automatically generating pipelines with LLMs involves trial and error. This often leads to redundant planning and inevitable retrieval failures, consuming significant time. To reduce time consumption, we propose prefetching additional candidate queries after each retrieval process.

Specifically, when the `Retrieve` operator generates a retrieval query, we let it simultaneously generate multiple candidate queries. Once the primary query is processed and its result is returned, the pipeline proceeds while additional retrieval queries are processed in the background when computational resources permit. This proactive retrieval ensures that if the initial result is unsatisfactory and the pipeline is adjusted, pre-fetched results may be already available without additional time consumption, thus minimizing delays.

Figure 5 illustrates this technique. After retrieving O_4, the system synchronizes the prefetching of O_3 and O_2 while processing the result of O_4. If the result of O_4 is inadequate and the adjusted pipeline requires retrieval results for O_3 and O_2, the pre-fetched results can be immediately utilized, thus reducing latency. Therefore, by employing prefetching, we significantly reduce the delay associated with retrieval failures, thereby improving overall efficiency.

Parallel Execution. Pipelines can be complex and involve numerous steps, often resulting in long execution latency. However, many parts of the Directed Acyclic Graph (DAG) structure pipeline can be executed in parallel, significantly reducing the overall execution time. For example, in Figure 4, both the `Retrieve` operations and their subsequent operations can be performed simultaneously due to their independence. This parallel execution reduces the end-to-end execution time, albeit at the cost of utilizing more computational resources concurrently. Consequently, converting the original serial chain structure pipeline into a DAG form (Section 4.1) is essential for optimizing efficiency through parallelism.

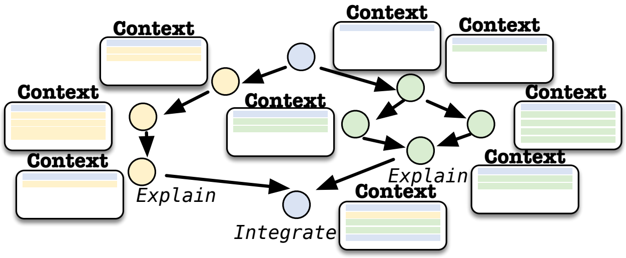


Figure 6: An illustration of context management.

Context Management. The context length of an LLM is limited, *e.g.,* Llama 3 has a context limit of only 8,192 tokens. This makes it impractical to record all pipeline steps within the context. To mitigate this limitation, we propose a strategy that greedily maintains only the context closely relevant to the current state through the `Summarize` operator executed by another LLM agent, thereby minimizing context consumption. In addition, we employ the `Explain` operator to reduce the context length when integrating multiple pipeline components. Figure 6 illustrates this approach with a concrete example. The pipeline consists of two independent parts (yellow and green) that explore the query answer without influencing the context of the other. Upon completion, the `Explain` operator compresses the context of each part, enabling the `Integrate` operator to combine the results efficiently. Without this compression, integrating the results would be infeasible due to exceeding the context limit.

6 EXTENSION TO INTELLIGENT DATA LAKE ANALYTICS

This section presents extensions of the `AOP` framework to support large-scale data analytics over heterogeneous data lakes. The unique challenge of data lakes lies in their inherent *heterogeneity*, *i.e.,* managing diverse data types at scale. We address key challenges brought by this heterogeneity, including linking across different data types, developing a unified cost model, enabling joint optimization across hybrid execution strategies (*e.g.,* SQL for structured data and pipeline execution for unstructured data), and introducing an efficient execution strategy tailored for heterogeneous data lake environments.

6.1 Linking Heterogeneous Data

A primary challenge in data lake analytics is establishing links across diverse data types, which is similar to schema/entity linking in NL2SQL [16, 20]. This is because accurate query answering requires both integration of heterogeneous information and alignment of semantically equivalent expressions across diverse data types.

However, in data lakes, many data types lack structured schemas, making direct linking infeasible. A straightforward solution is to preprocess each data type and convert them into a common format. For example, we can transform semi-structured and unstructured data into the structured format by extracting structured information (*e.g.,* entities) in preprocessing. However, this transformation inevitably results in information loss, which degrades linking accuracy.

To solve this limitation, `AOP` leverages the observation that *all types of data have literal descriptions, albeit in different formats:*

structured data has defined schema with named attributes, semi-structured data has flexible key paths, and unstructured data has textual content. Based on this observation, we propose to convert these literal descriptions into a unified form of *semantic embedding*, and then link them by measuring the similarity between embeddings. This method can be combined with the previously introduced structural data extraction technique to enhance accuracy, as they are complementary. Furthermore, since linking often occurs in real time during query execution, performance is crucial. To improve efficiency, we leverage semantic embedding indexes [25] to speed up the linking process.

6.2 Joint Optimization

Analytics in data lakes span unstructured, semi-structured, and structured data, each has distinct optimization strategies. Optimizing these data types separately results in suboptimal performance; therefore, a joint optimization strategy is essential.

Unified Cost Model. To support joint optimization, we introduce a unified cost model across data types to estimate efficiencies for candidate pipelines. The observation is that *different data types present distinct processing costs*: structured data is I/O-heavy with moderate CPU usage for filtering and joins, while unstructured data requires CPU/GPU-intensive operations (*e.g.,* embedding computation, attention computation). Semi-structured data, such as JSON or XML, adds parsing and path-based extraction costs. We capture these cost differences by defining resource-specific cost functions (I/O, CPU/GPU) for each operator, parameterized by input size and computational complexity. Before executing AOP, we fit these cost functions using sample workloads for accurate cost estimation.

Moreover, intelligent data lake analytics may involve multiple optimization goals, such as minimizing latency, maximizing accuracy, or minimizing LLM usage. The cost model can be adjusted to meet each specified optimization criterion effectively, or a compound cost model, *i.e.,* a weighted average of different sources, can be used to achieve a multi-objective optimization strategy.

Hybrid Optimization. With the unified cost model, we optimize execution pipelines by setting a threshold for each resource and reordering operators to maximize concurrency while not exceeding resource limits. Beyond cost, we prioritize pipeline processing between operators to reduce operator delay in materialization processing, enabling downstream operators to start processing partial outputs from preceding operators. Given the variability of semantic operators, the pipeline processing feasibility of some operators is challenging to determine. Therefore, after generating a pipeline, we utilize LLMs to evaluate the feasibility of each semantic operator in the pipeline, allowing AOP to output an optimized pipeline annotated with pipeline processing feasibility for each operator.

6.3 Hybrid Execution

Data Transformation. Executing pipelines across structured, semi-structured, and unstructured data involves integrating SQL, parsing tools, and NLP models into a unified workflow, which requires frequent transformations across data types. To improve the effectiveness of these transformations, we introduce an optimized `Transform` operator to convert data formats, *e.g.,* transforming structured results into text descriptions for NLP models.

Pipeline Adjustment. Processing heterogeneous data types increases the likelihood of execution failures, often due to misalignments across data formats. For instance, processing structured data based on unstructured intermediate results often requires NL2SQL transformations, which need the alignment between natural language expressions and structured table content. To improve the accuracy of such cross-data-type operations, AOP employs a pre-alignment step before the execution. For example, distinct values in structured tables are matched to expressions in natural language queries, allowing for adjustments prior to execution. In addition, AOP allows for trials of alternative expressions to increase the pipeline robustness against misalignment errors.

7 EXPERIMENTS

7.1 Experimental Settings

Dataset. We use the widely adopted CRAG dataset [37] for experiments, which consists of real-world Question Answering (QA) tasks. Each question is accompanied by five web pages that potentially contain relevant information for retrieval. To better simulate real-world scenarios, we aggregate all documents as external knowledge for answering all queries. CRAG includes English question-answer pairs and we use the following complex question types:

(1) `Multi-hop`: Questions that require chaining multiple pieces of information to form the answer. It contains 231 queries.

(2) `Post-processing`: Questions that require reasoning or processing of the retrieved information. It contains 108 queries.

(3) `Set`: Questions that expect the answer to be a set of entities or objects. It contains 249 queries.

(4) `Aggregation`: Questions that involve aggregation of partial results. It contains 315 queries.

(5) `Comparison`: Questions that involve comparisons between entities. It contains 333 queries.

Evaluation Metric. Following [8], we use accuracy as the evaluation metric. We employ the official automatic evaluation code of CRAG [37], which utilizes rule-based matching and GPT-4 assessment to verify answer correctness. Additionally, we report the average end-to-end latency for efficiency evaluation. To measure the cost of each method, we report the average token consumption amount for each method.

Baselines. We compare AOP with several state-of-the-art RAG methods that do not modify the LLM model. These baselines follow fixed pipelines, while AOP interactively determines the appropriate pipeline based on queries and intermediate results.

(1) `LLM`: Directly querying the LLM.

(2) `RAG`: A simple pipeline that retrieves relevant information using the query and generates answers based on the retrieved information and the query.

(3) `IterRAG` [30]: Similar to RAG, but iteratively generates different retrieval queries. We set the number of iterations to 5.

(4) `RecurRAG` [38]: Similar to RAG, but iteratively decomposes the query and retrieves information, also with 5 iterations.

Hyper-parameter Setting. We use Llama 3.1-8B Instruct as the LLM model for all experiments. We use the Sentence Transformer model to compute 384-dimensional sentence embeddings. Sentences are segmented, treated as chunks, and their embeddings are normalized to unit vectors. During retrieval, we compute cosine similarity

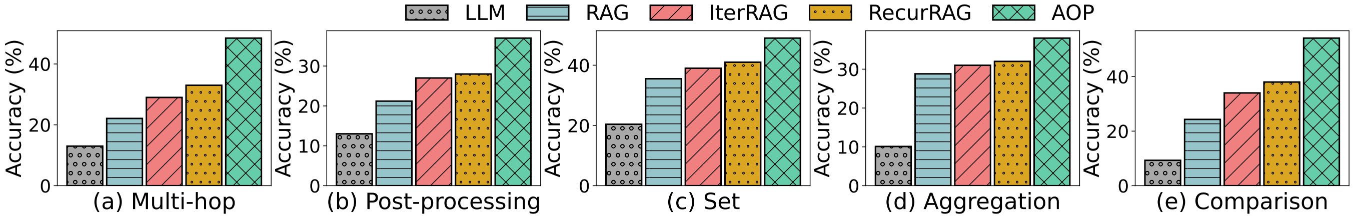
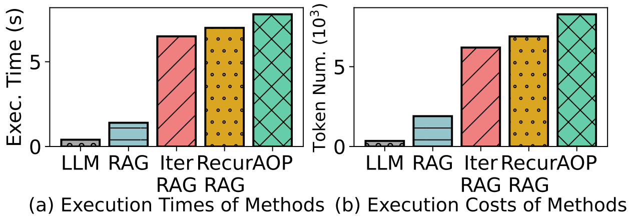
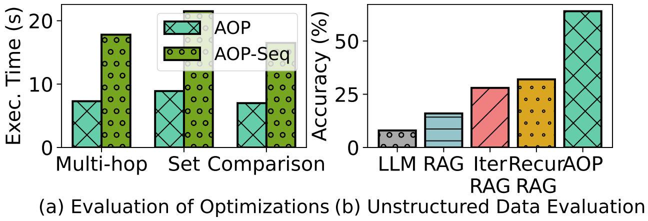


Figure 7: Accuracy of different methods for different types of queries.



(a) Execution Times of Methods (b) Execution Costs of Methods

Figure 8: Efficiency and cost of different methods.



(a) Evaluation of Optimizations (b) Unstructured Data Evaluation

Figure 9: Evaluation of: (a) query optimization; (b) unstructured data analytics.

between query embeddings and chunk embeddings to identify the most relevant chunks, selecting the top 20 based on similarity scores. **Environment.** All experiments are conducted on a Ubuntu server with an Intel(R) Xeon(R) 6242R CPU, 4 Nvidia 3090 GPUs and 256GB RAM.

7.2 Evaluation of Multi-hop Retrieval

7.2.1 Comparison of Accuracy. Figure 7 shows the accuracy of different methods on different query types. AOP consistently outperforms the other baselines, with accuracy improvement reaching up to 45%. For instance, on Comparison queries, AOP achieves an accuracy of 54.1%, while LLM, RAG, IterRAG, and RecurRAG obtain accuracies of 9.30%, 24.3%, 33.9%, and 38.1%, respectively.

The superior accuracy of AOP comes from its ability to interactively orchestrate and refine execution pipelines by leveraging intermediate results. Unlike LLM which directly queries the LLM, AOP integrates external knowledge and executes multi-step reasoning, thus obtaining higher accuracy. This capability also allows AOP to surpass RAG, which relies solely on query-based retrieval. Additionally, AOP outperforms IterRAG and RecurRAG by decomposing the original query from multiple perspectives and focusing on solvable sub-queries during execution. The interactive pipeline adjustment in AOP ensures the execution focused on operations likely to contribute to answering the query, thereby avoiding distractions from less relevant sub-queries and non-retrievable information.

7.2.2 Comparison of Efficiency. We also compare the efficiency of different methods, as shown in Figure 8(a). The results indicate that AOP achieves similar latencies to advanced RAG methods including IterRAG and RecurRAG. LLM has the shortest execution time due to its straightforward process, albeit with significantly lower accuracy than AOP. Although RAG is faster than AOP, its simplified pipeline results in less accurate outcomes. Despite AOP involving more steps than IterRAG and RecurRAG, the optimizations discussed in Section 5, such as parallel execution and prefetching, allow AOP to achieve comparable execution times.

7.2.3 Comparison of Cost. We compare the token consumption of various methods, as illustrated in Figure 8(b). While LLM has the lowest token usage due to its relatively simpler process, its accuracy is significantly lower than that of AOP. Although RAG uses fewer tokens than AOP, its static pipeline sacrifices accuracy. Despite the additional steps involved in AOP compared to IterRAG and RecurRAG, the context management optimizations detailed in Section 5 enable AOP to effectively reduce overall token usage while achieving significantly higher accuracy.

7.2.4 Evaluation of Query Optimization. In this section, we evaluate the effectiveness of the query optimization techniques proposed in Section 5. Figure 9(a) compares the execution time of AOP with and without these optimizations, where the latter (AOP −Seq) executes sequentially. The results show that AOP reduces execution time significantly compared with AOP −Seq, *e.g.,* on Multi-hop, reducing from 17.8s to 7.3s, achieving a 2.4x acceleration. This improvement is attributed to parallel execution that allows multiple steps to run concurrently, and the prefetch optimization that minimizes delays caused by retrieval failures.

7.3 Evaluation of Unstructured Data Analytics

To evaluate the unstructured data analytics capabilities, we collected 100 web pages from Sports StackExchange, converting them into plain text to standardize the data format. From this dataset, we generate 25 queries based on five manually designed templates inspired by the StackExchange Data Explorer [6]. Each template produces five queries by sampling specific values from the dataset. To enhance linguistic diversity, we use an LLM to generate paraphrased variants of each query, which are then manually verified to ensure equivalence. Ground truth answers are computed manually.

We compare AOP with baseline methods introduced in Section 7.1, with results shown in Figure 9 (b). The results demonstrate that AOP outperforms other methods by up to 52% in accuracy. This is attributed to the complex nature of these semantic analytical queries, which require aggregating information across multiple unstructured data sources and involve multi-step reasoning to arrive at the correct answer. Unlike RAG methods, which retrieve specific data segments, AOP effectively generates reasoning paths, scans necessary data and dynamically adjusts its steps based on intermediate results, achieving much higher accuracy.

8 RELATED WORKS

Retrieval-Augmented Generation (RAG). RAG integrates retrieval-based methods with generation-based models to improve the accuracy and relevance of LLM responses. RAG methods [8, 11, 30] typically involve chunking documents, indexing documents, retrieving documents most relevant to the given query, and generating responses conditioned on both the query and retrieved information via LLMs. Various methods, *i.e.,* execution pipelines, have been proposed to optimize the effectiveness of RAG, such as segmenting documents more fine-grained, decomposing complex queries into simpler sub-queries, and iteratively or recursively applying retrieval and generation steps. Recent RAG methods are extended to diverse data types, including knowledge graphs [10, 28], and structured tables [23], enabling LLMs to leverage heterogeneous data sources. *However, RAG is inherently limited to point lookups, where it assumes a query can be fully addressed by retrieving relevant segments [27]. This assumption is not satisfied in unstructured data analytics, which demands complex semantic operations, such as aggregating across heterogeneous documents and multi-step complex reasoning over intermediate results.*

LLM Programming Framework. Modern LLM programming frameworks are widely used for developing advanced query pipelines, offering user-friendly declarative methods to orchestrate workflows. For example, LangChain [1] and LlamaIndex [4] respectively introduce LangChain Expression Language (LCEL) and QueryPipeline abstraction to simplify pipeline composition. These abstractions enable the manual orchestration of pipelines that incorporate RAG, prompts, and tool-calling. *However, these frameworks lack semantic data analytical operators, automated pipeline orchestration, requiring manual configuration and optimization. These processes require significant expertise and are time-consuming, particularly when handling large-scale data and complex queries.*

Native-COT Reasoning of LLMs. During the submission of this paper, a new paradigm has emerged to improve the complex reasoning ability of LLMs by incorporating native chain-of-thought (NCoT) processes directly within model architectures [22]. A key example of this approach is OpenAI's o1 model [3]. NCoT allows LLMs to deep think through step-by-step reasoning before generating responses. However, LLMs still face limitations when applied to complex queries over heterogeneous data sources. First, such analytics are time-consuming, while effective query planning demands a thorough understanding of the underlying data, which LLMs inherently lack. Moreover, the NCoT training requires process reward models (PRM) to assign intermediate rewards during the reasoning process [17, 22], which are costly to obtain [29, 39] and the detailed multi-step reasoning paths specific to data analytics queries remains largely unavailable.

Agentic LLM Systems. LLM-based agentic workflows employ multiple LLMs working together to tackle complex tasks [21, 36, 41]. In these workflows, agents are similar to basic data analytics operators, such as `Filter` and `Scan`, in that each is designed for tackling specific tasks. While manually designed agentic systems yield high effectiveness, they are costly to develop and maintain [15, 40]. However, automatic agentic systems [14, 40] fall short of effectiveness due to limited human expertise. To address this, we propose predefining operators—given their stability in data analytics—to ensure accuracy. Our system is designed with flexibility, allowing new operators to be added as new functionalities arise.

LLMs for Data Management. NL2SQL methods enable natural language queries over structured data [12, 16, 20]. However, their reliance on strict syntactic and schema constraints limits their applicability to unstructured data. Recent advancements in LLMs have expanded the possibilities for querying large, complex unstructured datasets [7, 9, 18, 19, 24, 27, 33]. `ZENDB` [18] indexes documents based on structural templates, but its template dependency limits flexibility across general document formats. `Evaporate` [7] extracts tables from documents using LLM-generated code and synthesized rules but it is constrained by its SQL-based analytics, limiting the depth of semantic analytics on unstructured data. `LOTUS` [27] transforms unstructured data into tables through LLMs for semantic queries, though it requires manual pandas-like code for analytical pipeline orchestration. `PALIMPZEST` [19] optimizes performance for large-scale tasks like information extraction and multimodal analytics, but user-defined schemas and logical plans are required. In contrast to these approaches, `AOP` allows users to perform analytics directly via natural language queries across heterogeneous data.

9 CONCLUSION AND FUTURE WORK

In this paper, we propose `AOP`, the first systematic framework for automated pipeline orchestration using LLMs to address complex queries over heterogeneous data lakes. `AOP` extracts standard semantic operators, constructs optimized semantic execution pipelines tailored to the input query, and employs interactive execution with real-time adjustments to ensure high accuracy. By integrating advanced query optimization techniques, `AOP` achieves significant improvements in both accuracy and efficiency over state-of-the-art baselines. Extensive experiments demonstrate that `AOP` effectively handles challenges such as multi-hop semantic data retrieval and linking, multi-step logical reasoning, and multi-stage semantic data analytics across heterogeneous data formats, achieving up to 45% accuracy improvements on challenging datasets.

We hope that this work will pave the way for numerous research opportunities in the field of intelligent data lakes. For future work, we plan to explore several key directions. First, we aim to improve the cost estimation of semantic operators by leveraging advanced cardinality estimation techniques, enabling more precise query optimization, particularly for pipelines involving unstructured data. Second, we intend to fine-tune LLMs to specialize in pipeline orchestration, incorporating advanced reasoning capabilities such as native chain-of-thought. This will allow LLMs to inherently generate and optimize pipelines, similar to highly tuned database optimizers. Third, we will focus on enhancing intelligent data lake analytics at scale, which includes addressing challenges in linking heterogeneous data types, integrating hybrid optimization strategies, and refining cross-format data transformations to manage diverse and evolving data sources effectively.

Acknowledgment. This paper was supported by National Key R&D Program of China (2023YFB4503600), National Science and Technology Major Project of China (CJGJZD20230724093403007), NSF of China (61925205, 62232009, 62102215), Zhongguancun Lab, Huawei, and Beijing National Research Center for Information Science and Technology (BNRist). Guoliang Li is the corresponding author.

REFERENCES

[1] Langchain. https://www.langchain.com/.

[2] Langchain documentation. https://python.langchain.com/v0.2/docs/concepts/.

[3] Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/.

[4] Llamaindex. https://www.llamaindex.ai/.

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

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

[7] 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. *Proc. VLDB Endow.*, 17(2):92–105, 2023.

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

[9] Z. Chen, Z. Gu, L. Cao, J. Fan, S. Madden, and N. Tang. Symphony: Towards natural language query answering over multi-modal data lakes. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023.

[10] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, and J. Larson. From local to global: A graph RAG approach to query-focused summarization. *CoRR*, abs/2404.16130, 2024.

[11] Y. Gao, Y. Xiong, X. Gao, K. Jia, and et al. Retrieval-augmented generation for large language models: A survey. *CoRR*, abs/2312.10997, 2023.

[12] Z. Gu, J. Fan, N. Tang, S. Zhang, Y. Zhang, Z. Chen, L. Cao, G. Li, S. Madden, and X. Du. Interleaving pre-trained language models and large language models for zero-shot NL2SQL generation. *CoRR*, abs/2306.08891, 2023.

[13] S. Hao, Y. Gu, H. Ma, J. J. Hong, Z. Wang, D. Z. Wang, and Z. Hu. Reasoning with language model is planning with world model. In H. Bouamor, J. Pino, and K. Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 8154–8173. Association for Computational Linguistics, 2023.

[14] S. Hu, C. Lu, and J. Clune. Automated design of agentic systems. *CoRR*, abs/2408.08435, 2024.

[15] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

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

[17] H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K. Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[18] Y. Lin, M. Hulsebos, R. Ma, S. Shankar, S. Zeighami, A. G. Parameswaran, and E. Wu. Towards accurate and efficient document analytics with large language models. *CoRR*, abs/2405.04674, 2024.

[19] 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.

[20] 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.

[21] Z. Liu, Y. Zhang, P. Li, Y. Liu, and D. Yang. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization. *CoRR*, abs/2310.02170, 2023.

[22] L. Luo, Y. Liu, R. Liu, S. Phatale, H. Lara, Y. Li, L. Shu, Y. Zhu, L. Meng, J. Sun, and A. Rastogi. Improve mathematical reasoning in language models by automated process supervision. *CoRR*, abs/2406.06592, 2024.

[23] Z. Luo, C. Xu, P. Zhao, X. Geng, C. Tao, J. Ma, Q. Lin, and D. Jiang. Augmented large language models with parametric knowledge guiding. *CoRR*, abs/2305.04757, 2023.

[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] B. Peng, Y. Zhu, Y. Liu, X. Bo, H. Shi, C. Hong, Y. Zhang, and S. Tang. Graph retrieval-augmented generation: A survey. *CoRR*, abs/2408.08921, 2024.

[29] N. F. Rajani, B. McCann, C. Xiong, and R. Socher. Explain yourself! leveraging language models for commonsense reasoning. In A. Korhonen, D. R. Traum, and L. Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4932–4942. Association for Computational Linguistics, 2019.

[30] Z. Shao, Y. Gong, Y. Shen, M. Huang, and et al. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. *EMNLP*, 2023.

[31] N. Stiennon, L. Ouyang, J. Wu, D. M. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano. Learning to summarize with human feedback. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

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

[33] M. Urban and C. Binnig. CAESURA: language models as multi-modal query planners. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024.

[34] J. Wang, C. Chai, J. Liu, and G. Li. FACE: A normalizing flow based cardinality estimator. *Proc. VLDB Endow.*, 15(1):72–84, 2021.

[35] J. Wang, C. Chai, J. Liu, and G. Li. Cardinality estimation using normalizing flow. *VLDB J.*, 33(2):323–348, 2024.

[36] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.*, 18(6):186345, 2024.

[37] X. Yang, K. Sun, H. Xin, Y. Sun, and et al. CRAG - comprehensive RAG benchmark. *CoRR*, abs/2406.04744, 2024.

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

[39] E. Zelikman, Y. Wu, J. Mu, and N. D. Goodman. Star: Bootstrapping reasoning with reasoning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[40] J. Zhang, J. Xiang, Z. Yu, F. Teng, X. Chen, J. Chen, M. Zhuge, X. Cheng, S. Hong, J. Wang, et al. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024.

[41] X. Zhou, G. Li, Z. Sun, Z. Liu, W. Chen, J. Wu, J. Liu, R. Feng, and G. Zeng. D-bot: Database diagnosis system using large language models. *Proc. VLDB Endow.*, 17(10):2514–2527, 2024.